JOT
JOURNAL OF
OBJECT TECHNOLOGY

Sirius Web: Insights in Language Workbenches An Experience Report

Théo Giraudet[*†], **Mélanie Bats**[*], **Arnaud Blouin**[†], **Benoît Combemale**[†], **and Pierre-Charles David**[*]
[*]Obeo, France
[†]Univ Rennes, Inria, CNRS, IRISA, France

ABSTRACT Sirius Web is an open-source, web-based language workbench maintained by Obeo and hosted under the Eclipse Foundation. It is the successor of Sirius Desktop, an Eclipse-based language workbench used for producing numerous industrial graphical modeling workbenches in the past decades. Leveraging on this valuable experience, in this article we provide an overview of Sirius Web and document the rationales and good practices that have shaped its development. Specifically, we focus on: 1/ the rationales behind modeling and usability features; 2/ their impact on the development lifecycle of tool-supported modeling languages; 3/ the software architecture of the language workbench and the resulting modeling environments. Concrete examples illustrate both the detailed rationales and the use of the tool. We also discuss alternative approaches Obeo considered. In addition to introducing Sirius Web, this paper also aims to help language workbench developers make informed design choices for the future development of web-based language workbenches. It also identifies current open questions for the software language engineering community. Moreover, by addressing current open questions in software language engineering, this study contributes to the ongoing dialogue in the community, potentially steering future research directions.

KEYWORDS Language workbench, Language engineering, Domain-specific language, Low code

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 (Mernik et al. 2005; Combemale et al. 2016). In practice, domain experts handle DSLs within dedicated modeling environments. Developing such environments from scratch is effort expensive (Mernik et al. 2005).

Sirius Web is a web-based language workbench for developing web-based modeling languages. It follows on from the Eclipse-based Sirius Desktop. This paper introduces Sirius Web and discusses the main rationales that have been considered for developing it. Those rationales are based on documented good practices and lessons learned from the development of multiple industrial modeling workbenches using Sirius Desktop[1]. They come from the experience of several authors, including the first author, who is a developer of Sirius Web; the second author, who is the CTO of Obeo, the company behind the Sirius project; and the last author, who is the project lead for Sirius Desktop and a core developer of Sirius Web. 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 open questions when relevant. This experience report aims to assist at i) helping language workbench developers in making design choices; and ii) identifying open questions for the software language engineering research community.

Section 2 first introduces Sirius Web and the requirements we identified for developing language workbenches. Section 3 then details the rationales that aim at answering the identified

JOT reference format:
Théo Giraudet, Mélanie Bats, Arnaud Blouin, Benoît Combemale, and Pierre-Charles David. *Sirius Web: Insights in Language Workbenches:An Experience Report*. Journal of Object Technology. Vol. 23, No. 1, 2024.
 http://dx.doi.org/10.5381/jot.2024.23.1.a6

[1] https://eclipse.dev/sirius/gallery.html

requirements. Section 4 presents two concrete use cases of Sirius Web. Finally, Sections 5 and 6 discuss related work and future research opportunities.

2. Sirius Web

This section introduces the main concepts of Sirius Web. Then, we detail the requirements we identified for developing language workbenches dedicated to graphical modeling environments. These requirements motivated the development of Sirius Web, which builds upon the extensive experience gained with Sirius Desktop.

2.1. Overview of the tool

Sirius Web is both a language workbench and the platform on which the developed DSLs operate. This is an open-source project maintained by Obeo and hosted by the Eclipse Foundation[2], licensed under EPL v2 (*Eclipse Public License*). Sirius Web is a web application that can be accessed through a web browser client. The server part is developed using Java, Spring Boot, and EMF (Steinberg et al. 2008) (Eclipse Modeling Framework), and the client is made with TypeScript and React. Sirius Web uses GraphQL for defining the communication protocol between the server and the client. Similar to Sirius Desktop, it enables the development of graphical modeling languages and their associated modeling environments but now directly from a web browser. It also supports the concurrent and live editing of shared models and their graphical representations, through a centralized and optimistic collaborative mechanism. Sirius Web is agnostic of the application domain and mainly targets domain experts. It may be useful for any domain that has to build graphical DSMLs for their own concerns.

We distinguish two roles in Sirius Web: the language designer, named *studio maker* in Sirius Web, who is the person developing a modeling workbench, named *studio* in Sirius Web; and the *end user*, that uses the produced studio.

A studio is a modeling environment for one or several modeling languages. With a studio, end users edit models of languages through dedicated representations defined in the studio.

As most of the language workbenches, Sirius Web proposes artifacts over three levels of abstraction. The studio maker uses the artifacts provided by Sirius Web to develop the metamodels of a new studio. The end user uses the developed languages to create models. Figure 1 summarizes the dependencies between the different artifacts present in Sirius Web which are detailed below.

Sirius Web offers two meta-languages for defining domain and view aspects, allowing studio makers to create their own customized studios: the *Domain DSL* and the *View DSL*. There is no meta-language for operational semantics since Sirius Web does not propose facilities to develop this DSML concern. However, a language designer may use the Java API of Sirius Web to define semantics thanks to well-known EMF tools, such as GEMOC Studio (Bousse et al. 2016) or Eclipse Epsilon (Kolovos et al. 2006). The Domain DSL enables the design

[2] https://github.com/eclipse-sirius/sirius-web

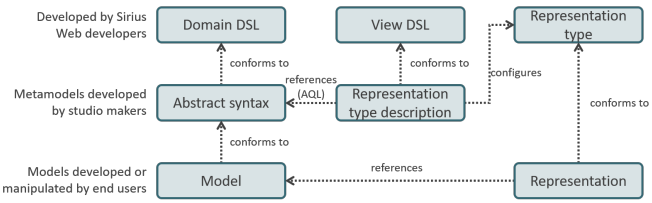

Figure 1 The dependencies between the different artifacts of Sirius Web.

of the language abstract syntax through a class diagram representation. It can be seen as a lightweight version of Ecore (Steinberg et al. 2008).

The View DSL enables the development of representation type descriptions. In Sirius Web, a *representation type* is concrete syntax concepts (*e.g.,* node or edge for diagrams) with a set of generic tools to be specialized. By default, Sirius Web proposes four representation types: diagram, form, Gantt, and kanban. A *representation type description* is a configuration that maps the abstract syntax concepts of a chosen language (*e.g.,* a class in UML) to the concrete syntax concepts of a representation type (*e.g.,* a UML class is represented with a rectangle). It also specifies the behavior of the tools for the chosen language. The representation type is language-agnostic. However, its description is used to instantiate the representation type concepts for a particular language. A representation type instance is called a *representation*. For example with a UML class diagram, the representation type used is "diagram" and the representation definition specifies that all the "class" elements should be represented by a rectangle node.

To describe the mapping between the concrete syntax elements and the abstract syntax elements, the studio maker uses from the View DSL an embedded query language called AQL (Acceleo Query Language) and inspired by OCL (Object Constraint Language) (Warmer & Kleppe 2003).

Finally, with the View DSL, a studio maker can also describe: the style of the concrete syntax elements (*e.g.,* node color, arrow type); and the tools used to interact with the model through the associated concrete syntax elements, using predefined operations (*e.g.,* instantiate/delete an abstract syntax element, set a value for an abstract syntax element's attribute). For the UML class diagram example, the studio maker can associate a tool with the node representing a class to enable the creation of a new attribute for the class.

Finally, a project in Sirius Web is a set of models that can reference each other and their representations. There are two different kinds of projects: the *basic* projects for the end user and the *studio* projects for the studio maker, enabling the use of the meta-languages.

Sirius Web also proposes to create languages with its Java APIs, enabling the use of Ecore to develop abstract syntax.

Figure 2 shows the development of a model with the Flow studio in a basic project of Sirius Web. The Flow Studio is an illustrative studio provided by default with Sirius Web, enabling the modeling of hardware architecture. The user interface of Sirius Web project is divided into three columns: the *explorer view*

(❶) that lists the different models of the project; the *representation editor* (❷) that shows a diagram representation of a model (here, a representation of the model "Robot Flow"); the *details view* (❸) that displays the attributes of the selected abstract syntax element. A studio project has exactly the same GUI with two metamodels to describe the abstract and concrete syntaxes. The first item in the model root, "Topography" (❹), is the representation itself. Here, the end user uses the representation editor to edit the displayed model. The user then uses the details view to edit a selected element. The explorer view is used to have a global view over the different models or elements of the current model not displayed inside the representation. Different aspects of this modeling task will be detailed in next sections, such as the relation between the model and the displayed representation, and the interactions with the representation to edit the model.

Sirius Web is currently used to develop some concrete studios. Most notable ones are: SysON[3] for systems engineering, Papyrus Web for UML[4], and parts of Obeo SmartEA[5] for enterprise architecture.

For more information, Obeo provides on its YouTube channel a quick overview of Sirius Web[6], and some webinars to have a detailed overview[7] or to understand how to create diagram representation[8].

2.2. Sirius Web team

Sirius Web development team is composed of seven full-time software engineers. The core members (including the second and fourth authors of this paper) of this team all have more than ten years of experience developing and supporting Sirius Desktop itself, as well as various concrete studios based on it for different domains. The development of studios are done by other employees of the Obeo company, with the help of a person from the Sirius Web team to facilitate the transfer of information and questions from one team to another. If the required features for a specific studio may be useful for Sirius Web itself, a person from the Sirius Web team may develop it to facilitate its integration in the language workbench.

2.3. Overview of the main considered requirements

Based on the experience Obeo acquired over nearly fifteen years of modeling environments development, we elicited six main requirements to fulfill for developing language workbenches dedicated to graphical modeling environments. These requirements define the scope of the lessons learned and open questions we illustrate with Sirius Web in Section 3.

Requirement 1 (Heterogeneous Modeling)**:** Systems have become more and more complex, with an increasing number of models to develop for describing the different facets of the system. Such models are usually interconnected to build the system. Language workbenches must thus be able to deal with different models in a coordinated and usable way.

Requirement 2 (Collaborative modeling)**:** Collaborative modeling has become an important feature that the modeling environments should support (Abrahão et al. 2017; David et al. 2021). In the context of heterogeneous modeling, each studio maker has different domain expertise but needs to work together to build a unique system. The studio and language workbenches should propose facilities to tool this collaboration. In other words, they should provide different stakeholders the opportunity to collaborate on the same model at the same time, and on one or several consistent models over the life cycle.

Requirement 3 (Agile language development)**:** Typically, language workbenches involve the two main roles we defined in Section 2.1: the studio maker and the end user. In practice, the end user, who is the domain expert, brings to the studio maker the domain knowledge needed to engineer the studio. The language workbench must provide a way to ease the iteration during the language development process between these two roles. Since the studio maker is also required to try out the studio they develop, this also requires a seamless process for switching between developing the studio and trying it out.

Requirement 4 (Separation of Concerns)**:** A DSL specification involves several concerns (*e.g.,* abstract syntax, concrete syntax). Following the time-honored separation of concerns (Ozkaya & Akdur 2021), studio makers must be able to design language artifacts independently, while limiting their coupling. Uncoupling the different artifacts eases the collaboration with studio makers working on different artifacts and the reuse of them. For example, a studio maker may design the abstract syntax once, to then let other studio makers work on various concrete syntaxes on the same abstract syntax and over different periods (*e.g.,* different projects over time). This also facilitates scalability.

As a consequence, a studio maker must be able to bridge the gap between far-removed abstract and concrete syntaxes.

Requirement 5 (Deployment)**:** As with any software system, easing the deployment and delivery of the resulting studios and their new versions is an important requirement. This requirement follows the DevOps approach, aiming to bridge the gap between the development and the production, so bridging the gap between studio makers and the end users. This demands a seamless delivery mechanism built-in into the language workbench.

Requirement 6 (Interoperability)**:** Developed studios usually have to interoperate with other tools of the end users. Thus, the language workbench must have facilities to communicate with other tools, such as development environments or third-party tools related to the developed DSLs.

3. Rationales

Based on the requirements detailed in the previous section, we now report the rationales the Obeo company applied for developing Sirius Web, their new language workbench dedicated

[3] https://mbse-syson.org/
[4] https://gitlab.eclipse.org/eclipse/papyrus/org.eclipse.papyrus-web
[5] https://www.obeosmartea.com/en/
[6] https://www.youtube.com/watch?v=CNS5LEr9QQE
[7] https://www.youtube.com/watch?v=in6KrDVmWmU
[8] https://www.youtube.com/watch?v=0QrK8h1Hxgo

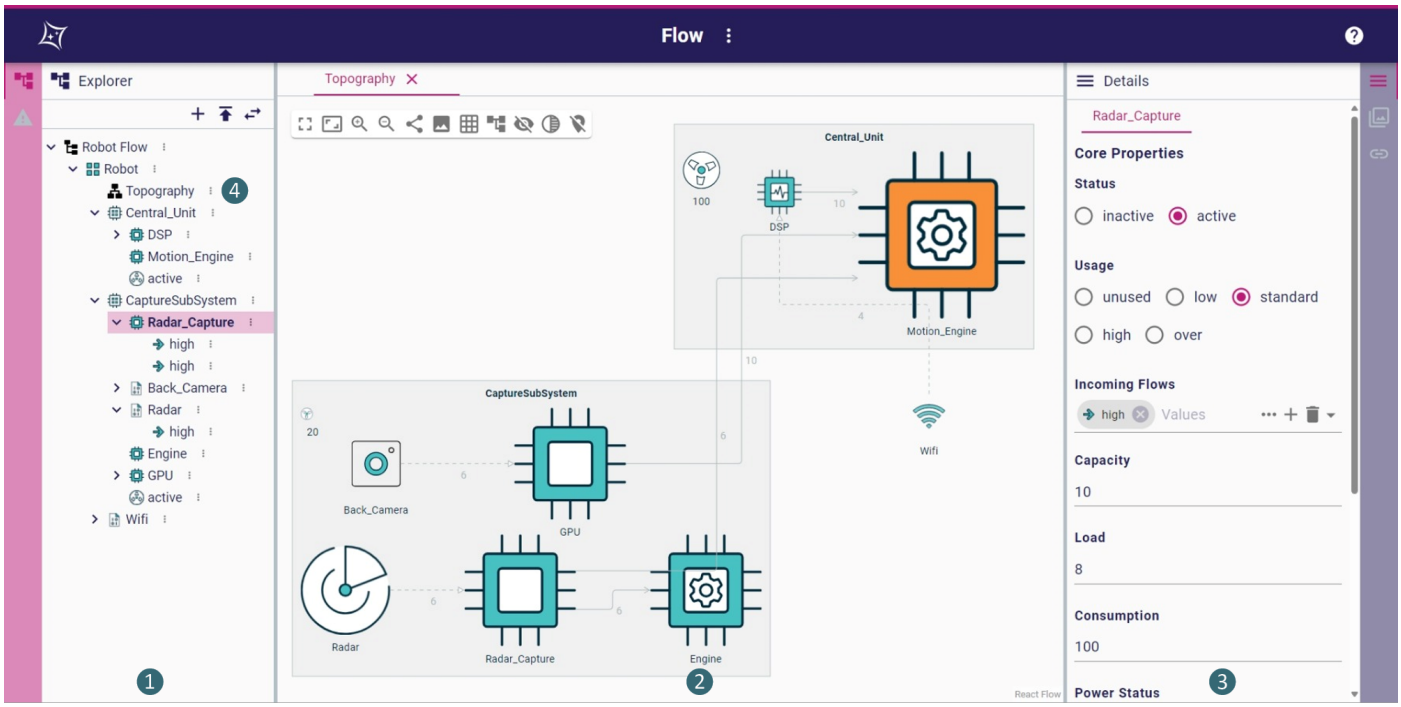

Figure 2 A flow model made with the *Flow* studio in Sirius Web.

to graphical modeling environments. We organize these rationales into three categories: language workbench development features (Section 3.1); the language development lifecycle (Section 3.2); and the architectural concern related to both the language workbench itself and the resulting studios (Section 3.3). Each rationale satisfies one or several requirements, as resumed in Table 1. Section 3.5 then discusses outcomes.

Table 1 Mapping between requirements and rationales.

Requirement	Rationales
1	2 and 7
2	2 and 5
3	6
4	4 and 5
5	6 and 8
6	9

3.1. Language workbench development features

This subsection discusses the rationales related to the specific features expected for a language workbench for developing graphical modeling environments.

Rationale 1: Adapting the language workbench to the studio maker expertise

Studio makers can come with different software development expertise, ranging from software engineers to domain experts. This is important to effectively align the development of studios with their different expertise.

Why. A language workbench should propose different development interfaces to satisfy the different possible levels of expertise in language development. It should also propose programming interfaces (APIs) to allow software engineers to access common tools (*e.g.,* the IDE or Git services) programmatically.

Illustration. Papyrus Web UML[9] is an industrial studio currently under development with Sirius Web. The first versions of the studio were developed with the low-code approach before switching to programmatic APIs since the studio makers that developed this studio are Java developers. The Java APIs enable them to work with the tools they are used to, for instance, a Java IDE. Tax Studio, an example studio presented in Section 4.1 is only developed with the low-code interface.

Implementation. Sirius Web offers three different development interfaces: the low-code facilities in the browser; a Java-based API based on the Domain and View DSLs; and a Java-based API directly based on EMF. Each interface provides different abstractions, each with varying degrees of customization possibilities. Through one specific interface, the studio maker can contribute to extending the other interfaces with new services. These interfaces can coexist: a studio can be created with the Java APIs and extended with new low-code representations.

[9] https://gitlab.eclipse.org/eclipse/papyrus/org.eclipse.papyrus-web

Lessons learned. Offering different development interfaces means better adaptation to the expertise of studio makers and studio complexity. This can also improve the learning curve of the language workbench, allowing for a studio maker to start with the low-code approach and progressively benefit from the programming interfaces.

Open questions. The ability of a language workbench to offer different interfaces of editing modes (*e.g.,* simple, advanced) raises open questions. By definition, domain experts have specific skills and backgrounds. So, predefined modes may not fit their skills perfectly. The question of adapting language workbenches, at finer-grained (*e.g.,* advanced low-code mode), to reflect domain expertise across different programming interfaces (ranging from APIs to low-code platforms) is still open.

Rationale 2: Enabling heterogeneous representations.

A language workbench should propose the development of multiple representations from different representation types (*e.g.,* diagram, form) for a DSL.

Why. As part of collaborative modeling, models tend to become more complex over time since different stakeholders can work on the same model for possibly different activities. For this reason, a given stakeholder may not be able to understand the whole model and have to keep focus on their own activities. Providing different representations for a model is a way to overcome this, such as dedicated representations are provided when needed for different stakeholders and activities (Requirement 1). The representations may not represent the whole model and can have different syntaxes, adapted to the end-users' needs. This approach is not new and follows the multi-notations principle we can find on blended modeling (Ciccozzi et al. 2019) and multi-view modeling (Cicchetti et al. 2019).

Illustration. EcoreTools[10], a studio for Ecore developed with Sirius Desktop, provides a tabular representation to review and edit the documentation of the model elements, while being consistent with the diagram notes displaying such documentation on the graphical representation (*i.e.,* class diagram). This tabular representation eases the documentation reviewing process to make sure everything is documented. Sirius Web follows the same approach but stands out from the standard.

Obeo also explored blended modeling several times, by integrating several notations. For instance, Obeo and Typefox, the company developing Xtext, proposed a prototype to integrate Sirius Desktop with Xtext to add a textual notation to Sirius Desktop (Köhnlein & Brun 2017). This prototype proposes two types of integration: two different representations (graphical and textual) to edit a model; and the embedding of Xtext inside the details view of Sirius Desktop. A prototype to integrate Langium, a language workbench for textual DSLs on the web, to Sirius Web was also developed.

Implementation. Sirius Desktop follows the multi-view approach with different viewpoints according to the IEEE 42010 standard (IEEE 2011). Sirius Web proposes different representation types (diagrams, forms) for which a studio maker can create a representation type description. Therefore, an end user can choose to instantiate the representation type according to one or several of these descriptions for a model. In this way, a model can have as many representations as necessary. Moreover, from the Java API the studio maker can create new representation types and use them from the low-code interface. Sirius Web also offers a specific meta-representation that cannot be defined by a studio maker, and which enables the aggregation of existing representations side by side. This meta-representation enables having several representations in a single view, even if these representations are not mixed.

Lessons learned. Enabling different representations for a model is important to support different viewpoints on the model, embracing the needs of various activities and stakeholders. Since the built-in representation types cannot cover all domain activities, a language workbench must provide a way for studio makers to extend or even create new representation types easily.

Open questions. Each time a studio maker wants to add a new representation type, they have to create it from scratch, even if the representation is close to an existing one. The language workbench should provide a way to create variants of specific representations or to reuse part of them. In another way, the end user should be able to adapt a representation developed by the studio maker to their intent. For instance, an end user may want to display only inheritance relations and classes involved within a UML class diagram.

Rationale 3: Bootstrapping the language workbench.

The features available in the language workbench (representations, interactive features such as completion, layouting) should also be available or even customizable for the developed studios. In other words, the language workbench should mainly be a specific studio that proposes meta-languages.

Why. Bootstrapping is a common technique when designing a language workbench. For example, Xtext (Efftinge & Völter 2006), MPS (Pech et al. 2013), Spoofax (Wachsmuth et al. 2014) are bootstrapped. For the language workbench developers, bootstrapping factorizes the development effort since the language workbench uses the same capabilities as the resulting studios, such as the editors, the way to interact with the model, *etc.* Hence, as the language workbench and the studios are really close, the communication between the studio makers and the end users is improved when it comes to developing a specific studio. Moreover, an end user can more easily take the role of studio maker if needed since the environment is similar, with just different languages. Similarly, bootstrapping the language workbench forces the language workbench developers to be studio makers, and studio makers are themselves language users of features they can provide to end users. This rationale covers Requirement 3.

Illustration. In Sirius Web, the language workbench itself is a studio using the meta-languages as modeling languages.

Implementation. In Sirius Web, the Domain DSL and the View DSL are available in a dedicated (meta-)studio, stressing the Sirius features. Sirius Web itself reuses representation types

[10] https://eclipse.dev/ecoretools/

that a studio maker can use for their own studios. For instance, the details view, a part of Sirius Web dedicated to the edition of the selected element's attributes, reuses the *Form* representation type. A Domain metamodel for its part can be edited through a diagram representation made with the View DSL. Sirius Web is not fully bootstrapped since the two meta-languages are based on Ecore and not on the Domain DSL.

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

Open questions. Bootstrapping notably aims to improve a method by using the method itself. A current challenge is how to help developers in reporting issues as they develop their language workbench. This goes beyond the classic use of manually opening an issue in the bug tracker: how to integrate such a facility within the language workbench, which would also gather and report related data (usage of the user interface, *etc.*). Moreover, such data collection could be automated during development sessions to then identify unused, misused, and highly used features of the language workbench.

Rationale 4: Bridging the gap between the concrete and the abstract syntax.

When the concrete syntax is decoupled from the abstract syntax, the studio maker needs a way to map these two syntaxes.

Why. This way may be a simple one-to-one mapping or a DSL to support bridging an arbitrarily complex gap between these two syntaxes (Requirement 4). For instance, one wants to represent an element of the abstract syntax by several concrete syntax elements or vice versa. This mapping language needs to be simple since it is used by the studio maker who is not necessarily a software engineer. Since the language is simple, it may not be able to handle complex scenarios, such as mapping depending on external information coming from a database. To enable such complex scenarios, the language workbench must provide a way to use a more general-purpose language in addition.

Illustration. An example of complex mapping is derived relations. For instance, consider a UML class diagram with packages, classes, and edges between classes of different packages. A studio may propose a specific representation to the end user to study the coupling between the different packages. This representation only includes packages and edges between them to represent the relations between classes of two different packages. Although these edges may appear to connect packages in the diagram, they do not represent relationships between the package abstract syntax elements themselves, but rather between the classes that are not displayed. To describe this type of mapping, the mapping language must provide a way to gather all the relations from the classes of a package and filter them to keep only those that end in other packages.

Implementation. AQL is the language that Obeo provides in Sirius Desktop and Sirius Web for navigating the model. In particular, it is used to describe the mapping from the abstract syntax elements to the concrete syntax elements in the View DSL. The mapping can be conditional to be applied only when an abstract syntax element's property has a specific value, for instance, enabling more dynamic concrete syntax. Finally, it is also used to describe the behavior of the different tools. AQL is inspired by OCL (OMG 2014) and has a very similar syntax, with the following specific difference: AQL has a static typing system like OCL, but the type system checking is separated from the execution phase. This enables to type check on demand: enabled for validation at design time, for the studio maker; and disabled in the studio, to avoid performance overhead and thus execute AQL queries faster. AQL supports union types to avoid falling back on a top type such as *Object* when a variable can have different types. This allows the list of common attributes of the possible types to be collected and provided to the studio maker via auto-completion.

Moreover, `null` or unsetted values are ignored at run time rather than failing the query, to be more resilient when creating models. Finally, AQL can be extended with Java code through Java services. This code consists of several Java functions developed in the Sirius Web backend that can be used directly as AQL functions.

Lessons learned. With Sirius Desktop, Obeo tried several kinds of mapping languages: concise and dynamically typed (Acceleo2[a]), that were complex to maintain; and general-purpose language (Java) and OCL, that were too verbose for a studio maker and their static strong typing system not flexible enough, falling back to a top type as *Object* when a variable can have different types. The main key factors Obeo needed for the mapping language are: simple and close to an existing standard to be used by non-expert studio makers; a flexible type system to provide useful completion even if an expression can have different types; resilient to constraint violation at run time since the model is incomplete; and extendable to complete the language when needed.

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

Open questions. In a low-code approach and a cloud environment, it is an open question to provide studio makers the ability to design complex mappings, while ensuring a correct balance between expressiveness, ease of use, data safety, and validity.

3.2. Language Development Lifecycle

This subsection details the rationales related to the development lifecycle of a language and the associated studio.

Rationale 5: Uncoupling abstract syntax and concrete syntax development lifecycles.

The abstract and concrete syntaxes should be developed as independently as possible, even at different times when possible.

Why. Uncoupling abstract syntax and concrete syntax developments helps studio makers in various scenarios. It permits the

development of multiple concrete syntaxes based on a single abstract one. It also follows the separation of concerns that permits: one studio maker expert on the domain of the DSL to work on the abstract syntax, while other studio maker experts on graphical representation can work on concrete syntaxes; the reuse of existing abstract syntaxes; the development of the concrete syntax first. Moreover, the abstract or concrete syntaxes may be standardized (*e.g.,* UML and its diagrams) and may be supplied by third parties. This rationale satisfies Requirements 2 and 4.

Illustration. UML Designer[11] is the UML studio developed by Obeo with Sirius Desktop. The first stable version was released in 2012. The language is implemented by two different teams: the first one developed the abstract syntax for Eclipse UML, independently of UML Designer; the second one developed the concrete syntax for UML Designer specifically. Since 2012, the studio followed multiple updates of UML that had impacts either on the abstract syntax or on the concrete syntax. These two artifact implementations were updated separately and have their own lifecycle.

Implementation. In Sirius Web, the development of 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 studio maker can create a new studio including a new representation of an existing abstract syntax of another studio. However, this requires the co-evolution of the abstract syntax and the concrete syntax: each time the studio maker modifies the abstract syntax, they might have to modify the concrete syntax and vice versa.

Lessons learned. Uncoupling the abstract syntax and concrete syntax development lifecycles enables more flexibility to maintain a studio over time. One or several studio makers can change the concrete syntax without affecting the abstract syntax, or co-evolve them according to end-user feedback, standard updates, and their own expertise. Similarly, an end user can more easily take the role of the studio maker and adjust the concrete syntax without requiring knowledge of metamodeling.

Open questions Uncoupling allows co-evolution between the abstract syntax and the concrete syntax. However, this raises the question of validating the concrete syntax and the mapping according to the evolution of the abstract syntax.

Rationale 6: Enabling seamless studio deployment.
The studios should be deployable seamlessly.

Why. Several language workbenches, such as Xtext, Langium, or Sirius Web, have the design-time distinct from the run-time environment. This separation can make cumbersome the testing and debugging phase of the language because the studio maker needs to rerun the run-time environment each time. In the same way, the studio maker needs to deploy the studio to enable the end user to use it, making more complex the iterations between the studio maker and the end users. Deploying seamlessly the studio allows easily developing a studio in a more agile way by switching between language development and a test model, and obtaining quick feedback. This rationale satisfies Requirements 3 and 5.

Illustration. When developing a studio, a studio maker may want quick feedback from end users, for testing or a rapid prototyping purpose. The seamless deployment of studios eases such use cases, as this shortens the distance between the studio maker's activities and its testing (by the studio maker) or use (by the end users).

Implementation. Sirius Web proposes different ways to deploy a studio. The main one is automatic, *i.e.,* when a studio maker creates or modifies a studio, Sirius Web automatically and seamlessly deploys this studio on the current Sirius Web instance. Sirius Web also proposes loading the studio directly from the backend, allowing built-in studios that cannot be modified from the low-code approach.

Lessons learned. By enabling a seamless deployment process, a studio maker can test the studio under development, and provide on a regular basis end users with new versions to test or use.

Open questions Currently, a studio maker cannot version their studio and choose which version(s) will be deployed. In the same way, existing models created with a previous version of a studio cannot be automatically migrated. In other words, this raises the question of co-evolving the language and the studio with the existing models.

3.3. Software Architecture

This subsection details the rationales related to the architecture of a language workbench and studio.

Rationale 7: Enabling model edition by interacting with the concrete syntax elements.

The edition of the model through the concrete syntax should be as intuitive as possible. The representation should not only be a projection of the model.

Why. Over time, software language engineering explored different approaches for managing the end user's interactions with the model through concrete and abstract syntaxes. Figure 3 illustrates these approaches. Originally, the textual languages have been implemented using a parser-based approach (*cf.* ❶ in Figure 3), implemented for instance by Xtext[12] and Langium[13]. In this approach, the instance of the abstract syntax (*i.e.,* the Abstract Syntax Tree) is directly inferred from the manipulation of the model through the concrete syntax. Each abstract syntax element must be represented in the concrete syntax or computable from it. Later, and more specifically to manage graphical notations or to easily combine different notations within a single concrete syntax, the projectional approach (*cf.* ❸ in Figure 3) has been proposed. This is implemented for example by MPS[14] and Freon[15], where the end user directly modifies the AST with-

[11] https://www.umldesigner.org/

[12] https://www.eclipse.org/Xtext/
[13] https://langium.org
[14] https://www.jetbrains.com/mps/
[15] https://www.freon4dsl.dev

out interacting with the concrete syntax, which is used only to project the AST to the end user. Since there is no parsing step, the form of the concrete syntax is freer: text, table, complex notations, diagram, *etc.* However, it is more complex to identify which abstract syntax element to edit from an end-user interaction, so more complex to develop for language workbench developers and eventually studio makers.

From this experience, Obeo explored an alternative as an intermediate approach between the two aforementioned (*cf.* ❷ in Figure 3), called *tool-based approach*. In this approach, we keep the reference of the represented abstract syntax element in each concrete syntax element. The edition of the abstract syntax is done using tools that can be triggered by interacting with a concrete syntax element. The tools are executed to edit the abstract syntax, which may trigger a recomputation of the concrete syntax. The updated representation is then sent to all clients who have it open. Finally, the representation can have its own state, independent of the model (typically visual attributes whose customization by end users does not affect the underlying model). For this reason, the server persists the representation. This solution does not offer as many possibilities as the projectional approach because we need to keep the reference of the abstract syntax element. For instance, representing multiple abstract syntax elements with a single concrete syntax element is challenging because the tools used must identify the specific abstract syntax element to which the action applies. However, it is simpler to put in place for the language workbench developers. Additionally, in the projectional approach, the concrete syntax is only a projection of the abstract syntax, resulting in a fixed layout with no own state. As the projectional approach, this tool-based approach facilitates the creation of heterogeneous representation types, enabling Requirement 1.

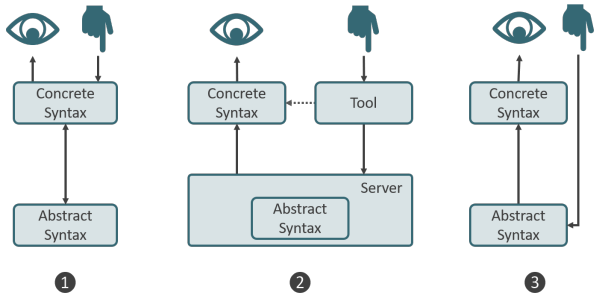

Figure 3 ❶ The parser-based editing approach; ❷ The tool-based editing approach; ❸ The projectional editing approach

Illustrations. As the representation can have its own state, the tool-based approach allows users to freely move nodes and edges of a diagram to adjust the layout when auto-layout is insufficient.

Implementation. Sirius Web follows the tool-based approach. Sirius Web proposes different kinds of tools, each being linked to a specific interaction or a part of the user interface: the contextual palettes for nodes and edges include a hard-coded delete icon button that the studio maker can describe using the delete tool; the studio maker can describe new node tools that will add each an icon button in the contextual palette of the specified concrete syntax node; *etc.* The behavior of a tool is open, and it is up to the language designer to define it. For instance, the studio maker may reconfigure the delete tool to not simply delete an element but also delete other model elements to keep the consistency of the model.

Lessons learned. The tool-based approach does not offer as many possibilities as a projectional approach, but in practice, it seems that this represents a good balance in terms of meeting customer needs.

Open questions. The tool-based approach requires the language designer to define tools for each interaction they want in the representation. This can be a long work to have a full-editable representation. The language workbench should facilitate this process without imposing too many intentions on the studio maker. One suggestion is to offer default tool implementations for new concrete syntax element descriptions that can be modified by the studio maker. However, this is only feasible for tools whose behavior can be predicted with a high degree of certainty, such as the delete tool. Another question is identical to the open question of Rationale 4: how to enable more complex behavior while ensuring a correct balance between expressiveness, ease of use, data, safety, and validity?

Rationale 8: Relying on a cloud-native architecture.

The language workbench and the developed studios should be accessible without any installation on the devices of the different users.

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; and 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 studio makers and end users from fully customizing their studios.

A cloud-native architecture with a web frontend is a good way to solve the first concern as there is nothing to install nor to update on the different computers of an organization, only the application installed in a server needs to be updated. This enables addressing Requirement 5 at the level of the architecture itself, complementary to Rationale 6 where the focus is more continuous deployment and delivery of the developed studios.

Moreover, web development is a rich ecosystem providing many facilities to develop the user interface. They can be used to create the graphic interface of the representation, such as CSS to lay out the content of a node for a diagram representation.

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

Implementation. Sirius Web relies on a cloud-native architecture with a web client, presented in Figure 4.

The client displays the diagram representation received from GraphQL via websocket using the client-side `Diagram Renderer` and listens for user-triggered tools to

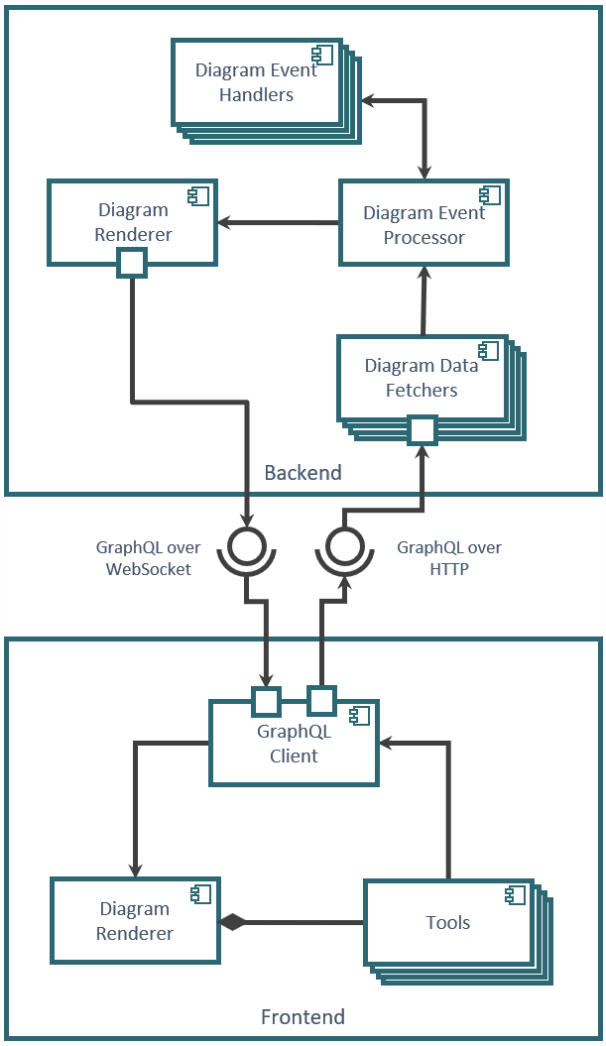

Figure 4 Overview of Sirius Web's architecture for the diagrams

send queries to the backend via GraphQL over websocket. All operations about models and representations are performed on the server side. The `Diagram Data Fetchers` receives the queries sent by the client and parses the payload before giving it to the `Diagram Event Processor`. According to the type of payload, the `Diagram Event Processor` dispatches it to the right `Diagram Event Handler` which interprets it and modifies the models or the representations accordingly. According to the type of modification, the `Diagram Event Processor` triggers the computation of the model's representations through the server-side `Diagram Renderer`. The `Diagram Renderer` applies the diagram representation description to compute the different nodes and edges, but also the layout of the representation. Finally, the new computed representations are broadcast to all the clients having them opened.

The reasons to place all the computations on the server are multiple. First, EMF is not available as a web library, preventing its use on the front-end side. Obeo wanted to capitalize on its 15 years experience of using this time-honored framework and its ecosystem. Next, Obeo targets models with around one million 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.

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 (*e.g.,* Eclipse RCP independent).

Open questions. Adopting a cloud-native architecture raises the question of what to compute on the server and client sides. 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 zooming (Blouin et al. 2015) for instance. Although they concern only the client that uses these features, the current architecture requires referring 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. To manage these kinds of features, providing a way to customize the location of operations or even to move these operations dynamically could be avenues to explore.

Rationale 9: Integrating the studios with other tools and vice versa.

The developed studios should be easily integrable inside other tools of the end-user workflow. In the same way, other tools should be integrable inside the studios.

Why. Many organizations already have their own workflows, of which modeling is only one part. To improve the user experience, it is better to be able to integrate a tool into another environment, in order to make its use more transparent. The Eclipse and EMF ecosystem eases the integration of tools of this ecosystem, like Xtext (Efftinge & Völter 2006) and Sirius Desktop but is mostly bound to desktop applications. In the other way, to be able to integrate other tools into its own studio avoids reinventing the wheel and is a way to propose something the users are already used to, for instance use a Google Map as a representation to visualize a position. This rationale satisfies Requirement 6.

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 sidebar but also to display the representation in a web view[16]. Another example is in Jupyter Lab using the Sirius Web

[16] https://www.youtube.com/watch?v=iqYsCy26eZg

API to get a representation in SVG format, and to display the information from a model in a Jupyter Lab cell[17].

Other applications in Sirius Web: Obeo created new Sirius Web representations using external applications, for instance with Google Map[18] to display a map and change position and zoom level directly from abstract syntax elements. Another example is the integration of Langium to add a textual representation in Sirius Web[19].

Implementation. 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 web view 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 a representation type for instance.

Lessons learned. The development lifecycle typically involves various tools, including modeling ones. To ensure their seamless integration, it is important to be able to integrate a studio in complex toolchains 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 in determining what model chunks to share.

3.4. The Sirius Web project development since 2020

This subsection aims to give more details about the development of Sirius Web itself. First, we will present some other more technical challenges Obeo met during the first years of the Sirius Web development. Finally, we will discuss how Obeo evaluates Sirius Web to real-world settings.

Sections 3.1 and 3.2 explained the main points of the Sirius project philosophy: heterogeneous configurable representations with arbitrary complex mappings between the abstract and concrete syntaxes. The main challenges of the first years of development concerned transposing this philosophy to the web ecosystem. The first challenge was to keep an extensible mechanism to be able, for example, to easily contribute new representation types. One of the chosen solutions was to rely the Sirius Web architecture on the Spring dependency injection mechanism. Thanks to that, the code is more uncoupled and Obeo can more easily add new features without impacting the core code, and reuse existing services without having to pull the dependencies of these services. Another benefit of this uncoupling is that Sirius Web is more modular. This means that an application using it as a framework can only consume parts of Sirius Web, such as diagrams, without having to pull Gantt, for instance. The second challenge was to leverage meta-modeling capabilities into the web. Indeed, EMF has no real equivalent in the web ecosystem. Moreover, studio generation, as was the case with Sirius Desktop, is not necessarily suited to the web, where we expect something more dynamic and seamless. The proposed solutions was to put all the computations in the backend (Rationale 8) and to interpret the studios (Rationale 6).

As Sirius Web notably targets industrial DSMLs, the language workbench needs to assess some real-word settings. Sirius Web is a generic tool, so most of the settings concern its capability to support large industrial models and generic features common to most DSMLs. Concerning the scalability, Sirius Web aims to support models with millions of elements, displayed through graphical representations with several thousand nodes. These different values come from the Obeo experiences with its different customers. Concerning the support of the most common features, part of them comes from Sirius Desktop and user requests for it. The others come from customer requests or from the development of specific standalone studios, such as SysON, where features that may be of interest to other studios are upstream to Sirius Web itself.

3.5. Discussion

Section 2.3 presents seven main requirements based on the experience Obeo acquired to fulfill for developing graphical language workbenches. In Section 3, we describe nine rationales that Obeo applied to develop Sirius Web for addressing these requirements.

The rationales identified several open questions that we can group into three categories: several usability concerns of the language workbench and the produced studios (Rationales 1 to 4 and 7); co-evolution between the studio and the language, as well as between the language and the models (Rationales 5 and 6); operation in an open ecosystem where there is a distinction between the clients and the server (Rationales 8 and 9).

The usability concerns Obeo worked on mainly match the the ergonomics criterion "Adaptability" from the Scapin and Bastien classification (Scapin & Bastien 1997), *i.e.,* how to propose a user interface that matches the expertise of the different users. The other criteria of this classification may also concern the development of language workbench, such the "Guidance" criterion that proposes to guide users in their tasks.

In the first category, we need low-code reuse at the level of representation, involving the concrete syntax but also the way to interact with it (tools in Sirius Web) (Rationale 2). Since the studio maker cannot identify all the needs of the end users, it is important to allow the end users to customize an existing representation to their needs and reuse the modifications they make in other representations of the same type. This will enable greater flexibility. Rationale 1 raises the need to adapt the language workbench according to domain expertise (*e.g.,* development of the concrete syntax, abstract syntax) which is a usability concern too. For instance, this can be done by adapting the graphic user interface according to the activity, in the same way Eclipse IDE provides with the perspectives. This can also be applied to

[17] https://www.youtube.com/watch?v=q9DeMS3G4TA
[18] https://www.youtube.com/watch?v=mcJUuW1c4wU
[19] https://www.youtube.com/watch?v=t-BISMWMtwc

the studio itself. Currently, the development of the studio by the studio maker is restricted at the level of the language itself (abstract and concrete syntax) and the way to interact with (tools). However, the studio is larger than the language it provides, and the language workbench should provide a way to customize other parts as the graphic user interface or the different default language-specific features (*e.g.,* undo-redo, history, semantic zoom, *etc.*). Finally, the presentation of the model itself must be tailored to the level of expertise or permissions of the end user.

The second category is about co-evolution, this is inherent when different artifacts that need to work together are decoupled. Rationale 5 raises the need to tool the co-evolution between the different artifacts of the studio, particularly when modifying the abstract syntax. The language workbench should identify necessary updates and assist in modifying the concrete syntax, mapping, and associated tools when changes are made to the abstract syntax. There is also the co-evolution between the studio and the different models (Rationale 5). If a concept of the abstract syntax is modified or even removed and the new version of the studio is deployed, the studio may automatically update the model or ask the end user the action to realize when the modification leads to data loss.

Finally, the third category is about evolving in an open ecosystem like the Web. Sirius Desktop is integrated inside the Eclipse IDE environment, as many other tools (Xtext, EMF, GMF, *etc.*). These different tools can interoperate more easily since they can share memory, for instance. They are also monolithic tools, without distinction between the backend and the frontend. Placing a language workbench on the Web brings new challenges. With a backend on a remote device, Rationale 8 raises the need to control where operations are executed between the backend and the frontend in order to optimize the latency or the performance. Since the Web is a more open ecosystem than Eclipse IDE, there is a bigger heterogeneity in terms of technology used to develop the different applications. Each application has its own memory space and their backend does not necessarily run on the same device. With Rationale 9, we illustrate the need to integrate the studio with other tools. However, this needs to define a standard protocol between the different tools in order to be able to exchange data such as the models themselves. LionWeb[20] (Language Interfaces on the Web) initiative is trying to resolve the interoperability at the level of the language workbench itself, by proposing a protocol that enables the share of language-oriented modeling features. It defines itself as "LSP for models"(*The LionWeb initiative* n.d.).

4. Use Cases

This section presents two use cases that illustrate the different implementation choices detailed in Section 3: the Tax studio, which is representative of the low-code development interface capabilities; and SysON, a studio for the industrial language SysML v2. These use cases show that Sirius Web can support simple studios such as Tax studio and more complex studios like SysON.

[20] https://lionweb.io/

All the needed information to install Sirius Web and to replicate the different use cases are available in Appendix A. The executable[21] and the bug tracker[22] are on the Sirius Web GitHub repository.

4.1. Tax studio

This example is developed using Sirius Web 2024.1.2. The tax studio example was developed for LangDev 2023[23] with the low code interface only (Rationale 1). This DSML took several tens of minutes to develop by a language designer who knows Sirius Web well.

This studio allows the calculation of the price of the customer's cart with the applied VAT (*Value-Added Tax*) according to the jurisdiction of the customer. The studio offers two activities for the end user: modeling of the different jurisdictions and VATs within them, as well as the customers residing in those and the products subject to the VATs; defining the content of a customer's cart, with computation of the price based on the customer's jurisdiction. Figure 5 shows a model created using the Tax studio which focuses on the former activity. This model has two representations, one for each activity (Rationale 2): the former one (pointed by ❶ in Figure 5) is the visible diagram representation; the latter (pointed by ❷) is a form representation for the second activity. ❸ shows the contextual palette where most of the tools of a selected element are, here the tools of the model root "World". This is thanks to these tools that an end user can interact with the model (Rationale 7). Figure 6 shows the representation of the second activity, representing the cart of Alice with the different products inside (here 1kg of apples).

For producing such an output studio, a studio maker had to develop the abstract syntax and the concrete syntax of the DSL using the Sirius Web language workbench. Figure 7 is a screenshot of Sirius Web during the development of the tax DSL abstract syntax, so from the studio maker's viewpoint. The explorer view shows the two models to describe the Tax studio: the *Domain* model (❶) and the *View* model (❷).

Now, the studio maker can develop the concrete syntax for representing the abstract syntax using the *View* DSL. Figure 8 is a screenshot of a diagram concrete syntax under development for the tax DSL. The selected element in the explorer view (❶) is the description of a diagram node to represent the `Jurisdiction` concept of the abstract syntax. On the details view the studio maker can map abstract syntax elements to the selected concrete syntax element.

Here, in the *Domain type* (❷), the studio maker specifies the abstract syntax concept represented by the described concrete syntax element. In the *Semantic Candidates Expression* (❸), the studio maker specifies an AQL expression returning all the abstract syntax elements they want to represent by the described concrete syntax element, filtered according to *Domain type*. This is how Sirius Web bridges the gap between the concrete and the abstract syntax (Rationale 4). This AQL expression starts from an instance of a concept de-

[21] https://github.com/eclipse-sirius/sirius-web/packages/1660490
[22] https://github.com/eclipse-sirius/sirius-web/issues
[23] http://langdevcon.org

fined as the root of the representation in the diagram description. Here the root is an instance of the concept `World` represented in the context of the *Semantic Candidates Expression* by the variable `self`. So, the `self.jurisdictions` simply follows the `jurisdictions` relation (❹) from `World` to `Jurisdiction` to get all the `Jurisdiction` instances. Finally, the studio maker opts to represent this node using an image, as we can see in the `Jurisdiction Node` subtree (❺).

Currently, Sirius Web does not provide a dedicated representation for developing diagram concrete syntaxes. For that, a studio maker has to use explorer view (❶ on Figure 7).

The studio is automatically deployed to the current instance (Rationale 6) when all the studio makers leave the project (*i.e.,* by returning to the Sirius Web home page). This choice is made to ensure that a studio is not deployed during its development. Following that, the studio maker can create a model based on the Tax DSL inside a new project to use the Tax studio and iterate between the studio project and their test project to improve the studio (*cf.* Figure 5).

4.2. SysON

SysON[24] is an open-source studio for SysML v2, hosted by the Eclipse Foundation and developed by Obeo in collaboration with the CEA-List (French Alternative Energies and Atomic Energy Commission — Laboratory for Integration of Systems and Technology). SysML v2 is the successor of the well-established SysML standard for complex systems engineering introduced by the OMG (Object Management Group). SysON demonstrates Sirius Web's ability to manage the development of complex languages and their studios for heterogeneous modeling (Requirement 1).

SysON is a standalone application that uses Sirius Web as a framework (*i.e.,* Maven dependencies) and reuses several of its components (*e.g.,* diagram representation). As this studio is much bigger than the previous one, using the programmatic APIs presented in Rationale 1 enables the definition of the SysML v2 metamodel with Ecore which is more expressive than the Domain DSL. This also enables the benefit of the IDE (*i.e.,* refactor, *etc.*), of the versioning through Git, and more customization of the GUI such as proposing more SysML v2-specific features in the explorer view.

Listing 1 gives an example of the diagram representation description of the SySML v2 "Definition" abstract concept, defined with the programmatic API. The definition of the representation descriptions is made with the builder design pattern. The `semanticCandidateExpression` (line 11) corresponds to the mapping, here the definition of this expression is delegated to the concrete description of the "Definition" concept that overrides the function 'getSemanticCandidatesExpression', for instance the representation description of "PartDefinition".

```
1  @Override
2  public NodeDescription create() {
3      String domainTypeName =
       SysMLMetamodelHelper.buildQualifiedName(
       this.eClass);
4      return this.diagramBuilderHelper.
       newNodeDescription()
5          .collapsible(true)
6          .defaultHeightExpression(ViewConstants
       .DEFAULT_CONTAINER_NODE_HEIGHT)
7          .defaultWidthExpression(ViewConstants.
       DEFAULT_NODE_WIDTH)
8          .domainType(domainTypeName)
9          .insideLabel(this.
       createInsideLabelDescription())
10         .name(this.getDescriptionNameGenerator
       ().getNodeName(this.eClass))
11         .semanticCandidatesExpression(this.
       getSemanticCandidatesExpression(
       domainTypeName))
12         .style(this.createDefinitionNodeStyle
       ())
13         .build();
14 }
```

Listing 1 Fragment of the representation description for the "Definition" SysML v2 concept using Sirius Web programmatic API

SysML v2 is still in development, so SysON must adapt to future versions. According to the artifacts impacted by the versions, the developers of SysON may only update the concerned artifact, as was the case for UML Designer, explained in Rationale 5.

Figure 9 is a screenshot of SysON illustrating the development of a model to define a vehicle and its steering subsystem.

5. Related Works

This section first discusses related language workbenches. It then reports related work on good practices for modeling.

5.1. Language workbenches

Erdweg et al. (Erdweg et al. 2015) proposed a feature model to identify and classify the main features of the numerous language workbenches that exist. In this subsection, we compare Sirius Web to some existing non-textual language workbenches according to two features of the Erdweg et al. (Erdweg et al. 2015) classification we discussed in Section 3 and which we consider to be Sirius Web's most important: the representation types (notation in Erdweg et al. (Erdweg et al. 2015)) and the editing approach. We also consider: if the language workbench is open source; the coupling between the abstract and the concrete syntax; the ability to be used and to deploy studios on the Web. The language workbenches we consider are: MetaCase MetaEdit+[25] (Kelly et al. 1996); CINCO Cloud[26] (Bainczyk et al. 2022); JetBrains MPS[27] (Pech et al. 2013); Gentleman[28] (Ducoin & Syriani 2022); JJodel (Di Rocco et al. 2023); and WebGME[29] (Maroti et al. 2014). MPS is the most common non-textual language workbench in scientific literature. MetaEdit+, WebGME, and

[24] https://github.com/eclipse-syson/syson

[25] https://www.metacase.com/mwb/
[26] https://scce.gitlab.io/cinco-cloud/
[27] https://www.jetbrains.com/mps/
[28] https://github.com/geodes-sms/gentleman
[29] https://webgme.org/

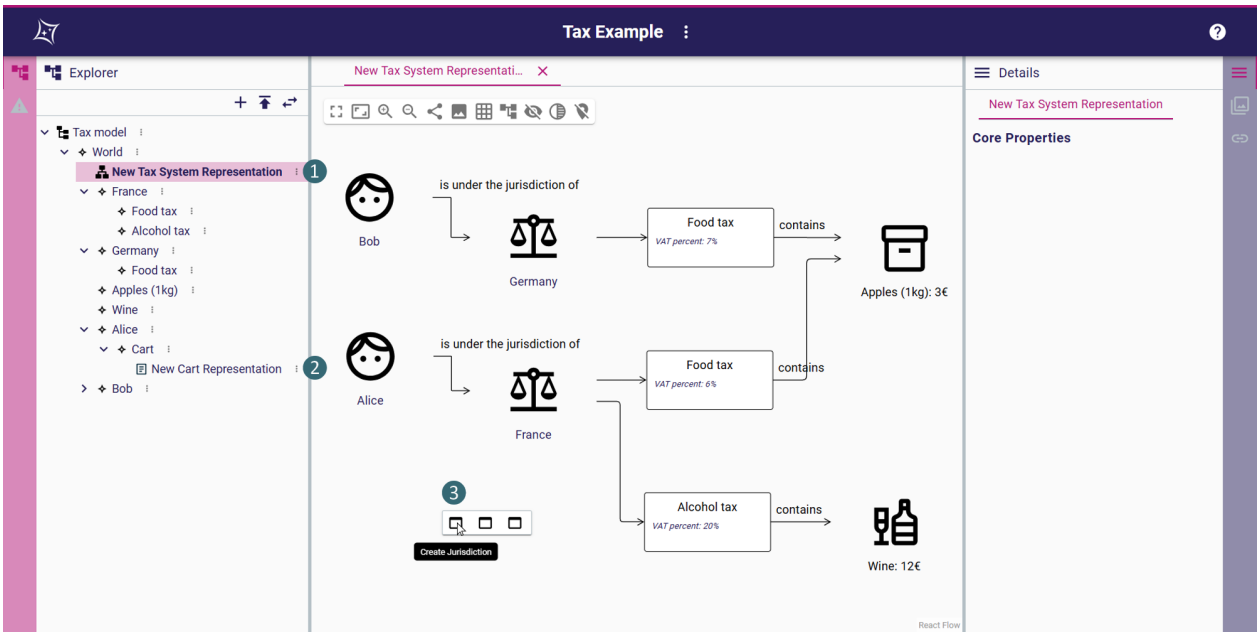

Figure 5 A tax model made with the tax studio, viewed according to the first activity.

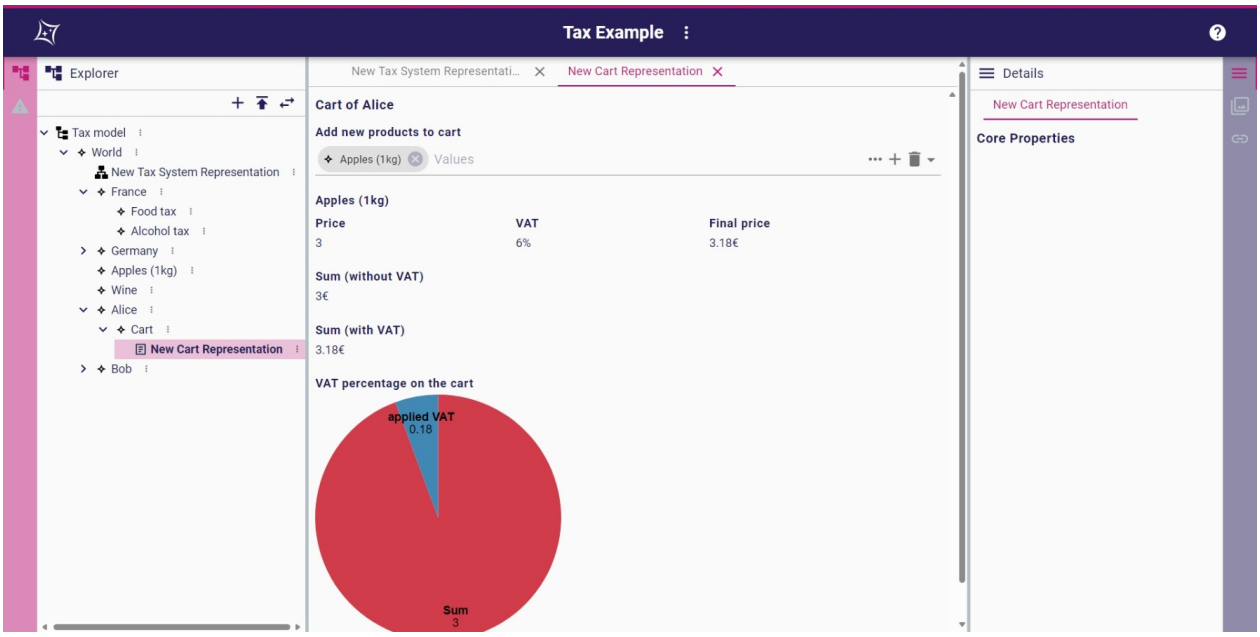

Figure 6 A tax model made with the Tax studio, viewed according to the second activity.

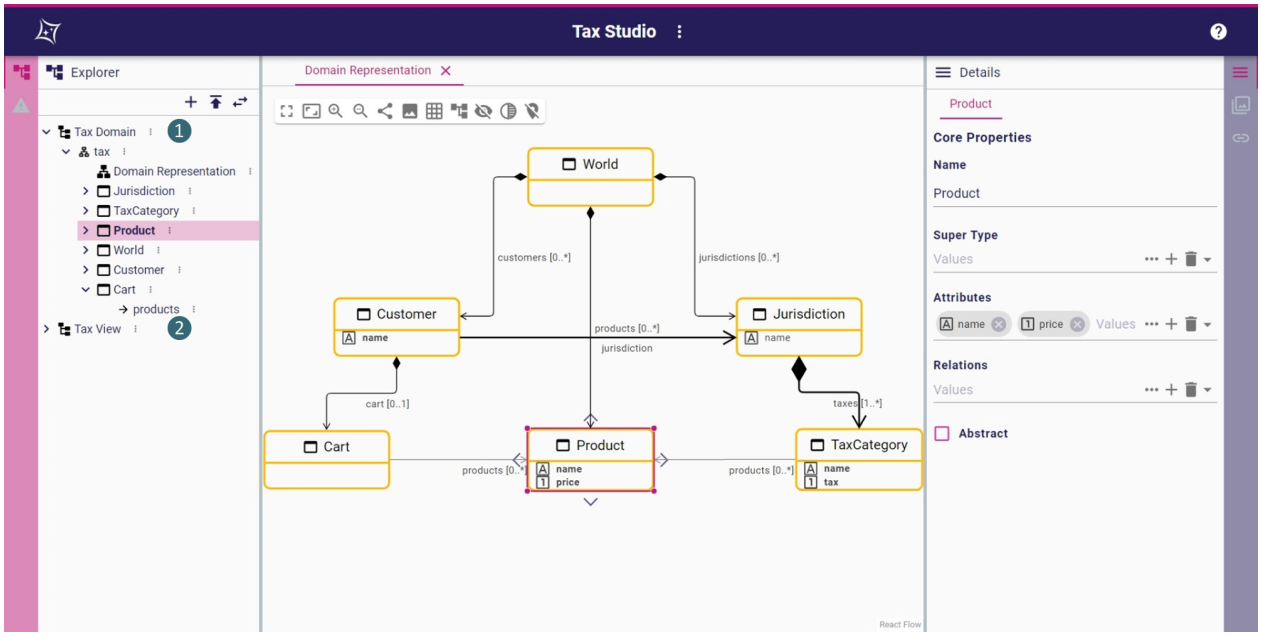

Figure 7 The description of the Tax DSL abstract syntax in Sirius Web.

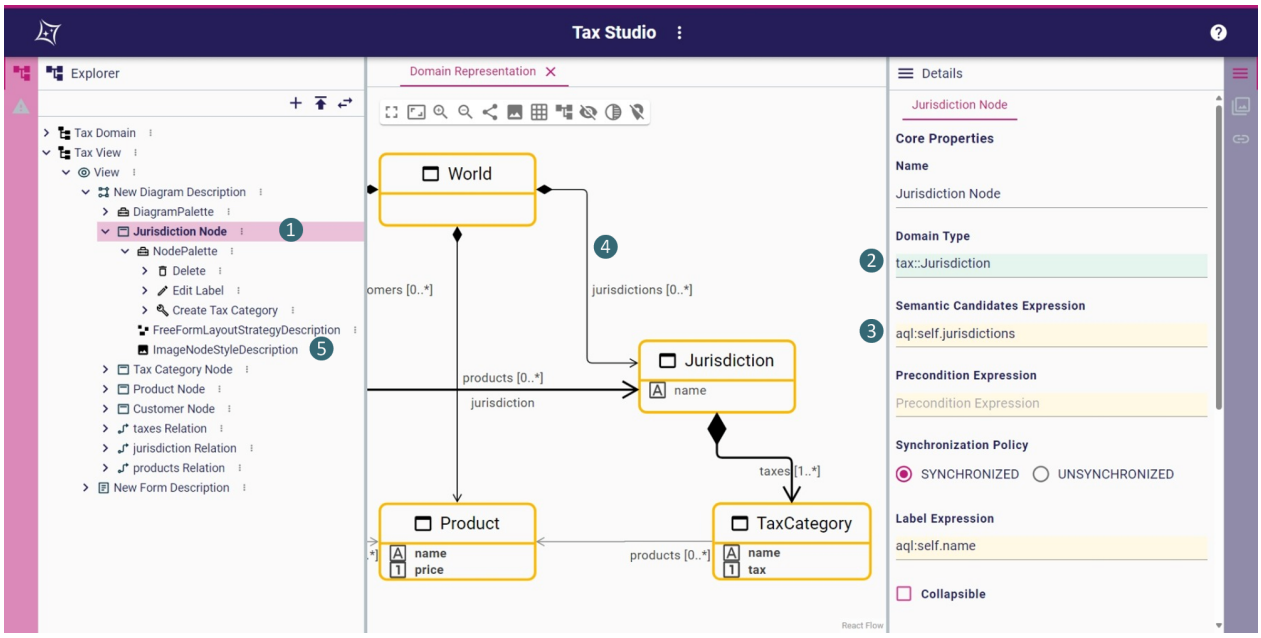

Figure 8 The description of the Tax DSL diagram concrete syntax in Sirius Web.

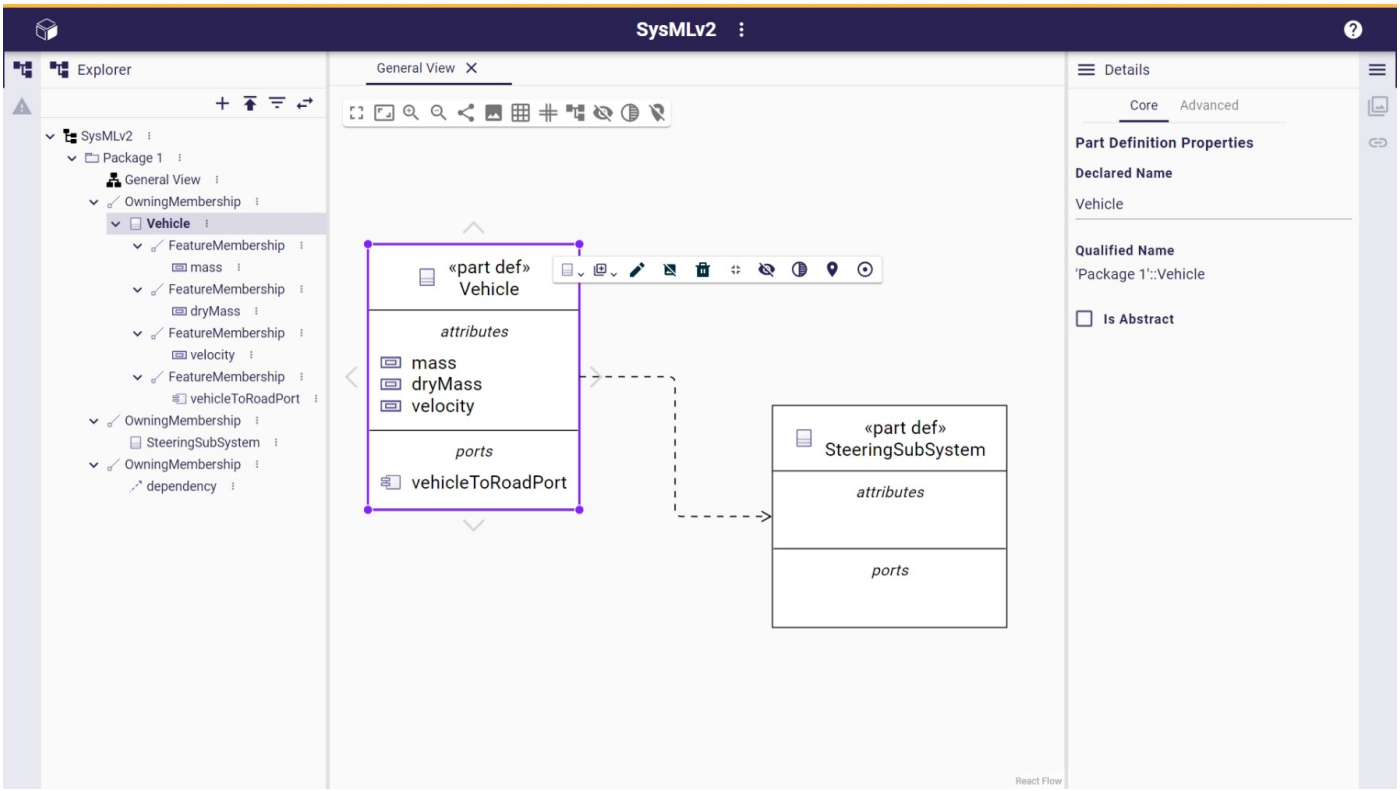

Figure 9 A vehicle example using SysML v2 in *SysON* (*SysON* 2024).

CINCO Cloud are the closest language workbenches to Sirius Web we found in the literature, proposing the development of diagram concrete syntax. Table 2 highlights the different features of the compared language workbenches. The discussion bellow compares them to those of Sirius Web.

The first comparison criterion is whether the language workbench is open source. Sirius Web is open source, under the EPL-2.0 (Eclipse Public License) like CINCO Cloud. WebGME is under MIT (Massachusetts Institute of Technology) license and MPS is under Apache License 2.0. Gentleman is under GPL-3.0 license and JJodel does not specify any license. Both are open source too. Finally, MetaEdit+ is closed source.

To answer to Requirement 1, Sirius Web provides several representation types (graphic, form, Gantt, and kanban) that can be used to support the different activities of the end users Rationale 2. These representation types align with the notations described in Erdweg et al. (Erdweg et al. 2015) as the means by which programs or models are presented to users. Due to its projectional nature, MPS allows the design of numerous representation types: text, table, matrix, mathematical notation, *etc.*. However, MPS no longer natively supports diagram representation[30], and a language user should use a community MPS plugin for that. Gentleman is also projectional and proposes two representation types: container (*i.e.,* HTML component tree) and graphical. MetaEdit+ supports graphics, tables, and matrices representation types (Kelly et al. 1996). On the other hand, CINCO Cloud, JJodel and WebGME support only graphical representation.

The different representation enables the visualization and the edition of the abstract syntax. Erdweg et al. (Erdweg et al. 2015) proposed two editing approaches: free-form editing where "the user freely edits the persisted model (typically as text)" (Erdweg et al. 2015); and the projectional editing where "the user edits a projection of the persisted model in a fixed layout" (Erdweg et al. 2015). MPS follows the projectional approach. As discussed in Rationale 7, the projectional approach allows more freedom in terms of representation, but this comes with the drawback of having purely static representation. In other words, the end user cannot interact with the concrete syntax except with the cells where they can input the model properties. While this is not a problem for a form representation, for a diagram representation the end user relies heavily on the ability of the automatic layout the studio proposed. They cannot move the diagram nodes or rearrange the edges if the automatic layout is not sufficient. The tool-based approach of Sirius Web is a third approach between the free-form and the projectional editing, as described in Rationale 7. The user does not interact with the serialized model itself, similar to projectional editing. However, the concrete syntax layout is not fixed, and the concrete syntax is also serialized to preserve user preferences, such as manual layout. Finally, the studio maker can describe tools for customizing the way to interact with the abstract syntax through the concrete syntax. Gentleman also follows the projectional approach and proposes graphical representations. To mitigate the layout impact on such an environment, Gentleman proposes different layout types. The

[30] https://www.jetbrains.com/help/mps/diagramming-editor.html

Table 2 Comparative of Sirius Web with other identified non-textual language workbenches. LW = Language workbench; math. notations = mathematical notations; AS = abstract syntax; CS = concrete syntax; DS = dynamic style

Feature \ LW	Sirius Web	MetaEdit+	CINCO Cloud	MPS	Gentleman	JJodel	WebGME
Open source	✓	✗	✓	✓	✓	✓	✓
Representation type	– graphic – form – Gantt – kanban	– graphic – table – matrix	graphic	– text – table – matrix – math. notations	– container – graphic	graphic	graphic
Editing approach	hybrid	hybrid	hybrid	projectional	projectional	hybrid	hybrid
Coupling AS-CS	$1 \rightarrow *$, DS	$1 \rightarrow [0..1]$, DS	$1 \rightarrow 1$, DS	*n.a*	$1 \rightarrow 1$	$1 \rightarrow [0..1]$, DS	$1 \rightarrow 1$
Web support	✓	✗	✓	✗	✓	✓	✓

different layouts are configurable by the language designer. For instance for the force layout Gentleman proposes, the language designer can customize the distance between two nodes. To improve usability, Gentleman proposes interaction points inside the graphical representation with which end users may interact to modify the abstract syntax, for instance to create a relation between two elements. According to Erdweg et al. (Erdweg et al. 2015), MetaEdit+ proposes a free-form editing approach. In practice, Erdweg et al. (Erdweg et al. 2015) does not provide a finer granularity for categorizing the editing approach. MetaEdit+ rather proposes a hybrid approach between free-form and projectional. In fact, MetaEdit+ persists both the concrete and the abstract syntax (Tolvanen & Kelly 2016) as in the tool-based approach. Finally, WebGME and CINCO Cloud, JJodel propose free interaction with the concrete syntax and persist the model. So they follow a hybrid approach too. However, unlike Sirius Web, WebGME does not offer customization for interaction.

When the concrete syntax is uncoupled from the abstract syntax, the language workbench should provide a way to bridge the gap between the former to the latter. This may be divided into two different aspects: the expressiveness of the mapping to bridge the gap, *i.e.,* if an abstract syntax element may be represented by zero, one or more concrete syntax elements (and vice-versa); and the capacity to adapt the style of a concrete syntax element according to the attributes of the represented abstract syntax element. For instance changing the border color if the class is an abstract one (dynamic style). As described in Rationale 4, Sirius Web proposes a mapping language that supports bridging arbitrarily gap between the abstract syntax and the concrete syntax. The mapping can also be conditional to enable a dynamic concrete syntax according to an abstract syntax element's property. Sirius Web also proposes dynamic styles named conditional styles to adapt the style of an concrete syntax element according to the attributes of the represented abstract syntax element. In MetaEdit+, the abstract syntax and concrete syntax are more strongly coupled: an abstract syntax concept has a unique concrete syntax that consists of several graphical elements (shapes, texts, images) that the studio maker can customize. Although MetaEdit+ offers a way to condition the display and the style of the different graphical elements based on the properties' values of the represented abstract syntax element, it does not provide a mapping language that allows for more freedom. For instance, an abstract syntax element cannot be represented multiple times in a diagram. Like MetaEdit+, WebGME does not propose a language dedicated to the mapping. Each abstract syntax element is represented by only one concrete syntax element that cannot be determined according to its properties. CINCO Cloud is also restricted to one-to-one mapping but enables dynamic styles. Gentleman seems to have the same limitation. To describe the concrete syntax of the abstract syntax, JJodel proposes to directly specify React code template through the JSX format. Thanks to that, a language designer can customize the style of a concrete syntax element according to the abstract syntax element (dynamic style). Moreover, it proposes OCL expressions to constraint the display of an element, enabling a partial model to be displayed. In other words, all the abstract syntax elements matching the OCL constraint will be displayed by the current concrete syntax element description. This approach is close to the one Sirius Web proposes, enabling more complex mappings. However, it does not seem possible to display an element several times.

Finally, a studio maker can use Sirius Web on the Web and deploy the studios in the same Sirius Web instance in which they were developed. CINCO Cloud, Gentleman, JJodel and WebGME have the same ability. MetaEdit+ and MPS cannot be used or deploy studio on the Web.

5.2. Good practices for (meta-)modeling

Our paper proposes rationales for the development of language workbenches that produce graphical studio. This is related to good practices for (meta-)modeling. Cho & Gray (Cho &

Gray 2011) discussed solutions for recurrent design issues while defining metamodels by proposing three design patterns for metamodel. Closely related, Ergin et al. (Ergin et al. 2016) proposes design patterns for model transformation.

With the aim to consider the end user's expertise when developing language, López-Fernández et al. (López-Fernández et al. 2015) proposed an approach to induct a language specification from example model fragments with a virtual assistant that provides suggestions to improve the metamodel.

Pescador et al. (Pescador et al. 2015) proposed patterns for building DSLs and their studios, with a focus on patterns for configuring services in such studios. Sirius Web does not currently provide template-based assistance for defining models. However, a studio maker can create such behavior by defining tools for their representation type descriptions. Izquierdo & Cabot (Izquierdo & Cabot 2016) discussed an approach for improving the design of DSML based on collaborative work between the studio makers and the end users.

If these research works focus on improving the design of DSLs, we focus in this paper on reporting lessons learned about the development of language workbenches.

6. Conclusion and perspective

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

Studios provide interactive features to help end users use the modeling language and the model produced (*e.g.,* help with navigation, understanding, editing). However, such interactive features are currently hand-coded for each studio. Future work on Sirius Web may investigate how to provide studio makers with the ability to integrate and adapt such interactive features within their studios for improving the usability.

References

Abrahão, S., Bourdeleau, F., Cheng, B., Kokaly, S., Paige, R., Stöerrle, H., & Whittle, J. (2017). User experience for model-driven engineering: Challenges and future directions. In *2017 acm/ieee 20th international conference on model driven engineering languages and systems (models)* (pp. 229–236).

Bainczyk, A., Busch, D., Krumrey, M., Mitwalli, D. S., Schürmann, J., Tagoukeng Dongmo, J., & Steffen, B. (2022). Cinco cloud: A holistic approach for web-based language-driven engineering. In T. Margaria & B. Steffen (Eds.), *Leveraging applications of formal methods, verification and validation. software engineering* (pp. 407–425). Cham: Springer Nature Switzerland.

Blouin, A., Moha, N., Baudry, B., Sahraoui, H., & Jézéquel, J.-M. (2015). Assessing the use of slicing-based visualizing techniques on the understanding of large metamodels. *Information and Software Technology*, *62*, 124-142. Retrieved from https://www.sciencedirect.com/science/article/pii/S0950584915000373 doi: https://doi.org/10.1016/j.infsof.2015.02.007

Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., & Combemale, B. (2016). Execution framework of the gemoc studio (tool demo). In *Proceedings of the 2016 acm sigplan international conference on software language engineering* (p. 84–89). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2997364.2997384 doi: 10.1145/2997364.2997384

Cho, H., & Gray, J. (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* (pp. 25–32).

Cicchetti, A., Ciccozzi, F., & Pierantonio, A. (2019, December). Multi-view approaches for software and system modelling: a systematic literature review. *Software and Systems Modeling*, *18*(6), 3207–3233.

Ciccozzi, F., Tichy, M., Vangheluwe, H., & Weyns, D. (2019). Blended modelling - what, why and how. In *2019 acm/ieee 22nd international conference on model driven engineering languages and systems companion (models-c)* (p. 425-430). doi: 10.1109/MODELS-C.2019.00068

Combemale, B., France, R., Jézéquel, J.-M., Rumpe, B., Steel, J., & Vojtisek, D. (2016). *Engineering modeling languages: Turning domain knowledge into tools*. CRC Press.

David, I., Aslam, K., Faridmoayer, S., Malavolta, I., Syriani, E., & Lago, P. (2021). Collaborative model-driven software engineering: a systematic update. In *2021 acm/ieee 24th international conference on model driven engineering languages and systems (models)* (pp. 273–284).

Di Rocco, J., Di Ruscio, D., Di Salle, A., Di Vincenzo, D., Pierantonio, A., & Tinella, G. (2023). jjodel – a reflective cloud-based modeling framework. In *2023 acm/ieee international conference on model driven engineering languages and systems companion (models-c)* (p. 55-59). doi: 10.1109/MODELS-C59198.2023.00019

Ducoin, A., & Syriani, E. (2022). Graphical projectional editing in gentleman. In *Proceedings of the 25th international conference on model driven engineering languages and systems: Companion proceedings* (p. 46–50). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3550356.3559092 doi: 10.1145/3550356.3559092

Efftinge, S., & Völter, M. (2006). oaw xtext: A framework for textual dsls. In *Workshop on modeling symposium at eclipse summit* (Vol. 32).

Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W. R., ... van der Woning, J. (2015, December). Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, *44*, 24–47. Retrieved 2022-12-14, from https://www.sciencedirect.com/science/article/pii/S1477842415000573 doi: 10.1016/j.cl.2015.08.007

Ergin, H., Syriani, E., & Gray, J. (2016, Novem-

ber). Design pattern oriented development of model transformations. *Computer Languages, Systems & Structures*, *46*, 106–139. Retrieved 2024-04-15, from https://www.sciencedirect.com/science/article/pii/S1477842416300148 doi: 10.1016/j.cl.2016.07.004

IEEE. (2011). *Iso/iec/ieee systems and software engineering – architecture description.* Retrieved from https://standards.ieee.org/ieee/42010/5334/

Izquierdo, J. L. C., & Cabot, J. (2016). Collaboro: a collaborative (meta) modeling tool. *PeerJ Computer Science*, *2*, e84.

Kelly, S., Lyytinen, K., & Rossi, M. (1996). Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In P. Constantopoulos, J. Mylopoulos, & Y. Vassiliou (Eds.), *Advanced information systems engineering* (pp. 1–21). Berlin, Heidelberg: Springer Berlin Heidelberg.

Kolovos, D. S., Paige, R. F., & Polack, F. A. (2006). Eclipse development tools for epsilon. In *Eclipse summit europe, eclipse modeling symposium* (Vol. 20062, p. 200).

Köhnlein, J., & Brun, C. (2017, 12). *Xtext / Sirius White Paper* (Tech. Rep.). Typefox and Obeo.

The lionweb initiative. (n.d.). Retrieved 2024-04-12, from https://github.com/LionWeb-io/.github/raw/main/profile/LIonWeb%20at%20MPS%20Meetup%202023.pdf

López-Fernández, J. J., Cuadrado, J. S., Guerra, E., & De Lara, J. (2015, October). Example-driven meta-model development. *Software & Systems Modeling*, *14*(4), 1323–1347. Retrieved 2024-04-15, from http://link.springer.com/10.1007/s10270-013-0392-y doi: 10.1007/s10270-013-0392-y

Maroti, M., Kecskes, T., Kereskenyi, R., Broll, B., Juracz, L., & Levendoszky, T. (2014). Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure.

Mernik, M., Heering, J., & Sloane, A. M. (2005, dec). When and how to develop domain-specific languages. *ACM Comput. Surv.*, *37*(4), 316–344. Retrieved from https://doi.org/10.1145/1118890.1118892 doi: 10.1145/1118890.1118892

OMG. (2014). *Object Constraint Language (OCL).* Version. Retrieved from http://www.omg.org/spec/OCL/

Ozkaya, M., & Akdur, D. (2021). What do practitioners expect from the meta-modeling tools? a survey. *Journal of Computer Languages*, *63*, 101030. Retrieved from https://www.sciencedirect.com/science/article/pii/S2590118421000095 doi: https://doi.org/10.1016/j.cola.2021.101030

Pech, V., Shatalin, A., & Voelter, M. (2013). Jetbrains mps as a tool for extending java. In *Proceedings of the 2013 international conference on principles and practices of programming on the java platform: Virtual machines, languages, and tools* (pp. 165–168).

Pescador, A., Garmendia, A., Guerra, E., Cuadrado, J. S., & de Lara, J. (2015). Pattern-based development of domain-specific modelling languages. In *2015 acm/ieee 18th international conference on model driven engineering languages and systems (models)* (pp. 166–175).

Scapin, D., & Bastien, C. (1997). Ergonomic criteria for evaluating the ergonomic quality of interactive systems. *Behaviour & information technology*, *16*(4-5), 220–231.

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *Emf: eclipse modeling framework*. Pearson Education.

SysON. (2024). Retrieved from https://mbse-syson.org/

Tolvanen, J.-P., & Kelly, S. (2016). Model-driven development challenges and solutions: Experiences with domain-specific modelling in industry. In *2016 4th international conference on model-driven engineering and software development (modelsward)* (p. 711-719).

Wachsmuth, G. H., Konat, G. D., & Visser, E. (2014). Language design with the spoofax language workbench. *IEEE Software*, *31*(5), 35-43. doi: 10.1109/MS.2014.100

Warmer, J., & Kleppe, A. (2003). *The object constraint language: Getting your models ready for mda* (2nd ed.). Addison-Wesley Professional.

About the authors

Théo Giraudet is a PhD student working at Obeo (France) and University of Rennes (France). Contact him at theo.giraudet@irisa.fr.

Mélanie Bats is Obeo's CTO. She has been leading the product teams for over six years. During this time, she has played a key role in shaping the strategy of the Eclipse Sirius project and also contributes to the development of various modeling projects like Sirius Desktop, Sirius Web, EEF and SysON. More information at http://melb.enix.org/. Contact her at melanie.bats@obeosoft.com.

Arnaud Blouin is associate professor at INSA Rennes (France). More information at https://people.irisa.fr/Arnaud.Blouin/. Contact him at arnaud.blouin@irisa.fr.

Benoit Combemale is Full Professor of Software Engineering at the University of Rennes, France. His research interests include Model-Driven Engineering (MDE), software language engineering (SLE), software validation and verification (VV) and DevOps. He holds a Ph.D. in Software Engineering from the University of Toulouse, France, and an Habilitation in Software Engineering from the University of Rennes. More information at http://combemale.fr/. Contact him at benoit.combemale@irisa.fr.

Pierre-Charles David holds a Ph.D. in Software Engineering from the University of Nantes, France. He is project lead on the Eclipse Sirius project. He works at Obeo as a software developer where he participated in the development of Sirius Desktop for more than 10 years before switching his focus on Sirius Web since its inception. More information at https://pcdavid.net/. Contact him at pierre-charles.david@obeosoft.com.

A. Appendix

This part explains how to install Sirius Web and the two cited studios: Tax and SysON (SysML v2). For these two examples, one needs to install Java 17[31] and a PostgreSQL database. In these instructions, Docker will be used to install PostgreSQL. The SysON studio is independent of Sirius Web and must be installed separately.

Sirius Web

1. Download the file *sirius-web-sample-application-2024.1.2.jar* from the download page[32] of the Sirius Web repository. In this article, the version used for Tax is 2024.1.2.

2. Start a PostgreSQL database using the command provided in Listing 2.

3. Start Sirius Web using the command in Listing 3. SysON is started when the line "Started SampleApplication in X seconds" is printed in the console (directly followed by the images loaded).

4. In a recent browser (Firefox, Chrome), go to `http://localhost:8080`.

More information can be found in the README of the Sirius repository Web[33].

A.1. Tax

1. Download the *tax_studio.zip*, *tax_example.zip* files and the four SVG images from the Tax Studio repository [34].

2. Import them into Sirius Web using the "Upload project" tile shown in Figure 10.

3. Currently, Sirius Web does not export the images used in the concrete syntax when exporting a studio. Figure 11 illustrates how to import them manually:

   (a) Open the Tax Studio project and go to the settings (❶) to upload the different images. Give them the name of the image as a label. To return to the project, you can use your browser's "go back" feature or click the Sirius Web icon to go to the home page.

   (b) Click on the first image style in the Tax View model (❷) and change the shape to select *jurisdiction* (❸).

   (c) Repeat the same step for the other image styles (❹-❻). The image style ❺ must have as shape the alcohol image. It is a conditional style that is applied only when a tax category of a product contains the word "alcohol".

4. Now, you can switch between the two projects to see how the studio is defined (Tax Studio) and an example of use (Tax Example).

5. You can create a new representation by following the steps as shown in Figure 12.

A.2. SysON

SysON is distributed apart from Sirius Web and is installed independently. The Sirius Web instance must be stopped if it is running to release port 8080.

1. Download the file *syson-application-2024.3.0.jar* from the download page[35] of SysON. In this article, the version used is 2024.3.0.

2. Start a PostgreSQL database using the command provided in Listing 4.

3. Start SyON using the command provided in Listing 5. SysON is started when the line "Started SysONApplication in X seconds" is printed in the console.

4. Go to `http://localhost:8080`.

5. Create a new *Batmobile* model directly from the homepage.

6. You can open the project and the representation named "General View". By default, the representation is "unsynchronized". You must drag and drop the elements from the explorer view to the opened representation to display them inside.

[31] https://adoptium.net/temurin/releases/?os=any&version=17
[32] https://github.com/eclipse-sirius/sirius-web/packages/1660490?version=2024.1.2
[33] https://github.com/eclipse-sirius/sirius-web
[34] https://github.com/ObeoNetwork/Tax
[35] https://github.com/eclipse-syson/syson/packages/2020337?version=2024.3.0

```
docker run -p 5433:5432 --rm --name sirius-web-postgres \
                            -e POSTGRES_USER=dbuser \
                            -e POSTGRES_PASSWORD=dbpwd \
                            -e POSTGRES_DB=sirius-web-db \
                            -d postgres:12
```

Listing 2 Command to launch a PostgreSQL for Sirius Web

```
java -jar sirius-web-sample-application-2024.1.2.jar \
          --spring.datasource.url=jdbc:postgresql://localhost:5433/sirius-web-db \
          --spring.datasource.username=dbuser \
          --spring.datasource.password=dbpwd \
          --spring.liquibase.change-log=classpath:db/changelog/sirius-web.db.changelog.xml
```

Listing 3 Command to launch Sirius Web

```
docker run -p 5434:5432 --rm --name syson-postgres \
                            -e POSTGRES_USER=dbuser \
                            -e POSTGRES_PASSWORD=dbpwd \
                            -e POSTGRES_DB=syson-db \
                            -d postgres:12
```

Listing 4 Command to launch a PostgreSQL for SysON

```
java -jar syson-application-2024.3.0.jar \
          --spring.datasource.url=jdbc:postgresql://localhost:5434/syson-db \
          --spring.datasource.username=dbuser \
          --spring.datasource.password=dbpwd \
          --spring.liquibase.change-log=classpath:db/changelog/sirius-web.db.changelog.xml
```

Listing 5 Command to launch SysON

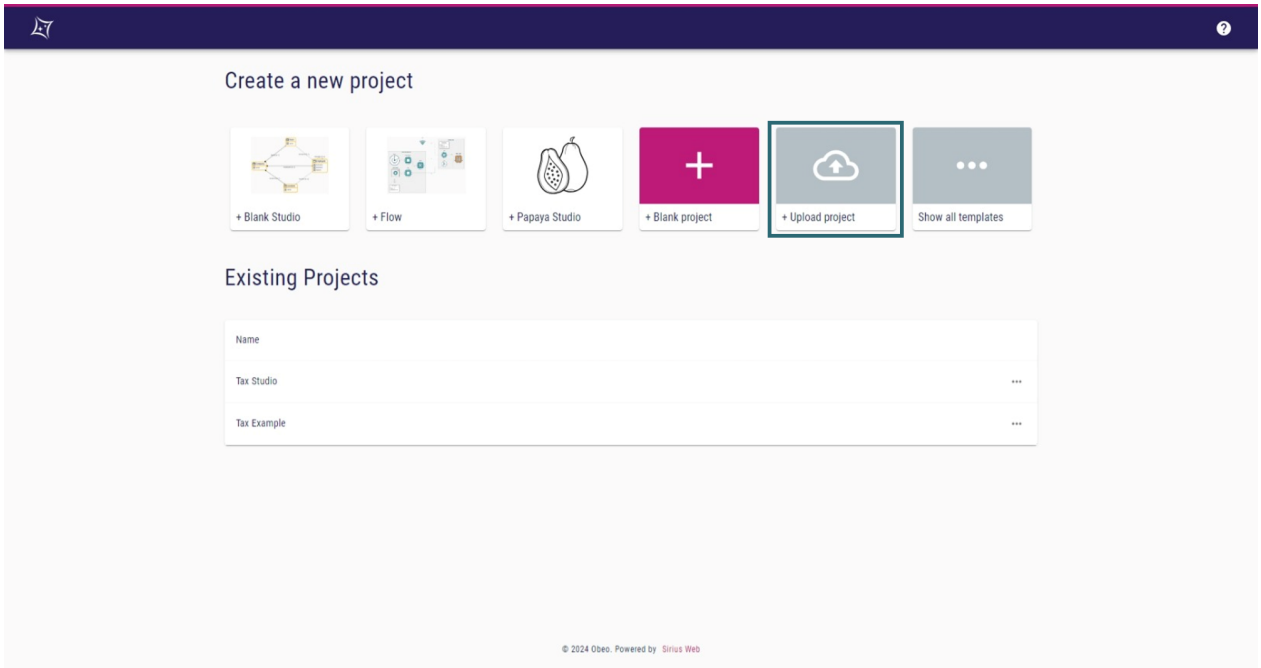

Figure 10 The homepage of Sirius Web. Boxed in blue, the tile to upload a new project.

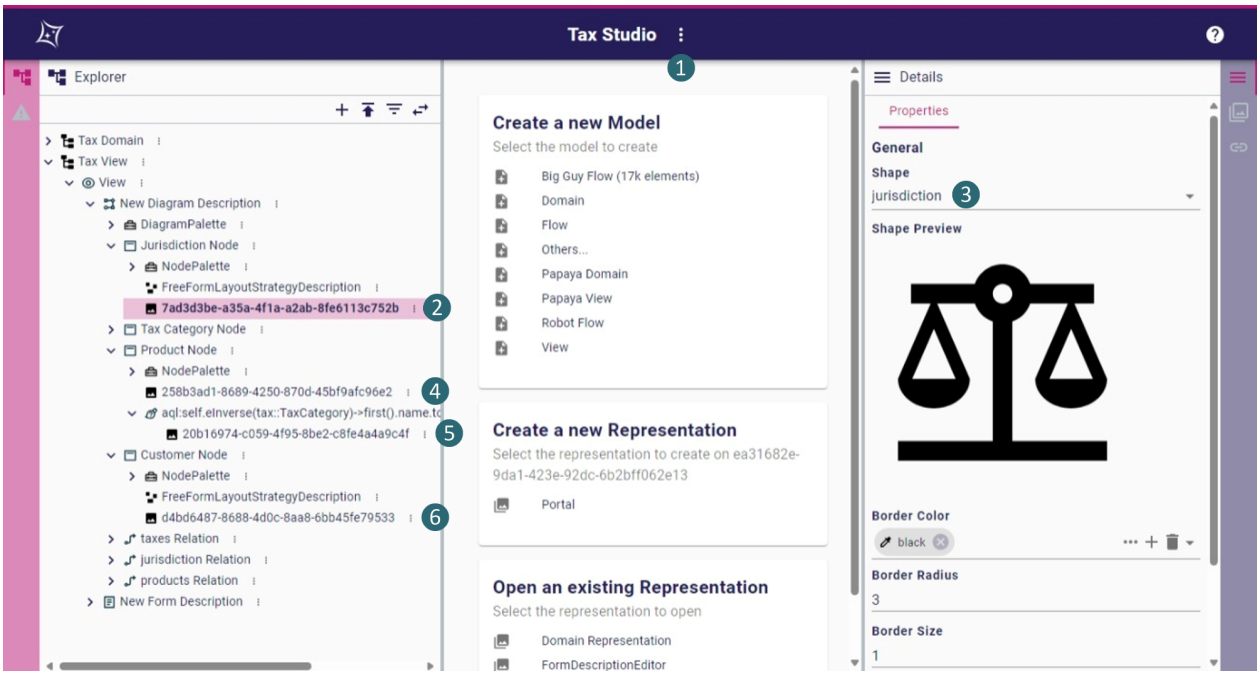

Figure 11 The Tax Studio project opened. The numbers in blue represent the ordered steps to import the images and use them to represent the different concepts.

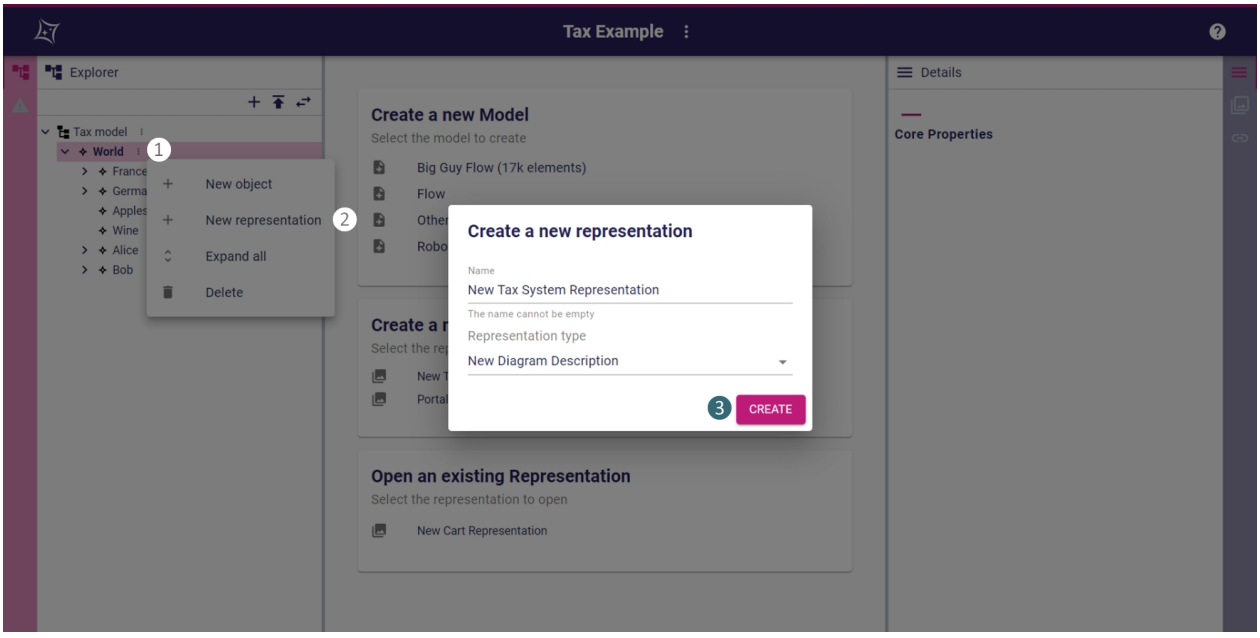

Figure 12 The steps to create a new representation.