
Proceedings Paper:
Richardson, Nathan, Kolovos, Dimitris and Garcia-Dominguez, Antonio (2024) Aconite:Towards Generating Sirius-Based Graphical Editors from Annotated Metamodels. In: Laemmel, Ralf, Pereira, Juliana Alves, Mosses, Peter D. and Mosses, Peter D., (eds.) SLE 2024 - Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering, Co-located with:SPLASH 2024. 17th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2024, Co-located with: SPLASH 2024, 20-21 Oct 2024 SLE 2024 - Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering, Co-located with: SPLASH 2024. Association for Computing Machinery, Inc, USA, pp. 16-28.

https://doi.org/10.1145/3687997.3695642

Aconite: Towards Generating Sirius-Based Graphical Editors from Annotated Metamodels

Nathan Richardson
University of York
York, United Kingdom
n.w.richardson@york.ac.uk

Dimitris Kolovos
University of York
York, United Kingdom
dimitris.kolovos@york.ac.uk

Antonio Garcia-Dominguez
University of York
York, United Kingdom
a.garcia-dominguez@york.ac.uk

Abstract

Sirius is a powerful framework for implementing graphical editors for modelling languages. Sirius can help manage model complexity by presenting the same model through multiple notations ("viewpoints" in Sirius), dedicated to different audiences and/or tasks. However, this flexibility comes at the expense of having to manually define the mapping between each viewpoint and the metamodel. This paper explores a textual notation to efficiently annotate a metamodel with such a mapping, and transform the metamodel into one or more Sirius viewpoint descriptors, with the aim to reduce the manual work required to produce and maintain Sirius-based graphical notations. We present Aconite, an open-source tool which implements this approach, and demonstrate it through the re-implementation of a common Sirius example notation, and a simplified version of a BPMN editor. Aconite includes mechanisms for automated generation of navigation expressions in common scenarios, and for inheritance of graphical styles to reduce repetition.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Graphical user interface languages**.

Keywords: Graphical Modelling, Model Transformation, Eclipse Sirius

ACM Reference Format:
Nathan Richardson, Dimitris Kolovos, and Antonio Garcia-Dominguez. 2024. Aconite: Towards Generating Sirius-Based Graphical Editors from Annotated Metamodels. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24), October 20–21, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3687997.3695642

SLE '24, October 20–21, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1180-0/24/10
https://doi.org/10.1145/3687997.3695642

1 Introduction

The defining of a domain-specific modeling language(DSML) is a common practice within the field of model-driven engineering (MDE), to enable the creation of models for a problem domain. A DSL is made up of two parts. The first part is an *abstract syntax*, which defines the concepts of the DSL and their attributes and relationships. The second part is one or more *concrete syntaxes* that modellers will use to view and manipulate the DSL concepts. Model-driven approaches are often used to develop these abstract and concrete syntaxes, in order to reduce the effort required.

One such model-driven approach for concrete syntaxes that has become commonplace is Eclipse Sirius [8]: a survey by Ozkaya and Akdur [10] showed that 43% of the respondents used Sirius to specify graphical, tree, or tabular concrete syntaxes. Sirius expects the developer to have previously defined the abstract syntax with the Ecore metamodeling language from the Eclipse Modeling Framework (EMF) [12]. The concrete syntaxes are then implemented in a Sirius *viewpoint specification model* (VSM), which maps the concepts to specific visual elements: this information is used by the Sirius-provided model editors to reconfigure themselves for a given DSML. Sirius VSMs allow for significant flexibility, allowing for representing the same underlying model in multiple notations, with the ability to define filters and organise the information differently to the metamodel (e.g. in a hierarchical notation, introducing new intermediate nodes or excluding unnecessary details).

This same flexibility from the VSMs complicates the co-evolution of the abstract and concrete syntaxes. Sirius VSMs include expressions that compute the elements to be shown within a given representation or parent element: these are called *Semantic Candidate Expressions* (SCEs). Sirius supports writing SCEs in three languages out of the box (Acceleo Query Language or AQL, Acceleo 3, and Java), but also gives the capability to integrate other languages.

The use of SCEs makes it possible for a diagram or parent element to include an arbitrary set of elements, at the will of the DSML designer: this is more flexible than the one-to-one mapping between model elements and representations from previous approaches, such as the Graphical Modelling Framework (GMF) [13]. On the other hand, it also introduces coupling between the VSM and the metamodel, which requires managing the co-evolution of two separate artefacts:

certain metamodel changes may also inadvertently break the SCEs in a VSM, and therefore the representations created with it.

One way to simplify the co-evolution of the abstract and concrete syntaxes is to describe both in the same artefact. The Eugenia tool from Kolovos et al. [6] achieved this for GMF-based concrete syntaxes, by automatically producing the models required by GMF from a set of annotations on an abstract syntax written in the Emfatic textual notation.

In this paper, we explore the possibility of using an annotated metamodel to generate a Sirius concrete syntax specification. Showing how this can aid in the co-evolution between a concrete syntax and its metamodel. Exploring how this can be achieved through targeted automation hiding parts of Sirius to reduce the learning curve and increase efficiency.

We therefore present Aconite[1], a tool that uses a model-to-model transformation of on an annotated metamodel to generate a Sirius-based graphical concrete syntax. The tool makes use of the information from the metamodel and annotations, as well as default values. This can provide initial diagrams quickly, which only requires customisation that deviates from these defaults. We demonstrate Aconite by recreating two existing concrete syntaxes to explore the level of customisation that they require, as well as exploring how some of the more advanced features of Aconite can further simplify the work involved.

2 Background: Eclipse Sirius

Sirius aims to allow graphical editors to be developed through its *Viewpoint Specification Model* (hereafter known as "the Sirius model"). This model is then interpreted by Sirius to generate each representation. The Sirius model contains one or more "descriptors", each of which encodes a representation (concrete syntax) of the abstract syntax concepts. For this paper, we shall focus on two categories of descriptions which encode the necessary components of a graphical editor: the ones describing entire diagrams, and those describing individual diagram elements. The diagram descriptions are at the top level and specify the set of objects that can act as the root object of their diagrams. To provide a sense of these descriptors and their sub-components, we will make a simple Sirius model in Section 2.2, based on the metamodel described in Section 2.1.

2.1 Project Scheduling Language (PSL)

The Project Scheduling Language's metamodel specifies the concepts relevant to the delivery of projects as well as the relationships between them. Listing 1 shows the metamodel written in the Emfatic textual notation [2]. *Project* is the top level concept, with a *title* and *description*, as well as containing a number of *People* and *Tasks* that are involved in that project. A task can contain multiple *Effort* objects, used to assign a percentage of the task to a person, as well as *Deliverables*, and it can reference any number of tasks that it depends upon. Finally, each *Deliverable* references the *Person* in charge of that deliverable.

Listing 1. Original PSL metamodel

```
@namespace(uri="psl", prefix="")
package psl;

class Project {
  attr String title;
  attr String description;
  val Task[*] tasks;
  val Person[*] people;
}
class Task {
  attr String title;
  attr int start;
  attr int duration;
  val Effort[*] effort;
  ref Task[*] dependencies;
  val Deliverable[*] deliverables;
}
class Person {
    attr String name;
}
class Effort {
  ref Task[1]#effort task;
  ref Person person;
  attr int percentage = 100;
}
class Deliverable {
  attr String title;
  attr int due;
  ref Person lead;
}
```

2.2 Representing Task Dependencies in Sirius

Sirius descriptors come in five key types: diagrams, layers, edges, containers, and nodes (shown respectively in Figure 1). Descriptors can be used to customise a representation. To show this, we will attempt to create a diagram of the *Task*s of a project containing their *Deliverable*s, with edges that represent dependencies between the different *Task*s. We will explain each descriptor and its use for this diagram.

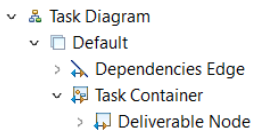

Figure 1. Diagram Element Descriptors

[1]Code available at: https://github.com/nwxrichardson/aconite

Diagrams. The aim of the diagram descriptor is to create a set of preconditions that must be met for a model element to be considered the root of one of its diagrams. The descriptor indicates a subtype of *EObject* for which it applies, by setting the *Domain Class*. For "Task Diagram" in this example, we set it to *psl::Project*[2] as the entire project should be the root.

It is also possible to set a separate *Precondition Expression*: if it evaluates to true, Sirius will allow creating a representation with the model element as its root. For example, the descriptor could only allow creating diagrams of projects which contain at least one task (*aql:self.tasks->size() > 0*)[3]. This expression would then be run by the Sirius interpreter, with *self* being the model element one is attempting to create a diagram for. For this particular example, we will not use a precondition expression as we want to edit all projects.

Layers. Sirius requires grouping nodes, edges and containers into layers. There will always be at least one "default" layer, but more complex diagram may include other layers whose visibility can be toggled by the user. As this is a simple diagram, it is enough to only use the default layer.

Containers. Container descriptors describe diagram elements which contain other diagram elements (e.g. the blue box in Figure 2), namely nodes and other containers. Each container creates a diagram region where they are the root element: they can represent any containment relationship. Container descriptors use *Domain Class* and *Precondition Expression* in the same way as diagram descriptors. However, the root of a container is selected by an SCE, unlike in a diagram where it is selected by the user.

For the example in Figure 1, the "Task Container" descriptor is within the layer descriptor. In the diagram descriptor, the variable *self* in its *SCE* refers to the root element of the diagram, and the *Domain Class* is *psl::Project*: we can access all its tasks with the *SCE aql:self.tasks*, where *tasks* is the reference on line 7 of Listing 1. For the container descriptor, we set *Domain Class* to *psl::Task* and leave *Precondition Expression* unset as we want to show all tasks. Alternatively, we could have set *aql:self.deliverables->size() > 0* [4] to get only those which contain at least one deliverable.

"Task Container" contains several sub-descriptors that are hidden due to space constraints, mostly dedicated to graphical styling (e.g. colour, shape, or borders). These graphical descriptors primarily rely on drop-down menus, tick boxes, and text boxes. One important sub-descriptor provides a label expression, written in the same way as an SCE, but produces a string. For the "Task Container" descriptor, the label expression is *aql:self.title* (the title of the *Task*).

[2]The domain class can be set in many ways, but this is set as *Package::Class*.
[3]The *aql:* prefix indicates the expression should be executed in the AQL interpreter.
[4]Here, *self* refers to the *psl::Task* being represented as a container, and not the diagram root.

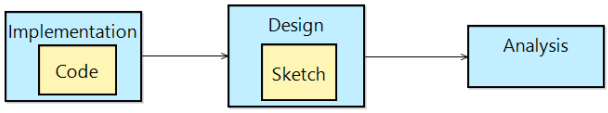

Figure 2. Task Dependency Diagram

Nodes. Node descriptors create diagram elements (e.g. the yellow boxes in Figure 2) which represent individual model elements. A node descriptor uses a *SCE* in the same way as a container descriptor, and the same goes for *Domain Class* and *Precondition Expression*. In this example, we have a "Deliverable Node" descriptor within "Task Container" whose SCE is *aql:self.deliverables* (reference on line 16 of Listing 1, i.e. the deliverables of a *Task*), with the *Domain Class* set to *psl::Deliverable*. We do not use a *Precondition Expression*. The label expression is *aql:self.title* (the title of the *Deliverable*).

Edges. An edge descriptor specifies line-based connections between any two diagram elements (e.g. the black arrow in Figure 2). They can be *reference-based* or *element-based*: while their edges appear identical to the user, they differ in how they select the three parts of an edge: source, model element, and target.

Reference-based edges are for cases where the reference to the target is in the source of the edge. Since source and model element are the same, they are set via a *Mapping*, which is a set of other descriptors that an edge can connect to. The targets are then specified via a *Target Finder Expression*, run with the model element as the *self* variable, which is represented via another *Mapping* for the target. A reference-based edge may have a *Precondition Expression* to constrain sources and targets. Our current example is a typical case for a reference-based edge, as the *dependencies* reference that we are trying to follow is contained within the source *Task*. To represent the edge, we will set the source and target *Mapping*s both to *Task* and the SCE will be set to *aql:self.dependencies*.

In contrast, an element-based edge is suited for cases where the relationship is in a separate model element from the source and the target. One possible use (not shown in the current example) would be the *Effort* class, which is a bridge between *Task* and *Person*. Such an element-based edge descriptor would have a SCE (*aql:self.tasks.effort* in this case), and a *Domain Class* (*psl::Effort* in this case). It could also have a *Precondition Expression* to constrain the set of acceptable sources and targets, as usual (not applicable to this case).

3 Proposal: Aconite

To manage the co-evolution and provide a more efficient way of specifying a Sirius-based concrete syntax, we have created Aconite: a tool based upon the successful approach

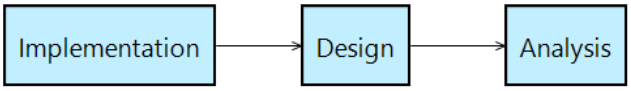

Figure 3. Initial Diagram for Aconite

applied to GMF by Eugenia, that proposes transforming an annotated metamodel to a Sirius model. The annotations to the metamodel provide the information required to define a Sirius model. Aconite is designed to require as little extra information as possible to create initial diagrams, to make best use of the fast feedback loops enabled by Sirius' interpreted nature. This heavily relies on the use of default settings, whilst still allowing the user to customise a wide array of Sirius' features. Aconite focuses on the features needed to create a graphical concrete syntax with basic editing capabilities. Below we present Aconite in the context of a concrete example, to introduce and explain some of the basic concepts.

3.1 Initial Diagram

As our first example, we will implement a diagram of the *Task* objects with edges to their dependencies, as shown in Figure 3. Listing 2 shows how this is possible. The `@aconite` annotation on line 2 indicates that this metamodel is meant to produce a Sirius model. In line 6, `@aconite.diagram` specifies that *Project* will be represented as a diagram: *name = "Project Diagram"* both sets the name within the Sirius diagram, and acts as the identifier of the descriptor within the Sirius model. This identifier is then used from line 9, which indicates that the nodes created to represent each *Task* by `@aconite.node` should be contained within the diagram. Similarly, line 14 uses the `@aconite.edge` annotation to represent the dependencies between the tasks.

The annotated metamodel shows the extent of automation that can be achieved, as these annotations did not specify any *Domain Class* nor SCE: these were inferred by Aconite. The *Domain Class* is the annotated class from the metamodel. The SCE of the *Task* node descriptor is generated by traversing from the *Project* to the *Task* via references. How this is done is explored in Section 3.7: in this section, the SCEs will be specified in the main text. In this case, the SCE only required the reference *tasks*. Aconite filled out the necessary attributes of the node and edge descriptors, getting the *Target Expression* of the edge descriptor from the reference being annotated and assigning it a *Target Mapping* for each mapping applied to the objects pointed to by the reference. This is enough for this initial diagram, as Aconite has also set each descriptor's styling using its preset defaults.

3.2 Graphical Containment

Besides the diagram above, a project manager might want to display every *Task* with its *deliverables* inside them, as

Listing 2. Annotations for a task dependency diagram

```
1  @namespace(uri="psl", prefix="")
2  @aconite
3  package psl;
4
5  @aconite.diagram(
6    name = "Project Diagram")
7  class Project { /* ... */ }
8
9  @aconite.node(name = "Task Node",
10 container = "Project Diagram")
11 class Task {
12   /* ... */
13   @aconite.edge(
14     name = "Dependency Edge",
15   container = "Project Diagram")
16   ref Task[*] dependencies;
17   /* ... */
18 }
19 // rest is unchanged
```

our Sirius example in Figure 2. Listing 3 shows how annotations would change for *Task* and *Deliverable* (everything else remaining the same). *Task* can be made a container by swapping `@aconite.node` to `@aconite.container`, with *Deliverable* being represented as a child node contained within its *Task*. Using the same *container* option as when specifying that an element is part of a diagram makes the Aconite annotations more consistent, reducing the memorisation effort involved, while allowing for setting different aspects of the description in ways that are consistent with the notion of containment. Furthermore, this consistency means that placing an element inside a container instead of a diagram does not require the user to cascade the change to other elements.

3.3 Element-Based Edges

To demonstrate the last remaining descriptor (*element-based edges*), we will produce a different diagram with the relationships between tasks and people in terms of the effort they will spend on that task. Listing 4 shows the use of basic nodes to present the *Person* and *Task* on lines 1 and 5 respectively. For an element-based edge, the annotation is similarly attached to the class as would a node or container, as the edge representing the *Effort* is similarly contained within the *Project* that underlies the diagram. Aconite is able to generate the SCE (*aql:self.references*) in the same way as for nodes and containers, but there is no viable method to guess which references to choose as the source and the target. Instead, these are to be manually specified as shown on line 14 of Listing 4.

Listing 3. Annotations for task-deliverable diagram

```
@aconite.container(
  name = "Task Container",
  container = "Project Diagram",
  color = "light_blue")
class Task {
  /* ... */
  @aconite.edge(
    name = "Effort Edge",
    container = "Project Diagram")
  ref Task[*] dependencies;
  /* ... */
}

@aconite.node(
  name = "Deliverable Node",
  container = "Task Container")
class Deliverable {
  // unchanged
}
```

Listing 4. Annotations for element-based edges

```
@aconite.node(name="Person Node",
  container="Effort Diagram")
class Person { /* ... */ }

@aconite.node(name="Task Node",
  container="Effort Diagram",
  labelExpression="title",
  color="light_blue")
class Task { /* ... */ }

@aconite.edge(name="Effort Edge",
  container="Effort Diagram",
  labelExpression="percentage",
  source="person", target="task")
class Effort {
  ref Task[1]#effort task;
  ref Person person;
  attr int percentage = 100;
}
```

3.4 Multiple Diagrams

One of the key improvements from targeting Sirius instead of GMF (which is what Eugenia targeted) is the capability to create multiple diagrams from a single model. Aconite allows for the creation of any number of diagram annotations, each creating a new diagram within Sirius. Listing 5 shows each annotation specifying their container, creating the node to only be container to be one of the diagrams as specified,

Listing 5. Multiple Diagrams with Nodes

```
@aconite.diagram(
  name="Leader Diagram")
@aconite.diagram(
  name="Project Diagram")
class Project {
  /* ... */
}

@aconite.container(
  name = "Task Container",
  container = "Project Diagram",
  labelExpression="title",
  color="light_blue")
@aconite.node(name="Task Node",
  container="Effort Diagram",
  labelExpression="title",
  color="light_blue")
class Task {
  /* ... */
}
```

this creates separate trees each with a diagram at its root. This enables multiple diagrams to co-exist within a single annotated metamodel, and produce a single VSM.

3.5 Reuse Methods

In this section we shall create a single file containing both the Effort and Project diagrams, and a new *Leader* diagram. The Leader diagram displays the leader of each deliverable within the project, to visualise the split of responsibilities or who to talk to about a deliverable. It will have two types of nodes (representing *Person* and *Deliverable*), and link them with *lead* reference-based edges as shown in Listing 6. By using these three diagrams we will show that despite having limited overlap between diagrams, it is possible to make use of the reuse methods within Aconite. Each method can be used to reduce either the amount or size of the annotations required. We will also present the benefits that this can provide for the consistency between diagrams.

3.5.1 Multiple Containment. The first way to simplify this set of annotations is by allowing the `container` option to refer to more than one container ID (which we call *multiple containment*). This allows users to easily reproduce all the design elements in both diagrams, including the nodes it contains. In Listing 7 we have to use two annotations which in essence say the same thing. Listing 8 replaces them with one, using `container = "Task Container,Leader Diagram"`. In addition to reducing the amount of code, using multiple containment can allow users to maintain the style of all of

Listing 6. Annotations for the Leader Diagram

```
@aconite.diagram(
  name="Leader Diagram")
class Project {
  attr String title;
  attr String description;
  val Task[*] tasks;
  val Person[*] people;
}

@aconite.node(name="Leader Node",
  container="Leader Diagram",
  color = "light_purple")
class Person {
  attr String name;
}

@aconite.node(name = "artefact Node",
  container = "Leader Diagram",
  labelExpression = "title")
class Deliverable {
  attr String title;
  attr int due;
@aconite.edge(name="Lead Edge",
  container="Leader Diagram",
  color = "black")
  ref Person lead;
}
```

Listing 7. Annotations without multiple containment

```
@aconite.node(
  name = "Deliverable Node",
  container = "Task Container",
  labelExpression = "title")
@aconite.node(name = "artefact Node",
  container = "Leader Diagram",
  labelExpression = "title")
class Deliverable {
@aconite.edge(name="Lead Edge",
  container="Leader Diagram",
  color = "black")
  ref Person lead;
}
```

the generated through syntax evolution, meaning users only need to change it in one place.

3.5.2 Inheritance. Aconite annotations can inherit the attribute values from another annotation, avoiding repetition and improving consistency. Listing 9 shows two annotations which aim to represent the same model element in

Listing 8. Single annotation with multiple containment

```
@aconite.node(name = "Deliverable",
  container = "Task Container,
    Leader Diagram",
  labelExpression = "title")
class Deliverable {
@aconite.edge(name="Lead",
  container="Leader Diagram",
  color = "black")
  ref Person lead;
}
```

Listing 9. Identical Annotation for node and container

```
@aconite.container(name =
  "Task Container",
  container = "Task Diagram",
  labelExpression="title",
  color="light_blue")
@aconite.node(name="Task Node",
  container="Effort Diagram",
  labelExpression="title",
  color="light_blue")
class Task {
  /* ... */
}
```

the same way: the only change being one is a node and the other is a container. Their visual descriptions are therefore the same, whilst differing in *name* for uniqueness, and in *container* to ensure the correct structure. Listing 10 shows how this can be converted to a single attribute on line 8 using the `extend` option, which will import all the attributes of the `@aconite.container` that have not been set in the `@aconite.node`.

Inheriting styles in this way simplifies maintaining consistency between the two annotations, representing the model element in the same way each time. It is worth noting that styles can be inherited across any two Aconite annotations (e.g. across classes). It is possible to have one annotation set the attribute for many other diagram elements.

To further this capability Aconite includes the concept of abstract annotations, which use this syntax:

```
@aconite.node(~abstract = "")
```

The above annotation would not be turned into a node descriptor by Aconite, but its attributes could still be inherited by other annotations. Our example in Listing 10 could be used for all of the classes that have a title that is going to be used as the *labelExpression* to handle the styling requirements in one place, such as underlining each label.

Listing 10. A node extending a container

```
@aconite.container(
  name = "Task Container",
  container = "Task Diagram",
  labelExpression="title",
  color="light_blue")
@aconite.node(name="Task Node",
  container="Effort Diagram",
  extend = "Task Container")
class Task {
  /* ... */
}
```

3.6 Model Validation

The move from a Sirius model to a textual language loses the enforced structure of the Viewpoint Specification Editor, since the textual editor cannot ensure that the specified concrete syntax will conform to the expectations of Aconite. Aconite therefore provides some automated validation in the form of a set of Epsilon Validation Language (EVL) [5] validation rules.

There are rules that ensure that each annotation has a unique name which can act as the target for the references. Each Aconite option that references these unique names (such as `container`) is checked to ensure it points at an existing annotation. This includes specific checks for features that require the referenced annotation to be of a specific type (e.g. the container of an edge must be a diagram).

The validation checks properties that require selecting from a predefined list are given valid values, such as `colour`. Similar rules are written for VSM properties that expect a non-string data type: these check that the provided values are convertible to the correct data type for their properties. For example, if the option requires an integer, `"two"` would not be acceptable.

3.7 Generating Semantic Candidate Expressions

Aconite includes an algorithm to automatically compute the SCE for a given descriptor in simple cases, based on the target of its source annotation. One such case is when the SCE only requires following one layer of referencing, and there was only one possible option to choose from. This was true for most of the SCEs in previous sections' examples. One exception to this is on line 17 of Listing 6 for the *artefact* of the *Leader Diagram*: this required following multiple references, as the class *Project* did not have any direct references of type *Deliverable*, leading to the SCE *aql:self.tasks.deliverables*.

To generate these SCEs, the algorithm performs a breadth-first search for a path from a source class to a target class on the metamodel, with classes as nodes and their references as directed edges. We enforce that the path has a minimum length of one (i.e. at least one reference is followed), and that Aconite does not simply return *aql:self* when the source and target types are the same.

Algorithm 1 Route from a source class to a target class

```
1:  procedure GETROUTE(src: EClass, tgt: EClass): String
2:    seen ⟵ {src}                    ▷ Track types visited so far
3:    targets ⟵ [tgt] + tgt.eAllSuperTypes
4:    rBT ⟵ nearestRefByType(src)
5:    prevS ⟵ 0
6:    while (rBT.keySet() ∩ targets) = ∅ ∧ |seen| > prevS do
7:      prevS ⟵ |seen|              ▷ Store previous size of seen
8:      seen ⟵ seen ∪ rBT.keySet()
9:      newRBT ⟵ Map()
10:     for t, query in rBT do      ▷ Expand search one level
11:       nextRBT ⟵ nearestRefByType(t)
12:       for n, subquery in nextRBT do
13:         if n ∉ newRBT.keySet() then
14:           newRBT.put(n, query + "." + subquery)
15:         end if
16:       end for
17:     end for
18:     newRBT.removeAll(seen)
19:     rBT ⟵ newRBT
20:   end while
21:   if |seen| = prevS then
22:     return ".eAllContents()"
23:   else
24:     return select({tgt}, rBT)
25:   end if
26: end procedure
```

The main procedure *getRoute()* is shown in Algorithm 1, which takes a source and a target *EClass*, and returns the route as a *String*. The returned *String* is the last part of an SCE (the bold text in *aql:self.**tasks.deliverables***), with *tasks* and *deliverables* each being references. The procedure begins with Line 2 initialising the set of visited nodes *seen* to include *source*. Line 3 initialises a fixed set of *targets*, with the transitive closure of the supertypes of the target plus itself. Line 4 sets the value of *rBT* ("reference by type") which contains a map from each *EClass* to the computed route up until that point, which can be returned if it is for one of the *targets*.

The rest of Algorithm 1 runs within a loop, which is exited if any entry of *rBT* is a member of *targets*, as a route has been found. It will also be exited if *seen* did not grow in size, as this indicates that no route can exist. The loop itself then runs through *rBT* to create a *newRBT*, which contains all of the routes following one further layer of references. For each member of *rBT*, it gets the *nextRBT* using the same *nearestRefByType()* procedure used on the source. Then for each member of *nextRBT*, its *subquery* is added to the end of the route of the member of the *rBT*. The first entry for each

EClass key is added to the *newRBT* map, and at the end of each loop this becomes the new *rBT*.

Algorithm 2 Map of nearest references by type

```
1:  procedure NEARESTREFBYTYPE(s: EClass): Map
2:      return nearestRefByType2(s, Map())
3:  end procedure
4:  procedure NEARESTREFBYTYPE2(s: EClass, m: Map)
5:      for r in s.eReferences do
6:          if r.eType ∉ m.keySet() then
7:              m.put(r.eType, r.name)
8:          end if
9:      end for
10:     for t in s.eSuperTypes do
11:         m ← nearestRefByType2(t, m)
12:     end for
13:     return m
14: end procedure
```

Algorithm 2 shows the procedure *nearestRefByType()*, used within *getRoute()* to get all of the references by their type. It does this by running through all of the references from the input type, each time checking that no reference of the same type has already been added. The same is done with all of the target's supertypes, starting from the direct supertypes, then their supertypes, and so on.

Algorithm 3 Selecting among available routes

```
1:  procedure SELECT(ts: Set<EClass>, rBT: Map): String
2:      for t in ts do
3:          if t ∈ rBT.keySet() then
4:              return rBT.get(t)
5:          end if
6:      end for
7:      return select(ts.collect(t | t.superTypes(), rBT)
8:  end procedure
```

Finally, Algorithm 3 selects a route from the map: while the algorithm would prefer to find a route to the exact target type of the reference, it could have to fall back to a route to one of its supertypes. This is achieved recursively, starting from the target class: if *rBT* contains the type, it returns the corresponding route. Otherwise, it will perform a breadth-first search among the direct supertypes, and then their supertypes as well, until a route is returned. By doing this, it selects the route to the most specific type in relation to the target type.

For the route from *Project* to *Deliverable*, the above algorithms would combine into these steps:

1. We begin in *getRoute()* by setting *seen* to a set containing *Project*, and *targets* to a set with *Deliverable*.
2. From *Project*, for each reference *nearestRefByType()* adds to *rBT* an entry to a map from the class of the reference to a path, producing two entries: (Task, ".tasks") and (Person, ".people").
3. As none of the keys in *rBT* are *Deliverable*, we add *Task* and *Person* to *seen* and use *nearestRefByType()* on both entries in *rBT*. The updated *rBT* has two entries: (Effort, ".tasks.effort") and (Deliverable, ".tasks.deliverables"), from which we remove *Task* as it is in *seen*.
4. The *rBT* map includes *Deliverable*, which is among our target types, so it is passed to *select()*, which returns the route we expect (*".tasks.deliverables"*).

4 Evaluation

To obtain an initial assessment of Aconite's current capabilities, we have created annotated metamodels which can generate Sirius-based concrete syntaxes that match a pre-existing syntax. We were able to reproduce both the Family notation from the Sirius tutorial [7] and the BPMN subset [9] used for demonstration by [3] and recreated in the paper by Kolovos et al. [4]. Between the two of them, it will be possible to contrast the simplicity of the two sets of annotations in terms of their respective metamodels and show how many of the SCEs can be automatically generated.

4.1 Family Notation

The first concrete syntax that we aimed to recreate was the *basicFamily* example from the Sirius tutorial by Obeo [7]. As shown in Figure 4, its models represent families, with diagrams that show each member as a node, with edges between family members indicating parent-child relationships, and node images indicating sex.

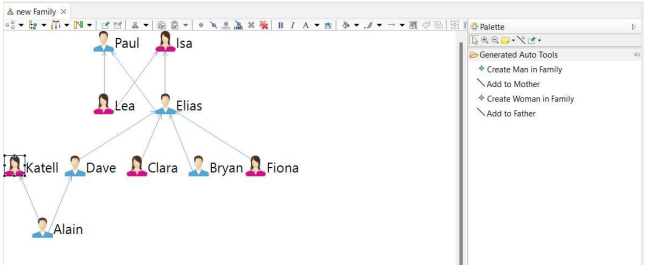

Figure 4. Sirius Basic Family Example [7]

4.1.1 Family Diagram. The Aconite annotated metamodel seen in Listing 11 implements the diagram shown in Figure 4, showing Aconite's capability to reproduce existing Sirius-based notations. These annotations do not require the manual specification of the the SCEs, saving effort and reducing the maintenance needed (e.g. if a reference was renamed). This example also demonstrates some of the styling abilities of Aconite, such as the inclusion of images instead of the default boxes through specifying the shape and the file

location. Each image's path can be specified in two parts: the first is shown in line 2 and specifies the folder, and the second is specified by the `imagePath` option in lines 30 and 37.

Outside of styling, the only option that needed to be set is the *sourceNode*, which would also need specifying in the Sirius model. This has a slight downside, as unlike the Sirius model this will not be kept up-to-date with the name of the mapped descriptor, requiring the user to manually update the option to match. However, it replicates the creation tools on the right of Figure 4 automatically, as they can be generated in a similar way to the SCE. Aconite has also generated tools to add edges and edit people's names through their diagram labels.

4.1.2 Relationship Diagram.

In addition to this family tree diagram, the Sirius model that we selected to replicate includes a diagram representing all of the relationships of a selected person. Figure 5 shows an example of this diagram for the person *Elias*, with the containment of the green box representing his children. Attached to the outside of the box are two border nodes representing his parents. Finally, the yellow box represents the siblings of Elias, as indicated by its label.

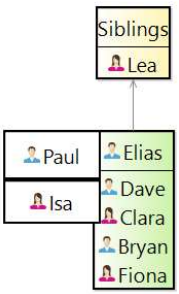

Figure 5. Relationship Diagram for Elias

Listing 12 shows the annotated metamodel replicating the Sirius model, which shows how this diagram works entirely based on one *Person* class. This use of a single class limits the applicability of the SCE generation algorithm in Aconite, requiring the manual specification of many SCEs, as Aconite does not distinguish between different references of the same class.

However, this annotation still has benefits over using the Sirius editor, beginning with style inheritance, which makes it easier to set the defaults for this diagram. This reduces repetition across the specification. Aconite also simplifies the writing of SCEs down to focusing on the references, automatically adding *aql:self.* as required by Sirius.

4.2 BPMN Model

The second syntax to be recreated was a subset of the Business Process Modelling Notation (BPMN) by OMG [9]. BPMN provides a way of representing the steps, decisions, and message flows that exist within a business process. Figure 6 shows an implementation of a process to notify a client by email in an abstract manner. Each action is a node with solid arrows representing the next step in the process, and dashed arrows representing a message. The diamond represents a exclusive decision point, and the circles represent the start and the end of the process.

Listing 11. Family Diagram

```
1  @aconite(iconFolder =
2    "aconite.family/icons/")
3  package basicfamily;
4
5  @aconite.diagram(name = "Family")
6  class Family {
7    attr String name;
8    val Person[*] members;
9  }
10
11 abstract class Person {
12   attr String name;
13   ref Person[*]#parents children;
14   ref Person[0..2]#children parents;
15 @aconite.edge(name = "Mother",
16   container = "Family" ,
17   sourceNode = "Man, Woman",
18   labelExpression = "",
19   color = "purple")
20   derived ref Woman mother;
21 @aconite.edge(extend = "Mother",
22   name = "Father",
23   color = "blue")
24   derived ref Man father;
25 }
26
27 @aconite.node(name = "Man",
28   container = "Family",
29   shape = "image",
30   imagePath = "man.svg",
31   borderSize= "0")
32 class Man extends Person {
33 }
34
35 @aconite.node(extend = "Man",
36   name = "Woman",
37   imagePath = "woman.svg")
38 class Woman extends Person {
39 }
```

Listing 12. Relationship Diagram annotated metamodel

```
@aconite.diagram(name = "Relationships")
@aconite.node(name = "relation",
  ~abstract = "true",
  color = "white",
 showIcon = "true",
  childrenPresentation = "List")
@aconite.container(extend = "relation",
  name = "Person",
  sce = "",
  foregroundColor = "light_green",
  container = "Relationships")
@aconite.border(extend = "relation",
  sce="parents",
  name = "Parent",
  container =  "Person")
@aconite.node(extend = "relation",
  name = "Child",
  container = "Person")
@aconite.container(extend = "relation",
  name = "Siblings",
  sce = "",
  labelValue = "Siblings",
  foregroundColor = "light_yellow",
  container = "Relationships",
  showIcon = "false")
@aconite.node(name = "Sibling",
  container = "Siblings",
  sce =
"parents.children->excluding(self)",
  showIcon = "true")
abstract class Person {
  attr String name;
@aconite.edge(name = "SiblingsEdge",
  creatable = "false",
  labelExpression = "",
  color = "gray",
  container = "Relationships",
  targetFinderExpression="aql:self",
  sourceNode = "Person",
  targetNode = "Siblings")
  ref Person[*]#parents children;
  ref Person[0..2]#children parents;
  derived ref Woman mother;
  derived ref Man father;
}
```

The BPMN abstract syntax can be broken down into four categories of classes excluding the top level class[5]: the swim-lanes, connecting objects, flow objects, and artefacts. The

[5]Which only exists as a top-level container of the other objects.

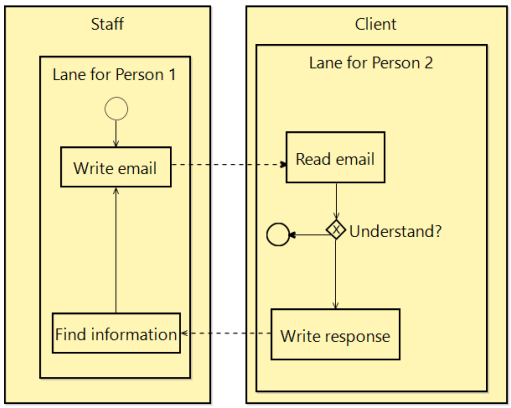

Figure 6. BPMN Diagram: Notifying a client by email

Listing 13. BPMN: Containers

```
@aconite.diagram(name = "BPD")
class BusinessProcessDiagram {
  val BPMNElement[*] elements;
}
@aconite.container(name =
  "Lane Container",
  container = "BPD, Group, Pool")
class Lane extends Swimlane {
  val FlowObject[*] flowObjects;
}
@aconite.container(name = "Pool",
  container = "BPD, Group")
class Pool extends Swimlane {
  val Lane[*] lanes;
}
@aconite.container(
  container = "BPD",
  name = "Group")
class Group extends artefact {
  val BPMNElement[*] elements;
}
```

swim-lanes are one of the containers which introduce a hierarchical structure in a BPMN model. This can be seen in Listing 13, which shows the *Lane* at the bottom that can be contained by all of the others, and the diagram at the top contained by none. This is simpler than with a Sirius model, which would require creating multiple import descriptors, whereas Aconite can generate all the relevant descriptors from the different choices of `container`.

Listing 14 contains the annotations for one of the three connecting objects, as outside of their styling they all have the same behaviour. This is represented by the abstract annotation named *Link*, which sets the source and target based

Listing 14. BPMN: Connections

```
@aconite.edge(container = "BPD",
  name="Link",
  ~abstract = "true",
  source = "from",
  target = "to",
  color = "black")
abstract class ConnectingObject
    extends BPMNElement {
  ref FlowObject from;
  ref FlowObject to;
}
@aconite.edge(extend = "Link",
  lineStyle = "dash",
  name="Message Flow")
class MessageFlow
    extends ConnectingObject {
}
```

Listing 15. BPMN: Flow Objects

```
@aconite.node(~abstract = "true",
  container = "Lane",
  name = "Flow",
  shape = "image",
  borderSize = "0")
abstract class FlowObject
    extends BPMNElement {
}
@aconite.node(extend = "Flow",
  name = "XOR",
  imagePath = "xor-gateway.svg")
class XOR extends Gateway {
}
```

on a abstract class that they all inherit from. The same process is shown for the flow objects in Listing 15, in which the only thing that is added is the *imagePath*. The use of style inheritance for these two types of diagram elements avoids unnecessary repetition, and improves consistency by sharing a single source for the features that are the same. Finally, Listing 16 shows the artefacts and activities which are nodes within the lane which only require specifying the styling information.

4.3 Discussion

In order to understand the potential time savings from Aconite, we will compare the number of Aconite annotations required to produce a Sirius model of a certain complexity. This is because every element and property that the user does not have to understand and define themselves implies a certain amount of time savings.

Listing 16. BPMN: artefacts and Activities

```
@aconite.node(name = "Activity",
  container = "Lane")
class Activity
extends FlowObject {
}
abstract class artefact
    extends BPMNElement {}
@aconite.node(container = "Lane",
  name="Data Object",
  shape = "image",
  imagePath = "data-object.svg")
class DataObject
    extends artefact {
}
```

When using Aconite to generate the Sirius family editor, we need to specify 14 annotations with 56 properties. These generate the necessary 71 elements and 156 properties in the Sirius model. These include 46 elements specifying how Sirius should create and edit elements within the diagram.

Even in the most minimalist scenario where the Sirius model only provided representations and did not provide editing tools, the Sirius model would require 25 elements and 74 properties, which is higher than the number of annotations and properties required by Aconite for a full-featured Sirius model.

Beyond replacing the need for the additional 46 elements, Aconite lowers the initial learning curve of producing Sirius-based editors by removing the need to learn about the mechanisms behind the editing tools. This allows new users to focus on the diagrams, and provides a resource for them to learn from as they advance with Sirius.

For BPMN, Aconite generates 213 elements with 425 properties from just 18 annotations with 53 properties. Specifying this representation in 145 elements using Sirius' *Reuse Mappings* is possible. However, doing so introduces complexity to specifying the editors, which would interfere with the user's ability to understand and edit them. Without the editors, this could make the smallest specification have 43 elements with 91 properties.

Overall, Aconite can realise the representations with fewer properties even in the worst-case scenarios. The precise extent to which these reductions can affect the time it takes for users to make or edit representations would need to be measured through user studies as set out in Section 7

5 Current Limitations

The current iteration of Aconite has three main limitations, the first being unable to automatically generate some complex SCEs, such as those in the *Relationships Diagram* in Section 4.1.2. There are cases where it will not be possible to predict the intentions of the designer, and therefore some intervention from the designer will be required. One possible option would be allowing the user to indicate preferences, in order to break ties between paths of equal length.

The second limitation is that Aconite does not cover every aspect of the Sirius model: for example, it does not allow for manual specification of tools. This will require exploring the methods for adding the tools to the VSM, either through user-provided *polishing transformations* (as named by Eugenia) or further annotation. However, for now this limitation has been slightly mitigated through the automatic generation for basic edit functions.

The final limitation of Aconite is the limited IDE integration: at the moment, the Aconite annotations are only validated accoring to the rules in Section 3.6 at the moment the user attempts to run the model transformation. Aconite could benefit from extending the Emfatic editor with live validation and automated refactoring capabilities, such as renaming the name of an annotation in a way that propagates across its references, or renaming a class in a way that preserves existing manually-written SCEs.

6 Related Work

As mentioned previously, this work is inspired by the approach of the Eugenia [4] tool while targeting Sirius instead of GMF. The two implementations therefore share many capabilities for handling model representations where only one reference needs to be followed at a time, and only one diagram is needed. Aconite goes beyond this, allowing more complex diagrams using multiple references, allowing for diagrams to diverge from the original model.

A similar approach was also taken by Rani et al. [11] to use a Eugenia-like approach for Sirius. However, their approach relied on the manual specification of all elements. It does not make use of any automated SCE generation algorithm to provide an efficient and robust method for evolving metamodels and concrete syntax design. Furthermore, alongside not providing model validation, it also has the issue of generating the Sirius model via a model-to-text transformation to XMI [9]. This increases the risk of producing invalid models, e.g. by not escaping characters to meet XMI syntactic rules, or producing XMI sources that do not conform to the latest version of the Sirius abstract syntax.

Another related work by Bedini et al. [1] uses a Sirius editor to generate both the metamodel and the Sirius model by transforming their model. To edit this model, they create a graphical concrete syntax instead of a textual syntax. The transformation is able to create its diagrams from a single artefact, as well as demonstrating the possibilities of integrating Java calls on top of AQL. However, similar to Eugenia, it does not allow defining diagram relationships that traverse multiple references, which is allowed by Sirius and Aconite.

7 Conclusions and Future Work

We have demonstrated a first version of an approach which can reduce the work involved in creating a Sirius-based concrete syntax, by attaching annotations to parts of a metamodel specified using a textual notation. This shows the potential to deliver productivity and upkeep benefits when compared with standard use of Sirius, including more accessible differencing in version control systems. This is by virtue of Emfatic being a more readable textual format, compared with the XMI serialisation of a model.

Whilst Aconite does a great deal to simplify the process of designing in Sirius, a significant amount of work still remains in order to improve the coverage of the Sirius feature-set. Future versions of Aconite will add the capability to specify the set of tools which edit Sirius models. We also plan to implement a transformation that would produce the Aconite version of an existing Sirius model, allowing for transformations in both directions across Aconite and Sirius. This bidirectional approach would give users the choice of editor depending on the task they are trying to perform, and further Aconite's capabilities to provide version control support to Sirius.

We also plan to undertake user studies to assess the time it takes to specify the concrete syntax between the approach outlined in this paper and the Sirius viewpoint editor, and validate that Aconite produces the expected gains in productivity. The studies will include users with different levels of experience on Sirius and DSMLs, and exploring the time taken to update the concrete syntax when the metamodel changes, or the requirements of the concrete syntax evolve.

Acknowledgements

The work in this paper has been funded through an EPSRC Industrial CASE Studentship from Rolls-Royce (reference 220035), and the SCHEME InnovateUK project (contract no. 10065634).

References

[1] Francesco Bedini, Ralph Maschotta, and Armin Zimmermann. 2021. A generative Approach for creating Eclipse Sirius Editors for generic Systems. In *2021 IEEE International Systems Conference (SysCon)*. 1–8. https://doi.org/10.1109/SysCon48628.2021.9447062 ISSN: 2472-9647.

[2] Eclipse. 2024. *Emfatic*. Retrieved 2024-02-25 from https://eclipse.dev/emfatic/

[3] Amine El Kouhen, Cedric Dumoulin, Sébastien Gerard, and Pierre Boulet. 2012. Evaluation of modeling tools adaptation. (June 2012). Retrieved 2024-04-26 from https://hal.science/hal-00706701/

[4] Dimitrios S. Kolovos, Antonio García-Domínguez, Louis M. Rose, and Richard F. Paige. 2017. Eugenia: towards disciplined and automated development of GMF-based graphical model editors. *Software & Systems*

Modeling 16, 1 (Feb. 2017), 229–255. https://doi.org/10.1007/s10270-015-0455-3

[5] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. 2009. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Rigorous Methods for Software Construction and Analysis: Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday*, Jean-Raymond Abrial and Uwe Glässer (Eds.). Springer, Berlin, Heidelberg, 204–218. https://doi.org/10.1007/978-3-642-11447-2_13

[6] Dimitrios S. Kolovos, Louis M. Rose, Saad Bin Abid, Richard F. Paige, Fiona A. C. Polack, and Goetz Botterweck. 2010. Taming EMF and GMF Using Model Transformation. In *Model Driven Engineering Languages and Systems (Lecture Notes in Computer Science)*, Dorina C. Petriu, Nicolas Rouquette, and Oystein Haugen (Eds.). Springer, Berlin, Heidelberg, 211–225. https://doi.org/10.1007/978-3-642-16145-2_15

[7] Obeo. 2024. Sirius/Tutorials/BasicFamily - Eclipsepedia. Retrieved 2024-04-26 from https://wiki.eclipse.org/Sirius/Tutorials/BasicFamily

[8] Obeo and Eclipse. 2024. Sirius | Home. Retrieved 2023-07-12 from https://eclipse.dev/sirius/

[9] OMG. 2010. About the Business Process Model And Notation Specification Version 2.0. Retrieved 2024-04-26 from https://www.omg.org/spec/BPMN/2.0/

[10] Mert Ozkaya and Deniz Akdur. 2021. What do practitioners expect from the meta-modeling tools? A survey. *Journal of Computer Languages* 63 (April 2021), 101030. https://doi.org/10.1016/j.cola.2021.101030

[11] Fatima Rani, Pablo Diez, Enrique Chavarriaga, Esther Guerra, and Juan De Lara. 2020. Automated migration of EuGENia graphical editors to the web. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. ACM, Virtual Event Canada, 1–7. https://doi.org/10.1145/3417990.3420205

[12] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework*. Pearson Education.

[13] Christoph Wienands and Michael Golm. 2009. Anatomy of a Visual Domain-Specific Language Project in an Industrial Context. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS '09)*. Springer-Verlag, Berlin, Heidelberg, 453–467. https://doi.org/10.1007/978-3-642-04425-0_35

Received 2024-07-01; accepted 2024-08-30