Development of a Language Workbench: Experience Report from the Sirius Project

Anonymous Author(s)

Abstract

Sirius Desktop is an Eclipse-based language workbench used for producing numerous industrial graphical modeling workbenches. From this valuable modeling experience has stemmed rationales that the Obeo company considered for starting a new project, Sirius Web, a web-based language workbench, as the successor of Sirius Desktop. This paper is an experience report that details those rationales that drove the development of Sirius Web. Those rationales are based on good practices and lessons learned while developing a language workbench for modeling languages. We specifically focus on: the rationales related to modeling and usability features; their impact on the development life cycle of tool-supported modeling languages; the software architecture of the language workbench and its resulting modeling environments. Concrete examples illustrate the detailed rationales. We also discuss alternative approaches Obeo considered. Those rationales aim to i) assist language workbench developers in making design choices; and ii) identify open questions for the software language engineering community.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**.

Keywords: language workbench, language engineering, domain specific language, low code

ACM Reference Format:
Anonymous Author(s). 2023. Development of a Language Workbench: Experience Report from the Sirius Project. In *Proceedings of ACM SIGPLAN International Conference on Software Language Engineering (SLE'23).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Engineers and scientists, experts in a particular field, broadly use Domain-Specific Languages (DSLs) to perform tasks specific to their work. In a way, DSLs are interfaces between domain experts and their engineering problems [3, 6]. In practice, domain experts handle DSLs in dedicated modeling environments. Developing such environments from scratch is expensive. Language workbenches, such as Sirius Desktop and Sirius Web, aim to simplify the process of creating such environments by providing an environment in which language designers can create their own modeling environment.

Sirius Web is a language workbench for developing modeling languages and succeeds to the Eclipse-based Sirius Desktop. This paper discusses the main Sirius Web rationales based on lessons learned from the development of multiple industrial modeling workbenches using Sirius Desktop. We specifically focus on the rationales related to features and usability, their impact on the development life cycle of tool-supported modeling languages, and the software architecture of the language workbench and resulting modeling environments. We detail each rationale with motivations, illustrations, lessons learned, and when adapted with open questions. Those rationales aim to assist at i) helping language workbench developers in making design choices; and ii) identify open questions for the software language engineering research community.

Section 2 first introduces Sirius Web to then detail the rationales (Sections 3 to 5). Finally, related work and future research opportunities are discussed (Sections 6 and 7).

2 Sirius Web Overview

Sirius Web is both a language workbench and the platform on which the designed languages operate. Similar to Sirius Desktop, it enables the design of graphical modeling languages and their dedicated modeling environments directly from a web browser. Sirius Web is a cloud-native application with a web browser client to use it. It also supports live concurrent editing on a same model. We distinguish three roles in Sirius Web: the *language designer* called *studio maker* in Sirius Web who is the person designing a modeling workbench; the *end-user* who is the person using the produced modeling workbench, called *studio* in Sirius Web. We define a modeling workbench as the combination of one or more modeling languages with the representation(s) that end-users can use to visualize and edit models. A representation in Sirius Web is both a concrete syntax for a language and a set of tools to interact with the abstract syntax through the concrete syntax.

Sirius Web offers two meta-languages, allowing language designers to create their own customized modeling workbench: the *Domain DSL* and the *View DSL*. The Domain DSL,

SLE'23, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

inspired by Ecore [10], enables the design of the language abstract syntax in the form of a metamodel through a class diagram representation. The View DSL enables the configuration of representation types, such as diagram and form, for abstract syntaxes. With the View DSL, the language designer can choose one or several representation types and design from the representation type specific elements (*e.g.,* node or edge for diagrams) the concrete syntax. To describe the mapping between the concrete syntax elements and the abstract syntax elements, the language designer use in the View DSL an embed navigation language inspired by OCL. Finally, a language designer can also configure through the View DSL: the style of the graphical elements (*e.g.,* node color, arrow type); the tools used to interact with the model through the associated concrete syntax elements, using pre-defined operations (*e.g.,* instantiate/delete a model element, set a value for an attribute of a model element). For instance with an UML class diagram representation, the language designer can create a tool associated to the node representing a class. They can configure the tool to create a new attribute for this class. The modeling workbenches are interpreted by Sirius Web and made available to the end-user automatically and seamlessly.

3 Feature and Usability Concerns

This section details the rationales related to features and usability concerns.

Rationale 1: Proposing different level of development interfaces to create a modeling workbench.
Why. Sirius Web offers different levels of development interface in the language workbench, ranging from low-code facilities in the browser to a Java-based API. This different levels aims to effectively align with the expertise of the language designer and accommodate the complexity of the modeling workbench features being developed. Using the Java API, the language designer can contribute to extend the higher levels, for instance the low-code level. For instance, from the Java-based API a language designer can add new representation types to be configured later on.
Illustration. Papyrus Web UML [1] is an industrial modeling workbench currently under development made with Sirius Web. The persons designing this modeling workbench are Java developers so the Java API enables them to work with the tools they are used to, for instance Java IDE, even if the features of this modeling workbench can be developed with the low-code approach.

> **Lessons learned.** Offering different levels means better adaptation to the expertise of language designers and modeling workbench complexity.

Rationale 2: Enabling multi-representations.
Why. Sirius Web proposes different representation types (diagrams, forms) that a language designer can configure to create concrete syntaxes. Therefore, an end-user can choose to instantiate one or several of these configured representation types for a model. In this way, a model can have as many representations as necessary. This permits heterogeneity: in time, where different stakeholders manipulate a same model using different representations; in space, where a same stakeholder operate on the same model but for different activities that imply different representations. While the Java API does not assume anything about the representation types, the main goal being to integrate them in the Sirius Web lifecycle. Thanks to that, it is easier to create very different representation types, for instance using Google Map[1] or the Monaco text editor[2].
Illustration. EcoreTools[3], a modeling workbench for Ecore, provides a table representation to review and edit all the model elements documentation, while being consistent with the diagram notes displaying such documentation on the class diagram representation. This table representation eases the documentation reviewing process to make sure everything is documented.

> **Lessons learned.** Enabling multi-representations for a model is important to support different viewpoints on the model addressing the various activities and stakeholders while this is not possible to provide all representation types a priori. Language workbench must provide a way for language designers to easily extend or even create new representation types.

> **Open questions.** Each time a language designer wants to add a new representation type they have to create it from scratch even if the representation is close to an existing one. This is the case, for instance, with an UML class diagram and a class diagram with only inheritance relations displayed where the latter one is the former one without some relations. Sirius Desktop provides several reuse and representation extension mechanisms. Sirius Desktop also lets the language designer creates groups of certain mappings, styles and tools called layer that the end-user can activate or deactivate.

Rationale 3: Bootstrapping the language workbench.
Why. In Sirius Web, the languages designers use the same environment to define new modeling workbenches that the environment the end-users will use. The proximity between the language workbench and the modeling workbenches makes the communication between the language designers and the end-users easier when it comes to develop the modeling workbench. In the same way, bootstrapping the language workbench forces the language workbench developers to be language designers. Indeed, the Domain DSL

[1] https://www.youtube.com/watch?v=mcJUuW1c4wU
[2] https://www.youtube.com/watch?v=t-BISMWMtwc
[3] https://eclipse.dev/ecoretools/

and the View DSL are built using a dedicated Sirius Web language workbench, stressing the Sirius features.

Illustration. The View DSL and the Domain DSL of Sirius are developed in a Sirius modeling workbench, *i.e.,* the Sirius Web language workbench. Sirius language designer thus test, as both end-users and Sirius testers, all the features a modeling workbench can provide.

> **Lessons learned.** The bootstrapping creates bridges between the different roles: language workbench developers need to think as language designers; language designers can easily try their modeling workbench as an end-user; end-users as they become more advanced can become language designers to contribute to the modeling workbench they use.

Rationale 4: Providing a specifically tailored navigation language.

Why. AQL is the navigation language Obeo developed to configure the mapping from the abstract syntax elements to the representation type elements in the View DSL. This permits to bridge an arbitrarily complex gap between the concrete syntax and the abstract syntax. AQL is inspired by OCL [7] and has a very similar syntax, with the following specific difference: AQL has a static typing system, as OCL, but the type system checking is separated from the execution phase. The motivation behind this choice is to enable the type system checking on demand: enabled for validation at design time, for the language designer; disabled in the modeling workbench, to avoid overhead and executing AQL faster. AQL also supports union types in order to not falling back on a top type as *Object* when a variable can have different types.

Moreover, `null` or unsetted values are ignored at run time instead of making the query fail: having `null` or unsetted values in models is a classical situation observed during the incremental creating of models, that are usually not valid until fully created. Finally, AQL can be extended with Java code through Java services (see rationale Rationale 5).

Illustration. With Sirius Desktop, Obeo tried several kinds of navigation languages: concise and dynamically typed (Acceleo2[4]), that were complex to maintain; general-purpose language (Java) and OCL, that were too verbose and their static strong typing system not flexible enough.

> **Lessons learned.** AQL has evolved over time and reached an expressiveness that complies with the actual needs for developing the modeling workbench addressed by Sirius in the past. Key factors are: fast evaluation, smart validation, union types and similarity with OCL.

Rationale 5: Extending the navigation language to manage more complex mappings.

Why. As described in Rationale 4, AQL is designed around a small core language that any Java programmer can extend to extend AQL capabilities with Java services.

Illustration. An example concerns how to implement specific services to improve the expressiveness of AQL mappings. A language designer can create there own types to be used in AQL mappings. For example[5], a type *DescriptiveStatistics* is defined in Java and used in AQL mappings to enable the use of means, variances, and others statistical computation from EMF models.

> **Lessons learned.** By Extending AQL with Java services, language designers can benefit from the expressiveness of a general-purpose language. This may raise security concern as the language designer can execute the code they want. In practice, the language designer needs to access to the server code to register those services, which mitigates this problem. By using Java, the language designer lose the tooling they have with the AQL languages, for instance validation. This is not a real problem since they are power users.

> **Open questions**. In a low-code approach, how to give language designers, non-expert in programming languages such as Java, the ability of expressing more complex mappings with the concerns of expressiveness, ease of use and security? How to find a trade-off between these concerns?

4 Development Lifecycle Concerns

This section details the rationales related to the development lifecycle of a modeling workbench.

Rationale 6: Uncoupling abstract syntax and concrete syntax development lifecycles.

Why. In Sirius Web, the development of the abstract syntax and concrete syntax are two different activities with their own lifecycle. This rationale follows the classical separation of concerns between *views* and *models* as widely adopted in the human-computer interaction domain (the MV* pattern). Thanks to that, a language designer can create a new modeling workbench including a new representation for an existing abstract syntax of another modeling workbench. However, this requires the co-evolution of the abstract syntax and the concrete syntax: each time the language designer modifies the abstract syntax, they have to modify the concrete syntax and vice versa.

Illustration. In many cases the abstract or concrete syntaxes are already provided. They can be standards (*e.g.,* UML and its diagrams). A company may also want to reuse a legacy syntax (abstract or concrete). Uncoupling abstract and concrete syntaxes helps in dealing with both cases.

[4] https://eclipse.dev/acceleo/

[5] https://cedric.brun.io/eclipse/news-on-the-aql-front/

Lessons learned. Uncoupling these two lifecycles enables more flexibility to maintain a modeling workbench over time. A language designer can change the concrete syntax without affecting the abstract syntax or co-evolves them according to end-user feedback or standard updates.

Open questions Uncoupling allows co-evolution between the abstract syntax and the concrete syntax. But how to co-evolve the modeling workbench and the models produced with it?

Rationale 7: Enabling seamless modeling workbench deployment.
Why. Sirius Web proposes different ways to deploy a modeling workbench. The main one is automatic: when a language designer creates or modifies a modeling workbench, Sirius Web automatically and seamlessly deploys this modeling workbench on the current Sirius Web instance. This allows to easily develop a modeling workbench in an iterative way by switching between the language definition and a model prototype and obtain quick feedback.
Illustration. When developing a modeling workbench, a language designer may want quick feedback from end-users, in a testing or a rapid prototyping purpose. The seamless deployment of modeling workbenches eases such use cases as this shortens the distance between the language designer work and its testing by end-users.

Lessons learned. By enabling a quick deployment process, a language designer can: test their ongoing modeling workbench; provide on a regular basis end-users with new versions to test.

Open questions Currently, a language designer cannot control the lifecycle of a modeling workbench. They cannot version their modeling workbench and choose which version(s) will be deployed. In the same way, existing models created with a previous version of a modeling workbench cannot be automatically migrated. For a better user experience, the language workbench must provide language designers and end-users with a way to ease model migration.

5 Architectural Concerns

This section details the rationales related to the architecture of a language workbench and modeling workbench.

Rationale 8: Enabling model edition by interacting with the concrete syntax elements.
Why.

The software language community identified two main editing approaches: the parser-based and the projectional approaches. Sirius Web does not follow these approaches as illustrated in Figure 1. The parser approach (1) directly infers the abstract syntax from the concrete syntax, enabling less freedom in the design of the language. In the projectional approach (3), the end-user directly modifies the abstract syntax without interacting with the concrete syntax making more complex the identification of the model element to edit. In the Sirius Web approach (2), the end-user has to use a tool to interact with the model. As this tool is linked to a concrete syntax element, Sirius Web is able to identify the associated model element.

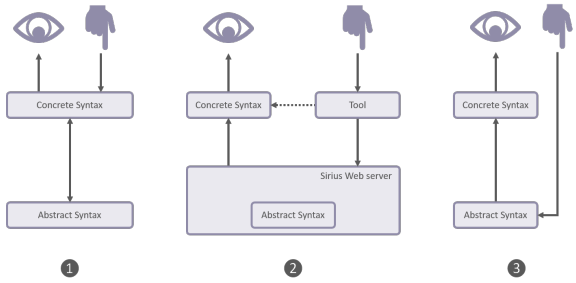

Figure 1. (1) the parser-based editing approach; (2) the Sirius Web editing approach; (3) the projectional editing approach

Lessons learned. Sirius Web does not follow a projectional approach. The Sirius Web approach does not offer as many possibilities as a projection approach, but it is sufficient to meet the needs of Obeo and its clients.

Rationale 9: Relying on a cloud-native architecture.
Why. Developing Sirius Desktop as an Eclipse-based application brings the two following concerns: i/ Delivering the application or the updates to the end-user. ii/ Running Sirius Desktop in a large environment like Eclipse RCP brings several constraints. Eclipse RCP does not give full control over its features, which hampers language designers and end-users from fully customizing their modeling workbenches. To overcome these two concerns, Sirius Web relies on a cloud-native architecture with a web frontend. Through that, application updates are centralized on servers only and Sirius Web now has its own dedicated features for building modeling workbenches.

However, adopting a cloud-native architecture raises the question of what to compute on the server and client sides. With Sirius Web, Obeo decided to place all computations about models and representations on the server, even the representation mapping evaluation: First, EMF is not available as a web library, preventing its use on the front-end side. Obeo wanted to capitalized over their 15 years' experience of using this time-honored framework and its ecosystem. Next, Obeo targets models with around one million of elements. Handling such a volume on the client side (*i.e.,* in a Web browser) would face scalability issues. Moreover, end-users handle only specific and limited parts of such huge models. One can consider models as database objects that front-ends query, as in classical Web applications. Lastly, the server is the single source of truth and Sirius Web does not have to synchronize the local state of the different clients.

Illustration. A modeling workbench can leverage EMF-based frameworks such as *EMF Compare* to perform state-of-the-art model comparisons, and provided associated services that a Sirius Web front-end can use. Updating such services would be done on the server side and thus would not require updating end-users desktops.

Lessons learned. Thanks to the cloud-native architecture, we can use the state-of-the-art tools of both stacks (back-end and front-end), and transition a rich desktop ecosystem (*e.g.,* EMF) to a cloud native application, as many technologies in the EMF ecosystem make little assumptions regarding their execution context, for instance Eclipse RCP independent.

Open questions. Placing all these computations on the server is not a perfect solution. Some client features need more information about the model itself like the auto-completion or the semantic zoom for instance. Although they concern only the client that uses these features, the current architecture means you have to refer to the server to apply them properly. This can lead to more exchange between the client and the server and so to a higher network latency. In practice, for the features currently present in the application and for those planned, this constraint is not really an issue and does not seem to have a significant impact. To manage this kind of features, providing a way to customize the location of operations or even to move these operations dynamically could be avenues to explore.

Rationale 10: Integrating modeling workbenches within other tools and vice versa.
Why. Sirius Desktop is build on top of the Eclipse and EMF ecosystem, so that it eases its integration with other tools of this ecosystem, like Xtext [4]. Integration is a major feature since domain experts may already have their own environment. Sirius Web provides mechanisms to be integrated with, or to integrate, other environments. Since the Sirius Web front-end is web-based, it can work with other environments that support webview or are web-based, *e.g.,* Eclipse RCP, Visual Studio Code, documentation websites. Moreover, as being cloud-native Sirius Web uses a standardized protocol between its front-end and its back-end, namely GraphQL over HTTP. So integration can be performed with environments that use this protocol to get data from and send data to a Sirius Web server. Finally, other web-based environments can be integrated in Sirius Web as representation type for instance.
Illustration. Obeo made several prototypes over the last few years.

Sirius Web in other applications: Obeo created a Visual Studio Code plugin linked to an existing Sirius Web server. This plugin proposes to navigate through the different projects, models and representations of the application thanks to a native VSCode side bar but also to display the representation in a webview[6]. Another example is in Jupyter Lab using the Sirius Web API to get a representation in SVG format, and to display the information from a model in a Jupyter Lab cell[7].

Other applications in Sirius Web: Obeo created new Sirius Web representations using external application, for instance with Google Map[8] to display a map and change position and zoom level directly from model elements. Another example is the integration of Langium, a language workbench for textual DSLs, to add a textual representation in Sirius Web[9].

Lessons learned. Modeling is a step in a larger engineering process. The different stakeholders already have their own environments, such as IDEs and forges like Github for software engineers. It is important to be able to integrate a modeling workbench in other environments or vice versa to match the engineering process.

Open questions. Such integration raises several open questions. First, communications between tools would be done using events. This requires the definitions of events for this specific purpose and for encompassing a large set of possible scenarios, which is challenging. Second, such events would embed chunks of models to be shared between the tools. There is a challenge for determining what model chunks to share.

6 Related Works

Research work focused on (meta-)modeling good practices. Cho and Gray [2] discussed solutions for recurrent metamodeling problems. Closely related, Pescador et al. [8] proposed patterns for building DSLs and their environments, with a focus on patterns for configuring services such environments will provide. Sánchez-Cuadrado et al. [9] designed an approach for helping the creation of metamodels. Izquierdo and Cabot [5] discussed an approach for improving the design of DSL based on collaborative work between experts. If these research works focus on improving the design of DSLs, we focus in this paper on reporting lessons learned about designed language workbenches.

7 Conclusion and Perspectives

This paper is an experience report that details ten rationales that led the development of Sirius Web, a Web-based language workbench. By explaining the motivations, the Open questions, the lessons learned, and by giving examples, these ten rationales aim at helping language workbench developers in their design choices. These rationales also identified open questions for the software language engineering research community.

[6]https://www.youtube.com/watch?v=iqYsCy26eZg
[7]https://www.youtube.com/watch?v=q9DeMS3G4TA
[8]https://www.youtube.com/watch?v=mcJUuW1c4wU
[9]https://www.youtube.com/watch?v=t-BISMWMtwc

References

[1] [n. d.]. *Eclipse Projects / papyrus / org.eclipse.papyrus-web · GitLab*. https://gitlab.eclipse.org/eclipse/papyrus/org.eclipse.papyrus-web

[2] Hyun Cho and Jeff Gray. 2011. Design patterns for metamodels. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*. 25–32.

[3] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. 2016. *Engineering modeling languages: Turning domain knowledge into tools*. CRC Press.

[4] Sven Efftinge and Markus Völter. 2006. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, Vol. 32.

[5] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2016. Collaboro: a collaborative (meta) modeling tool. *PeerJ Computer Science* 2 (2016), e84.

[6] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[7] OMG. 2014. Object Constraint Language (OCL). http://www.omg.org/spec/OCL/

[8] Ana Pescador, Antonio Garmendia, Esther Guerra, Jesús Sánchez Cuadrado, and Juan de Lara. 2015. Pattern-based development of domain-specific modelling languages. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 166–175.

[9] Jesús Sánchez-Cuadrado, Juan De Lara, and Esther Guerra. 2012. Bottom-up meta-modelling: An interactive approach. In *Model Driven Engineering Languages and Systems: 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30–October 5, 2012. Proceedings 15*. Springer, 3–19.

[10] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.