
Proceedings Paper:
Kolovos, Dimitris orcid.org/0000-0002-1724-6563, De La Vega, Alfonso and Cooper, Justin (2020) Efficient Generation of Graphical Model Views via Lazy Model-to-Text Transformation. In: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20). ACM

https://doi.org/10.1145/3365438.3410943

Efficient Generation of Graphical Model Views via Lazy Model-to-Text Transformation

Dimitris Kolovos
Department of Computer Science
University of York
York, UK
dimitris.kolovos@york.ac.uk

Alfonso de la Vega
Department of Computer Science
University of York
York, UK
alfonso.delavega@york.ac.uk

Justin Cooper
Department of Computer Science
University of York
York, UK
justin.cooper@york.ac.uk

ABSTRACT

Producing graphical views from software and system models is often desirable for communication and comprehension purposes, even when graphical model editing capabilities are not required – because the preferred editable concrete syntax of the models is text-based, or for models extracted via reverse engineering. To support such scenarios, we present a novel approach for efficient rule-based generation of transient graphical views from models using lazy model-to-text transformation, and an implementation of the proposed approach in the form of an open-source Eclipse plugin named Picto. Picto builds on top of mature visualisation software such as Graphviz and PlantUML and supports, among others, composite views, layers, and multi-model visualisation. We illustrate how Picto can be used to produce various forms of graphical views such as node-edge diagrams, tables and sequence-like diagrams, and we demonstrate the efficiency benefits of lazy view generation approach against batch model-to-text transformation for generating views from large models.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**;
• **Information systems → Process control systems**.

KEYWORDS

Model Visualisation, Graphical Modeling

1 INTRODUCTION

Being able to produce graphical views of software and system models is often desirable in model-driven engineering settings. Such views are useful, for example, to explore reverse-engineered models [3, 12] from different viewpoints of interest, to visualise relationships between heterogeneous models, and to facilitate presentation of text-based models to a wider audience of stakeholders [13]. Common approaches for producing graphical representations of models include (a) implementing a graphical editor using a framework such as Sirius, GMF, or Graphiti; (b) implementing a bespoke graphical viewer using frameworks such as Zest, GEF or JavaFX; and (c) using batch model-to-text transformation to generate textual artefacts (e.g. Graphviz graphs, HTML pages), which can be subsequently rendered in a web browser.

In this paper, we discuss scenarios in which the model visualisation methods above are applicable, as well as their main strengths and weaknesses. We then present a novel Eclipse-based framework, called Picto, for producing transient views from models conforming to different metamodels and modelling technologies, by lazily transforming them into textual formats such as Graphviz, PlantUML, SVG and HTML, which are subsequently rendered in an embedded browser. We demonstrate the building blocks and capabilities of Picto through a running example and we showcase how it can be used to produce non-trivial views (e.g. class-diagram-like, tabular, sequence-diagram-like) from heterogeneous models. We also evaluate the performance and scalability benefits of the lazy transformation approach employed by Picto, compared to visualisation via batch model-to-text transformation.

The contributions of this paper are:

- A critical review of commonly used techniques for producing graphical views from models (Section 2);
- A novel approach for generating transient graphical views from models via lazy model-to-text transformation and an implementation of the proposed approach in the context of the Picto open-source tool (Section 3);
- An experimental evaluation of the benefits of lazy – compared to batch – model-to-text transformation, for model visualisation (Section 4).

Section 5 reviews related work, and Section 6 concludes the paper and discusses future work.

2 BACKGROUND AND MOTIVATION

In this section we review common approaches for producing graphical views from models and we highlight their main strengths and weaknesses. For cohesion, we focus on the Eclipse Modelling Framework (EMF), as an example of a comprehensive ecosystem that offers interoperable facilities for metamodelling, modelling, and for graphical and textual model editing. However, the discussion is also valid for other language workbenches such as as JetBrains MPS [21], Spoofax [20, 22], MetaEdit+ [9, 10] or the Whole Platform [18] that offer similar facilities too.

Developing a Graphical Editor. The Eclipse modelling ecosystem includes sophisticated frameworks such as Sirius[1], GMF[2] and Graphiti[3] for developing graphical editors for EMF-based languages. As such, when there is a need to visualise models, one option is to use one of these frameworks to develop a graphical (e.g. node-edge, tabular, tree-based) editor for the language that the models conform to and then use the editor to hand-craft views of interest. The main appeal of this approach is that users have complete control over the content and the appearance (fonts, colours, positions of nodes/edges) of the constructed views.

[1] https://eclipse.org/sirius
[2] https://eclipse.org/modeling/gmp
[3] https://eclipse.org/graphiti

On the other hand, the visualisation options provided out of the box by graphical modelling frameworks – both within and beyond Eclipse – are relatively limited compared to the wealth of visualisations (e.g. 3D graphs, heatmaps, treemaps) available in the broader web-based data visualisation ecosystem. In addition, graphical modelling frameworks seek to provide built-in model editing capabilities. To offer this, the graphical syntaxes of these frameworks are geared towards one-to-one mappings (or close) between model elements and syntax symbols, so that edits at the graphical level can be unambiguously propagated to semantic elements. This often makes visualisations which involve aggregating multiple elements into a single graphical symbol challenging to achieve. Last but not least, the complexity and learning curve of such frameworks is non-negligible, which is more palatable when their full range of capabilities (i.e. editing) are required, but can be a substantial overhead when only viewing facilities are needed.

Developing a Bespoke Graphical Viewer. To avoid the complexity overhead and restrictions imposed by graphical modelling frameworks, an alternative is to implement read-only viewers for the models of interest using lower-overhead and more generic frameworks such as Eclipse's Graphical Editing Framework or JavaFX. The main appeal of this approach is that viewer developers are not bound by the assumptions and restrictions of graphical modelling frameworks, and can produce graphical views of arbitrary complexity. Also, in a bespoke viewer, views can be produced lazily (on demand), which is useful for visualising large models. Where node-edge graphical representations are required, graph visualisation and auto-layout capabilities can be reused from open-source libraries such as Eclipse ELK [19], Zest [4], GraphStream[4], or commercial alternatives such as yFiles[5].

The main challenge with this approach is that it requires a substantial amount of hand-written code to implement – beyond the visualisations themselves – commonly-needed features such as the ability to navigate back and forward between views, to zoom in and out, to show/hide elements on demand by applying layers/filters and to export generated views as image files for embedding into reports.

Visualisation by M2T Transformation. Another commonly employed technique for model visualisation in the literature [1, 2, 8] is the use of model-to-text transformation to generate views in textual syntaxes such as Graphviz [7] and PlantUML [14] – which can be transformed into auto-laid-out SVG graphs using mature off-the-shelf tools – or even directly in SVG/HTML if no auto-layout capabilities are required (e.g. for form-based and tabular views or when the coordinates of view elements can be computed from information in the model).

The main appeal of this approach is the delegation of much of the "heavy lifting" to powerful tools such as Graphviz, PlantUML, and to JavaScript libraries such as D3.js[6], TheeeJS[7], and Google Charts[8], which support for a wealth of ready-made graphical notations and widgets. In addition, views produced in this way can be explored through standard web-browsers, which provide out-of-the-box support for commonly-needed features such as navigating back/forward between views, zooming in/out, and exporting views as images.

On the flip side, using a model-to-text transformation to generate all possible views from a model upfront, can be wasteful when only a small number of views is actually inspected by users between consecutive executions of the transformation. This can become a usability issue too as the size of models and the number of views grow and transformation execution times start exceeding a few seconds. This is demonstrated experimentally in Section 4. Also, while features such as zooming in/out, exporting images etc. are provided for free by web browsers, supporting filters/layers in generated views need to be implemented from scratch in every visualisation transformation.

3 PICTO

To combine the efficiency of bespoke graphical viewers with the strengths of M2T view generation and browser-based view rendering discussed in the previous section, in this work we propose an approach for on-demand generation and browser-based rendering of graphical model views via lazy model-to-text transformation. We have implemented the proposed approach in the context of the Picto open-source tool[9], which also provides built-in support for commonly required features such as layers, navigating between generated views, zooming in/out of views and exporting views as images. This section discusses the architecture of Picto and presents the features listed above in detail, through a running example.

3.1 Running Example

In this example, we wish to visualise models conforming to the contrived social network metamodel of Figure 1. In particular, we will use a sample model containing 3 persons (Alice, Bob and Charlie), which is shown in the form of an object diagram in Figure 2. For such social network models we wish to produce one node-edge view for the entire social network, and one view for each member of the network that omits any persons they neither like nor dislike. A preview of the two views in Picto is provided in Figure 3 (complete social network) and Figure 4 (focused on Bob). Picto's user interface has two main components. On its left-hand side is a tree widget that displays the titles and icons of the views (Social Network, Alice, Bob and Charlie in Figures 3 and 4) that users can select from. Once a view is selected, its content is generated and rendered in an embedded web browser – through a series of transformations – on the right-hand side of Picto.

3.2 Model-to-Text Transformation

Picto reuses an existing model-to-text transformation language (Epsilon's EGL [15]) to transform an input model (or a set of input models as discussed later) into a tree of views in a rule-based way. The EGL transformation used to produce the desired diagrams from a social network model is illustrated in Listing 1. The transformation consists of two rules, *Network2Graphviz* (line 1) which applies to elements of type *SocialNetwork* (line 2) and *Person2Graphviz*

[4] http://graphstream-project.org
[5] https://www.yworks.com/products/yfiles
[6] https://d3js.org
[7] https://threejs.org
[8] https://developers.google.com/chart

[9] Picto is part of the Eclipse Epsilon project – http://eclipse.org/epsilon/doc/picto

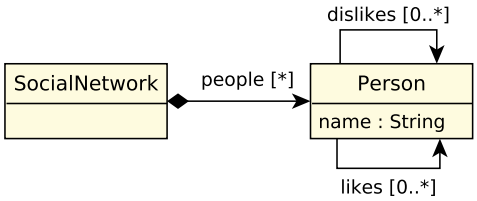

Figure 1: Social network metamodel in Ecore

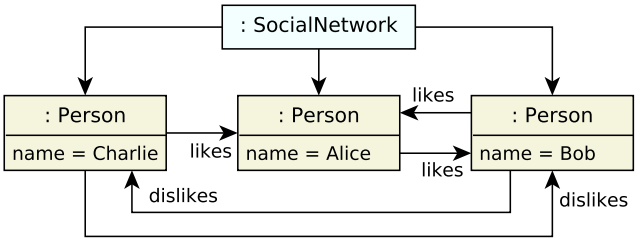

Figure 2: Sample social network model (abc.xmi) that conforms to the metamodel of Figure 1, visualised as a UML object diagram

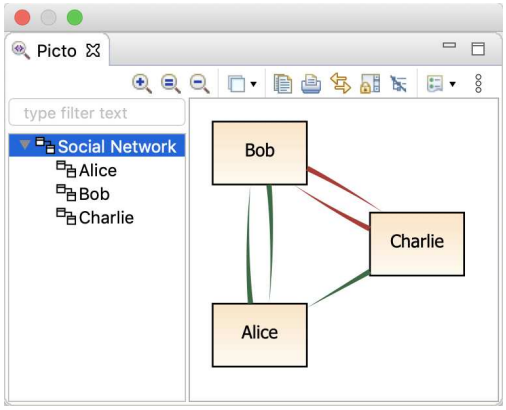

Figure 3: Social network model of Figure 2 visualised in Picto

(line 13) which applies to elements of type *Person* (line 14). When instructed to visualise the sample model of Figure 2, Picto executes the model-to-text transformation (details on how transformations are bound to models are discussed in Section 3.4), which results in 4 invocations of the two rules (one for *Network2Graphviz* and three for *Person2Graphviz*). Execution of the transformation happens in two phases, using a lazy EGL interpreter we have implemented in the context of this work.

```
1  rule Network2Graphviz
2    transform n : SocialNetwork {
3
4    template : "people2graphviz.egl"
5
6    parameters : Map {
7      "path" = Sequence{"Social Network"},
```

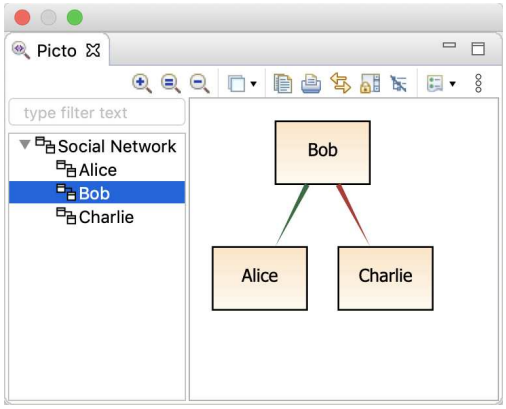

Figure 4: Bob's likes/dislikes relationships visualised in Picto

```
8      "format" = "graphviz-circo",
9      "people" = n.people
10   }
11 }
12
13 rule Person2Graphviz
14   transform p : Person {
15
16   template : "people2graphviz.egl"
17
18   parameters : Map {
19     "path" = Sequence{"Social Network", p.name},
20     "format" = "graphviz-dot",
21     "people" = Sequence{p}
22   }
23 }
```

Listing 1: Model-to-text transformation for visualising social network models

3.2.1 Phase 1: View Tree Construction. In the first phase, the *parameters* part of each rule is executed (lines 6-10 and 18-22 of Listing 1), which returns a *Map* containing (minimally) the following key-value pairs:

- *path*: specifies the path of the produced view in the left-hand-side tree of Picto. Paths for individual views are expected to be sequences (ordered collections) of strings which Picto then assembles and displays into a tree.
- *format*: specifies the format of the generated text. In this example, both rules produce text that conforms to the grammar of the Graphviz visualisation tool. Additionally, line 8 specifies that the *circo* layout algorithm should be used to render the produced Graphviz graph for the entire social network, while line 20 specifies that the *dot* layout algorithm should be used for individual person diagrams. Other built-in supported formats in Picto include *plantuml*, *svg*, *markdown*[10] and *html*. Picto also provides an Eclipse extension point that can be used to register processors for additional textual formats (e.g. Mermaid[11]).

[10] https://daringfireball.net/projects/markdown/syntax
[11] https://mermaid-js.github.io/mermaid/

Apart from the mandatory parameters described above, there is a third parameter, *people*, which is specific to this particular transformation. It appears in lines 9 and 21 to provide the people that will be fed into the *people2graphviz.egl* template defined in lines 4 and 16.

In this first phase, Picto does not perform the execution of the *people2graphviz.egl* template against the elements of the *people* parameter. This is done lazily – for efficiency purposes – in the second phase.

3.2.2 Phase 2: View Content Computation. At the end of the first phase, Picto has computed all the information needed to populate its left-hand side tree widget, but none of the actual (Graphviz in this example) contents of each view. This happens lazily when a user selects a view on the tree. When this happens, Picto will parse and execute the template specified by the rule that produced the view, and generate the content of the view (Graphviz graphs in this example). It will then transform the produced content through a chain of built-in transformations[12] (Graphviz → SVG → HTML in this case) and display the final result in its embedded browser, as shown in Figures 3 and 4.

The *people2graphviz.egl* template used in both rules of the running example is displayed in Listing 2. In the listing, text with bold font defines executable EGL statements that produce dynamic content from the model, while text with regular font is static (it is included unmodified into the content of the produced view). In particular, lines 1-6 produce Graphviz statements configuring the appearance of the graph (shape, color and font of nodes, style of edges); line 8 loops over all *people* provided to the template via the respective parameters in lines 9 and 21 of Listing 1; and, for each person, line 10 produces a node for the person while lines 12-14 and 16-18 produce green and red edges between each person and the persons they like/dislike. Listing 3 shows the produced Graphviz code behind the diagram in Figure 4.

```
1  digraph G {
2    node[shape=rectangle, fontname=Tahoma,
3         fontsize=10, style="filled",
4         gradientangle="270",
5         fillcolor="bisque:floralwhite"]
6    edge[penwidth=3, style=tapered, arrowhead=none]
7
8  [%for (p in people){%]
9
10   [%=p.name%]
11
12   [%for (o in p.likes){%]
13     [%=p.name%] -> [%=o.name%] [color=green]
14   [%}%]
15
16   [%for (o in p.dislikes){%]
17     [%=p.name%] -> [%=o.name%] [color=red]
18   [%}%]
19
20 [%}%]
21 }
```

Listing 2: people2graphviz.egl: EGL template for visualising networks of people

[12]Picto provides an Eclipse extension point for contributing such reusable transformations

```
1  digraph G {
2    node[shape=rectangle, fontname=Tahoma,
3         fontsize=10, style="filled",
4         gradientangle="270",
5         fillcolor="bisque:floralwhite"]
6    edge[penwidth=3, style=tapered, arrowhead=none]
7
8    Bob
9    Bob -> Alice [color=green]
10   Bob -> Charlie [color=red]
11 }
```

Listing 3: Graphviz code behind the diagram in Figure 4

3.3 Layers

A common way for tools to facilitate managing the complexity of visual artefacts is to provide support for layers, which can be turned on/off on demand to show or hide subsets of information of interest. Picto supports layers through a *layers* (optional) parameter in the *parameters* part of view-generating EGL rules. In the context of our running example, we wish to add two layers to views generated through the *Person2Graphiz* rule of our transformation, to allow the user to show and hide likes and dislikes relationships. To achieve this, in Listing 4 we extend the *parameters* map (originally in lines 18-22 of Listing 2) with lines 4-7, which define two layers, *likes* and *dislikes*. Each layer also specifies a human-readable title to be presented to end users. Layers can also be marked as active/inactive by default using an *active* boolean parameter (true by default).

```
1  parameters : Map {
2    "path" = Sequence{"Social Network", p.name},
3    "format" = "graphviz-dot",
4    "layers" = Sequence {
5      Map {"id"="likes", "title"="Likes"},
6      Map {"id"="dislikes", "title"="Dislikes"}
7    },
8    "people" = Sequence{p}
9  }
```

Listing 4: Network2Graphviz rule of Listing 1: extended parameters including *likes* and *dislikes* layers definition

We also need to extend the view-generating template to honour the user's layer selection. This is achieved by adding two conditional statements to the EGL template[13] that use the built-in *isLayerActive()* function to check whether the *likes* and *dislikes* layers are active (lines 1 and 7 of Listing 5) before emitting respective edges in the produced graph.

```
1  [%if (isLayerActive("likes")){%]
2  [%for (l in p.likes){%]
3    [%=p.name%] -> [%=l.name%] [color=green]
4  [%}%]
5  [%}%]
6
7  [%if (isLayerActive("dislikes")){%]
8  [%for (l in p.dislikes){%]
9    [%=p.name%] -> [%=l.name%] [color=red]
10 [%}%]
11 [%}%]
```

Listing 5: Modified transformation of Listing 2 to only render the content of active layers

[13]Originally in lines 12-18 of Listing 2

The result is demonstrated in Figure 5, where the user has turned off the *dislikes* layer, effectively hiding edges to people that Bob dislikes.

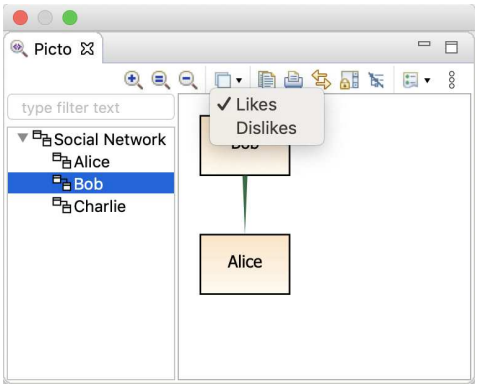

Figure 5: View of Listing 4 with the *dislikes* layer turned off

3.4 Binding Visualisations To Models

To enable binding visualisation transformations to specific models, Picto provides a small EMF-based domain-specific language, the abstract syntax of which is illustrated in Figure 6 and uses the Flexmi [11] fuzzy XML-based syntax for its instantiation. When a Flexmi model conforming to the Picto metamodel is opened, activated or saved in Eclipse, Picto is triggered and it executes the transformation specified in the *transformation* property of its root *picto* element against the models it contains. For example, in line 2, the Picto model in Listing 6 binds the *socialnetwork.egx* visualisation transformation illustrated in Listing 1 to the *abc.xmi* model that contains Alice, Bob and Charlie.

As shown in Figure 6, a Picto visualisation can refer to multiple models. As they are written in EGL, Picto visualisation transformations can access multiple models of different technologies (e.g. EMF, Simulink [16]).

```
1  <?nsuri picto?>
2  <picto transformation="socialnetwork.egx">
3    <model type="EMF">
4      <parameter name="metamodelUri"
5                 value="socialnetwork"/>
6      <parameter name="modelFile"
7                 file="abc.xmi"/>
8    </model>
9  </picto>
```

Listing 6: Binding socialnetwork.egx to abc.xmi in abc.picto

3.5 Custom Views

The model-to-text transformation visualisation in our example produces two types of views. One for the social network as a whole, and one for each member of the network. What it does not facilitate is the specification of views that involve a custom selection of members of the network (e.g. only Alice and Bob). To allow such user-defined views, Picto provides the concept of "custom view",

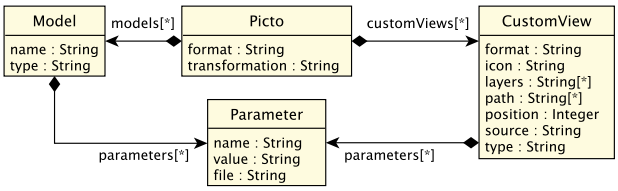

Figure 6: The Picto Metamodel

through the respective class in its metamodel. There are two types of custom views.

3.5.1 Dynamic Views. These are views that have their *type* attribute set to the name of one of the rules in the view transformation. Their content is produced by executing the said rule against the parameters specified in the view through the *CustomView→Parameter* reference in Figure 6. To support custom views in our running example, we need to extend the view transformation with another rule, *Persons2Graphviz*, which is displayed in Listing 7. This rule is similar to the other two rules, with three notable differences:

- It is annotated as *@custom* (line 1) so that Picto will only run it if it is referenced by a custom view
- It does not define a *path* in its parameters (as rules *Network2Graphviz* and *Person2Graphviz* do in lines 7 and 19 of Listing 1) as the path of custom views on the tree is expected to be provided through the Picto visualisation model
- It expects an additional *names* variable (lines 12 and 14) from which it will compute the people to display

```
1  @custom
2  rule Persons2Graphviz {
3
4    template : "socialnetwork2graphviz.egl"
5
6    parameters : Map{
7      "format" = "graphviz-dot",
8      "layers" = Sequence {
9        Map {"id"="likes", "title"="Likes"},
10       Map {"id"="dislikes", "title"="Dislikes"}
11     },
12     "people" = names.isDefined() ?
13        Person.all.select(p|
14         names.includes(p.name)):
15         Sequence{}
16   }
17 }
```

Listing 7: Persons2Graphviz rule for producing views of user-selected groups of persons

To instantiate this view, in lines 3-6 of Listing 8 we add a new custom view element to *abc.picto*. As seen in Figure 7, the path of the new view in the Picto tree is *Custom→Alice and Bob*, and it displays Alice, Bob and their like/dislike relationships (compared to the full network diagram in Figure 3, it omits Charlie's likes/dislikes relationships).

```
1  <?nsuri picto?>
2  <picto transformation="socialnetwork.egx">
3    <view path="Custom, Alice and Bob"
4        type="Persons2Graphviz">
```

```
5       <parameter name="names" values="Alice, Bob"/>
6     </view>
7     <view path="Custom, Readme"
8         source="readme.html"/>
9     ...
10  </picto>
```

Listing 8: A dynamic and a static custom view

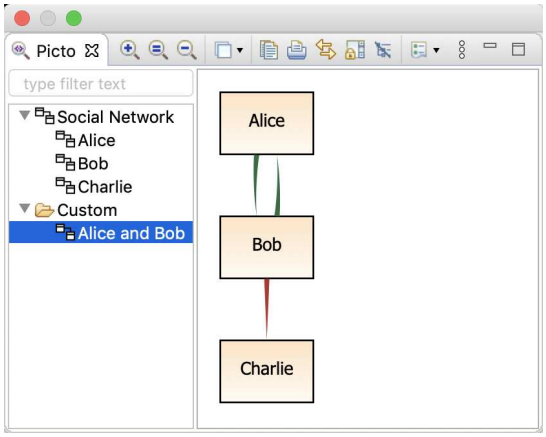

Figure 7: The custom *Alice and Bob* view

3.5.2 Static Views. In some cases, it is useful to produce one-off views that also contain information not captured in one of the models. To accommodate such use-cases Picto also supports *static* custom views. In contrast to dynamic views which have a *type* attribute that references a transformation rule (discussed above), static views have a *source* attribute that points to a static HTML or Markdown file with the content of the view. A static view is defined in lines 7-8 of Listing 8. The content of the referenced *readme.html* file is shown in Listing 9. Of particular interest is line 6 of the latter, which demonstrates the *picto-view* tag which can be used to embed copies of other (static or dynamically-computed) views in a static view. The resulting visualisation appears in Figure 8.

```
1  <html>
2    <body>
3      <h1>Overview</h1>
4      <p>Nodes represent people, and green/red
5      edges show who likes/dislikes who.</p>
6      <picto-view path="Social Network"/>
7    </body>
8  </html>
```

Listing 9: The readme.html file referenced in line 8 of Listing 8

3.6 Additional Examples

In this section we briefly discuss additional use-cases of Picto, to demonstrate its applicability beyond node-edge diagrams. All presented examples are available in Picto's source code repository[14].

[14]https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/tree/examples

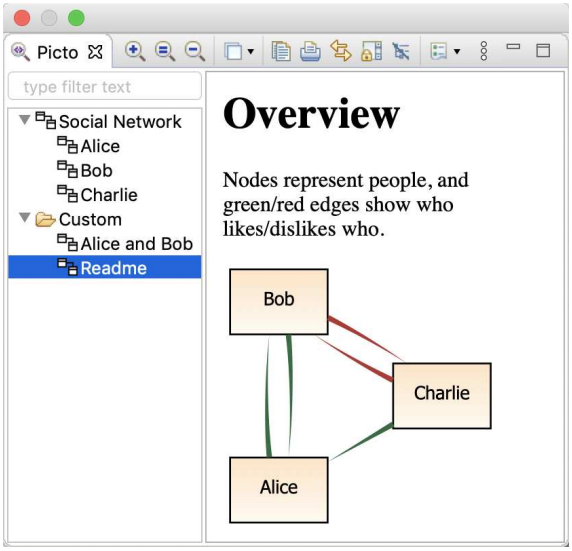

Figure 8: Static (HTML) view embedding a copy of the Social Network view of Figure 3

Figure 9 showcases the resulting view of a M2T transformation composed of 550 lines of code (LOC) of EGL that can produce Graphviz class diagrams from Ecore metamodels (a subset of Ecore.ecore is visualised in this case). The view in the figure is a custom diagram for a hand-picked set of core Ecore classes. Figure 10 demonstrates a tabular visualisation (88 LOC) of a model capturing risks to demonstrate that Picto can also produce table/form-based views trough transformations that produce HTML content. Finally, Figure 11 shows an application of Picto for contextual visualisation of sequence diagrams (87 LOC), using PlantUML[15]. In this use case, from a single interaction scenario, Picto is used to produce a number of sequence diagrams, one for each alternative path of the scenario. Finally, Figure 12 shows an application (42 LOC) that can generate interactive 3D inheritance graphs from Ecore metamodels using ThreeJS and WebGL[16].

4 EVALUATION

To measure the benefits of the lazy view generation strategy implemented by Picto, we have carried out performance evaluation experiments where we compared view generation times of Picto to those of a batch M2T transformation that produces identical views[17]. The following sections describe the experiments and discuss the obtained results.

4.1 Comparison Method

We start by describing the visualisation scenarios, the compared approaches, and the measuring platforms and methods used during the comparison.

[15]https://plantuml.com/sequence-diagram
[16]https://github.com/vasturiano/3d-force-graph
[17]Instructions to reproduce this evaluation are provided in the following external repository: https://github.com/kolovos/models2020-picto-data

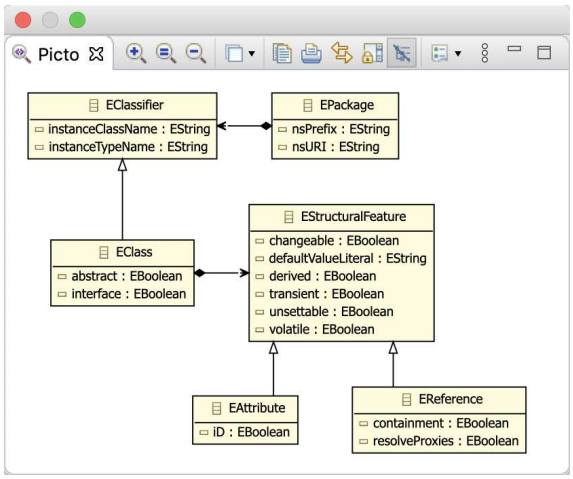

Figure 9: Picto used to visualise Ecore metamodels with Graphviz

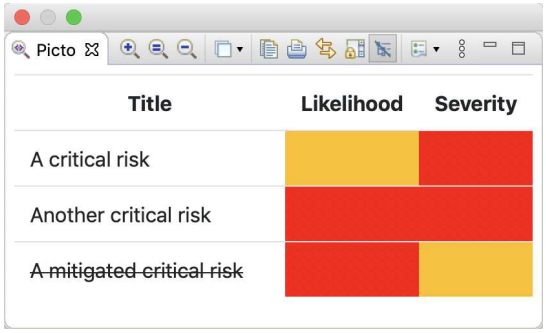

Figure 10: Picto used to visualise risks in a tabular form with HTML

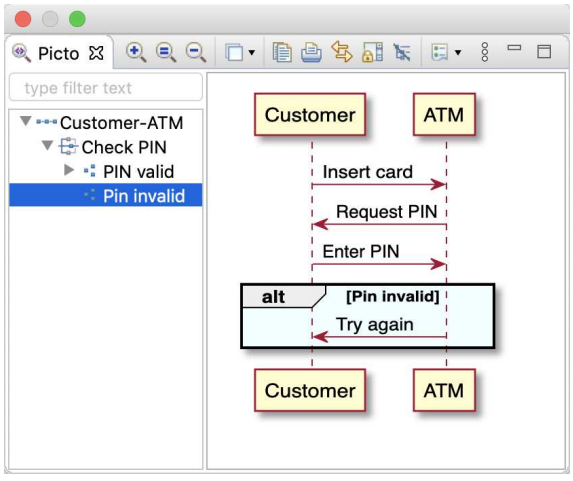

Figure 11: Picto used for contextual visualisation of sequence diagrams with PlantUML

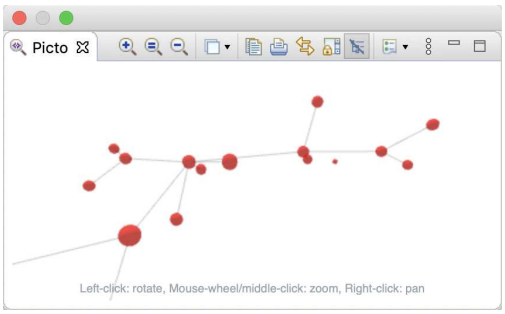

Figure 12: Picto used for 3D representation of inheritance hierarchy in Ecore metamodels using ThreeJS and WebGL

Table 1: Ecore metamodels used in the comparison.

Name	Size (MiB)	# EClasses
UML.ecore	1.3	243
CIM.ecore	2.6	600
eMoflonTTC17.ecore	3.3	1090
RevEngSirius.ecore	4.7	5208

4.1.1 Visualisation scenarios. Two scenarios were used during this evaluation. The first one involved Ecore metamodels and generating views such as the one depicted in Figure 9. More specifically, one view was generated for each EClass in the input metamodel (considered the *main* one of the view). The view contains the EClass itself, as well as the EClasses it refers to through EReferences, and its supertypes. We used the BigQuery Github dataset[18] to search for very large publicly available metamodels, from which we included the following four in the comparison:

- the UML2 metamodel;
- the Common Information Model (CIM)[19], which is a standard for the definition of electrical networks;
- a metamodel used internally by the *eMoflon* solution of the 2017 Transformation Tool Contest (TTC17)[20];
- a reverse-engineered metamodel of the Sirius codebase[21] that has been used by the developers of the EcoreTools diagramming tool to carry out performance tests.

The details of the selected metamodels are shown in Table 1. For instance, *RevEngSirius.ecore* is the largest of these metamodels, with a size of ~4.7 MiB, and around 5.2K EClasses.

The second scenario involves generating views from synthetic models conforming to a contrived (Simulink-like) component/connector metamodel. In this metamodel, each component has input and output ports, and can contain other nested components, which are interconnected between them and with the available ports to represent a modular system. For this visualisation, a view, like the

[18] https://cloud.google.com/blog/products/gcp/github-on-bigquery-analyze-all-the-open-source-code
[19] https://www.dmtf.org/standards/cim
[20] https://www.transformation-tool-contest.eu/2017/solutions_smartGrid.html
[21] https://www.eclipse.org/sirius/

one in Figure 13, is generated for each component that contains at least one nested sub-component.

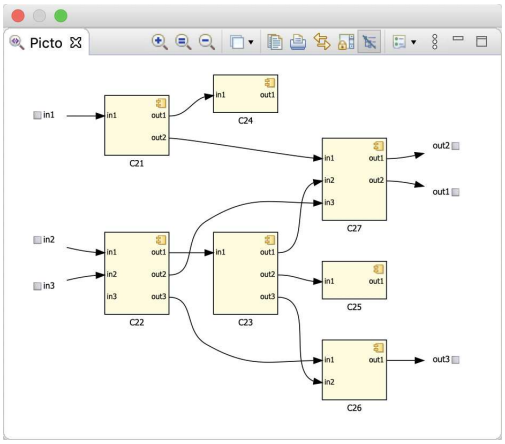

Figure 13: Example component/connector view in Picto

4.1.2 Compared approaches. We measured the time it took to generate the views for the scenarios described above both using Picto and with standalone batch M2T transformations. For the batch transformations, we used the same language as in Picto, this is, EGL (see Section 3.2). This ensures that what we are measuring is the impact of the lazy generation strategy we devised for this work, as opposed to more fundamental differences in the performance of two M2T transformation languages. Also, using EGL facilitated creating identical M2T transformations as those in Picto, with only minimal changes to make them work in batch/standalone mode.

In the two visualisation scenarios, the M2T transformations generate DOT graphs that are then translated to SVG/HTML for in-browser rendering through the Graphviz program. While Picto has facilities to do that transparently for the user (see Section 3.2.2), we need to provide the same in the batch M2T approaches. Therefore, after the M2T batch transformation concludes, a post-processing step is carried out to, starting from DOT, generate the SVG and HTML files that would be rendered in a browser. The time to perform this post-processing step is included in the results of the batch transformations.

One of the advantages of using a batch transformation instead of Picto is the possibility of parallelising the generation of views in different system cores/threads. Therefore, we created two variants of the batch transformation approach: the first one uses sequential (single-threaded) execution, while the second one employs multi-threaded computation via a parallel EGL execution engine[22] for the M2T transformation and the Java 8 Streams API for the post-processing phase.

Summarising, three approaches were compared: Picto, a single-threaded and a multi-threaded batch M2T transformation.

4.1.3 Measuring platforms. The experiments were carried out on a desktop computer running Ubuntu on a 6-core, 12-thread AMD Ryzen 1600 CPU with 32GiB of ram and a PCIe NVM SSD. As this powerful hardware might not be typical of a developer workstation yet, we also ran the transformations in a lower-spec laptop featuring the same Ubuntu system and a 2-core, 4-thread Intel Core i5 7200U CPU, 16GiB of RAM, and again a PCIe NVM SSD.

4.1.4 Measuring method. For the batch M2T transformations, we measured the time it took to run the transformations against the target models. On the other hand, Picto's lazy computation strategy required some instrumentation for performing the measurements. We included relevant code in a fork of Picto's implementation that forces the generation of each individual view just as if a user has selected it from the user interface, and gathers these measurements in a results file.

For both types of approaches, generation times were measured 10 times, and then the results were averaged. To ensure that average figures were not disproportionately affected by outliers, we also calculated the standard deviation of these times. The coefficient of variation, this is, the ratio of the standard deviation to the mean, was not higher than 0.005 for the single-thread batch transformation, 0.127 for the multi-thread one, and 0.164 for the individual Picto views, which indicates a low spread in the obtained results. The higher dispersion of the Picto times can be due to their measurement inside an Eclipse instance, as opposed to the batch transformations' execution that happened through a standard Java process. Also, to prevent any inconsistencies due to low CPU states during the initial measurements, we warmed up the measuring platforms by executing initial generations whose obtained times were discarded.

4.2 Results

4.2.1 Ecore metamodel visualisation. Figure 14 shows the measured generation times of the three approaches for the four selected Ecore metamodels. Our experiment simulates a scenario in which a user is accessing the generated views one by one, i.e., selecting the generated view for each EClass in the input metamodel, until all produced views have been accessed. The y-axis represents the accumulated generation time of the accessed views, while the x-axis indicates the number of views that have been accessed up to that point.

The number of accessed views is irrelevant for the batch transformation approaches, as all views are generated upfront. Because of that, batch approaches are represented by horizontal lines indicating the time they took to generate all views of each model, with the single-thread variant in dotted red, and the multi-thread one in dashed green. As expected, the multi-threaded variant took less time to complete, providing savings of 82.7 to 84.7% compared to the single-threaded execution.

On the other hand, the number of accessed views is very relevant for Picto, whose execution time is represented with a solid blue line. As the number of accessed views increases, so does Picto's accumulated execution time (since views are generated and rendered lazily). For Picto, the y-axis value at the "0 accessed views" point depicts the time it took to complete the upfront view tree computation phase (see Section 3.2.1). This time is almost negligible, as it only amounts to 22.3, 39.2, 62 and 302 milliseconds for the

[22] https://www.eclipse.org/epsilon/doc/articles/parallel-execution

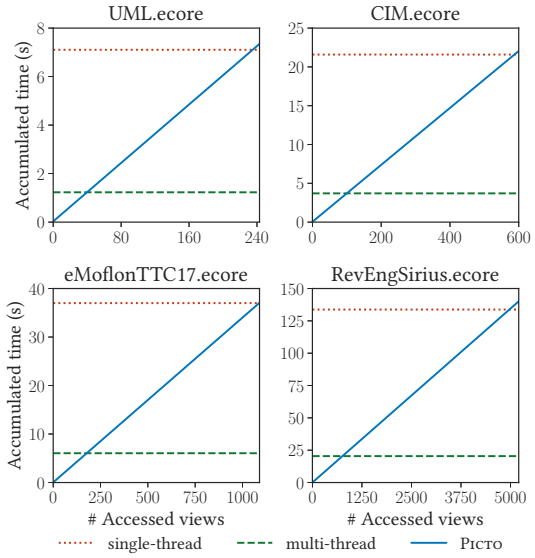

Figure 14: Time results for the Ecore metamodel visualisation scenario on the 6-core/12-thread CPU of the desktop computer. The x-axis represents the number of views accessed up to a certain point, while the y-axis indicates the accumulated time it took to generate those views.

UML, CIM, eMoflonTTC17 and RevEngSirius metamodels, respectively[23]. Lastly, to improve presentation, the time it took Picto to generate each individual view has been averaged. Showing the real time would have made relevant the order in which the views are accessed, i.e., if those views that took more time to get computed are accessed earlier, then the Picto accumulated time would increase quicker at first, and vice versa. In any event, the generation times were fairly uniform across all EClass views of the metamodels, so this averaging only has a minor aesthetic impact.

Of particular interest in the graphs of Figure 14 are the crossing points at which the Picto time meets with the batch transformation times. When that crossing happens, it means that the accumulated time it took Picto to generate the accessed views at that point has reached the time that took the crossed batch transformation to generate all views of the model. So, the greater the number of accessed views required to reach those crossing points, the more substantial benefit the lazy generation of views (i.e. Picto) is providing. In contrast, if the number of accessed views increases past the crossing point with certain batch transformation, then the final generation time of Picto would be greater.

The first crossing point involves Picto and the multi-thread batch transformation times. This crossing happens at 40, 100, 177, and 758 accessed views (14 to 16% of the total number of views). These numbers show that, when considering the generation of all views, parallelising this generation contributes to a great reduction of the computation times.

[23]The same model order is used when enumerating values below.

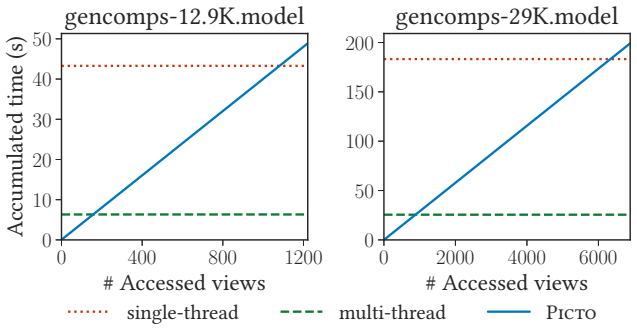

Figure 15: Time results for the component model visualisation scenario on the 6-core/12-thread CPU of the desktop computer. The x-axis represents the number of views accessed up to a certain point, while the y-axis indicates the accumulated time it took to generate those views.

The second crossing takes place when Picto's accumulated time reaches the execution time of the single-thread batch transformation. This crossing happens in all experiments when almost all the views have been generated. Precisely, it takes place when 235, 588, 1088, and 4972 views have been generated. The extra time that Picto requires to generate the remaining views (i.e. the ones that have not been accessed yet) is an overhead of its lazy M2T functionality, which is avoided when generating all views at once in the batch transformation. However, the measured overhead is very small, oscillating between 2 and 4% of the total generation time for Picto when compared with single-threaded batch figures.

4.2.2 Component model visualisation. With respect to the (synthetic) component model visualisation scenario, Figure 15 includes the results for the two biggest models we generated. The first model, *gencomps-12.9K*, is 9.3 MiB in size, contains around 12.9K components (hence the name), and its visualisation included generating a total of 1221 views. As for the bigger *gencomps-29K* model, its numbers go up to 23.8 MiB in size, 29K component elements, and 6888 views. With respect to crossing points, Picto and the multi-thread batch execution crossed after accessing 158 and 883 views (~12% of the total number of views for both cases) for the gencomps-12.9K and gencomps-29K models, while the crossing with the single-thread execution happens at 1081 and 6337 views (88% and 91% of the views, respectively).

If we compare the results of both visualisation scenarios, we can see that the obtained times for the component models are consistent with those shown for Ecore metamodels. There is an increase in the total generation time in the case of the components scenario that we attribute to the larger size of these models, which translated into bigger view computing times. For instance, the eMoflonTTC17 metamodel and the gencomps-12.9K model visualisations contain a similar number of views, with 1090 and 1221 views, respectively. Nevertheless, the size of these models is 3.3 and 9.3 MiB which, summed to the difference of 121 total views between the visualisations, causes a noticeable difference in the single-thread and Picto total times (36.9 and 37.0 seconds for eMoflonTTC17 and 43.2 and 48.8 for gencomps-12.9K). For the multi-threaded batch execution,

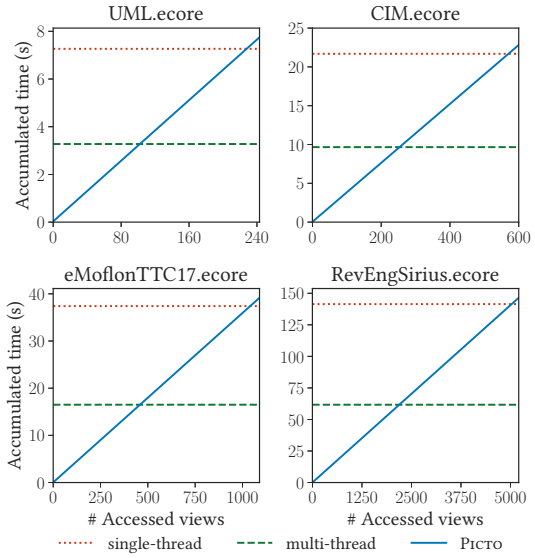

Figure 16: Time results for the Ecore metamodel visualisation scenario on the 2-core/4-thread CPU of the laptop. The x-axis represents the number of views accessed up to a certain point, while the y-axis indicates the accumulated time it took to generate those views.

though, the times remained fairly similar for both models (6.0 and 6.3 seconds, respectively). These times suggest that the parallel execution of Ecore metamodels was not able to deplete all the computing resources offered by the 6-core/12-thread computer CPU, so there were some resources available to cope with the generation of 121 extra component views (that require more computation time) in a very close total time. A similar comparison can be carried out between the RevEngSirius metamodel and the gencomps-29K component model.

4.2.3 Laptop Platform results. Lastly, Figure 16 shows the obtained times for the Ecore experiments when the generations were executed on a laptop. In that case, the obtained times were very similar for the Picto and single-thread executions to those of the desktop machine. On the other hand, the benefits of the multi-threaded version were not as significant, because the parallel execution in a 2-core/4-thread CPU could not provide the same performance as a more capable 6-core/12-thread CPU. In that case, and for the Ecore experiments depicted in Figure 16, the crossing between the multi-thread execution and Picto happened at 102, 253, 458 and 2192 accessed views (~42% of the total views for all models), which indicates that, for lower-spec platforms, the use of Picto is even more beneficial.

4.3 Discussion

The obtained results indicate that Picto's lazy M2T view generation approach can have a very low upfront cost, it scales up linearly with the number of views accessed, and it only has a very small cumulative overhead compared to single-threaded batch execution.

Its low upfront execution time and linear scalability makes Picto's lazy view generation approach particularly efficient for visualising large evolving models, where a modest number of views are accessed between edits. For scenarios where models and visualisation transformations are immutable, and a substantial number of views are expected to be accessed by users, batch (and particularly multi-threaded) transformation is more efficient.

As the discussion in this section has focused solely on performance, it is worth noting that even in such cases involving immutable models and transformations, a reason for considering Picto could be its support for features such as layers, custom and composite views, that would need to be reimplemented from scratch in a batch M2T approach.

4.4 Threats to Validity

The main threat to the validity of the obtained results is potential bias in the selection of the two visualisation transformations we used for experimental evaluation. As Picto is a new tool, at the time of writing this paper, there are no externally-developed visualisation transformations that we could reuse. To mitigate this threat, we chose to develop and use transformations that produce views that closely follow established graphical notations (class and component diagrams).

Regarding the low upfront execution time of Picto demonstrated in both experiments, it should be stressed that this is not a property guaranteed by Picto, but a property of the individual transformations instead. Care has been taken for both transformations to do as little work as possible during their upfront view tree computation phase (see Section 3.2.1) and to defer all other computation to the lazy view content generation phase (see Section 3.2.2). Transformations that need to do a substantial amount of work during the former phase, can lead to higher upfront execution times, negating some of the efficiency benefits of Picto.

5 RELATED WORK

Sprotty [6], is a state-of-the-art visualisation framework that allows the rendering of graphical views using web-based technologies. A Sprotty application is formed of an S-Model (Sprotty model) to represent the current diagram, a client component (responsible for rendering an S-Model in a browser) and an optional server component (responsible for mapping the semantic model – which can be in any format, such as XMI or database for example – into an S-Model). Layouting can be performed either on the client or server by using frameworks such as ELK[24] and ELKJS[25] respectively.

As Sprotty runs in a web-browser and server, it can be used in a variety of scenarios, including being in a standalone web-browser application, browser-based IDEs such as Eclipse Theia or embedded into the Eclipse IDE. Sprotty has been shown to have good compatibility with the Language Sever Protocol (LSP) and can work well with visualising Xtext DSLs[26].

Sprotty has been shown to provide features such as bi-directional navigation between the textual models and diagrams, and filtering.

[24] https://www.eclipse.org/elk/
[25] https://github.com/kieler/elkjs
[26] https://github.com/TypeFox/theia-xtext-sprotty-example

Depending on how the mapping from the semantic model to the S-Model is defined, model transformations can be implemented using batch transformations or on-the-fly. Sprotty is very customisable and extensible as it uses dependency injection allowing additional components to be added or the default components to be replaced.

Sprotty does not support modification of the semantic model, however projects such as the Eclipse Graphical Language Server Protocol (GLSP) [5] built atop of Sprotty and can allow diagrams to be edited via a web-browser based application.

Also related to Picto is the KIELER Lightweight Diagram framework (KLighD) [17]. This framework allows for on-demand model visualisation of models by using EMF, Xtend and Piccolo2D (2D graphics framework). KLighD provides three EMF based models for describing the diagram: a KGraph, KLayoutData and KRendering model. KLighD supports layouting provided by KIELER Infrastructure for Meta Layout (KIML), a predecessor to ELK. KLighD can provide filtering in the form of "Hierarchy Levels" and limited fine tuning of diagrams (by allowing a user to adjust whitespace, direction components are facing). To map a semantic model to the KGraph and KRendering models, KLighD provides an extension point and Java/Xtend interface which a user must implement allowing a user to define model-to-model transformations written in Xtend to transform the semantic model into respective KGraph and KRendering models. An example of a tool created with KLighD is EcoreViz where Ecore metamodels can be created dynamically[27].

Both Sprotty and KLighD only support node-edge diagrams and do not provide built-in support for generating hierarchies (trees) of views from a single model, or from multiple models conforming to different modelling technologies. In contrast, Picto supports Graphviz and Plantuml diagrams, form/table views using HTML, as well as views based on arbitrary Javascript libraries such as Three.js (see Figure 12). It also provides support for hierarchical visualisation of hybrid models by leveraging the respective facilities of the Epsilon platform. In principle, Picto and Sprotty can be used complementarily, with Sprotty acting as one of the rendering technologies supported by Picto.

6 CONCLUSIONS AND FUTURE WORK

In this paper we have conducted a critical review of common approaches for producing graphical views from models, and then introduced a novel approach for producing transient graphical views using lazy model-to-text transformation. We have also presented the open-source Picto tool, which implements the proposed approach, and evaluated the efficiency benefits it delivers compared to batch model-to-text transformation.

The proposed method has been shown to have a low upfront cost, to scale up linearly and to deliver substantial efficiency benefits when a modest number of views is accessed by users between changes in the underlying models – which is often the case in practice. In terms of visualisation capabilities, being based on M2T transformation and browser-based rendering, Picto can reuse any JavaScript-based visualisation library and can also be extended through dedicated Eclipse extension points with support for additional 3rd party tools beyond Graphviz and PlantUML. Having said that, it is worth reiterating that Picto is not a replacement for graphical model editing frameworks such as Sirius but instead targets use-cases where read-only views are desirable/sufficient.

Future work on Picto includes view-based model differencing, and developing a bespoke rule-based language for view generation which will provide first-class support for core Picto concepts (e.g. view format, layers, path) as opposed to piggy-backing on EGL's *parameters* block (see lines 6-16 of Listing 7).

Acknowledgements The work in this paper has been partially funded through the HICLASS InnovateUK project (contract no. 113213), an InnovateUK co-funded Knowledge Transfer Partnership between the University of York and Rolls-Royce plc (contract no. KTP011043), and the TYPHON EC H2020 project (contract no. 780251).

[27] https://github.com/kieler/ecoreviz

REFERENCES

[1] El Arbi Aboussoror, Ileana Ober, and Iulian Ober. 2012. Seeing Errors: Model Driven Simulation Trace Visualization. In *Model Driven Engineering Languages and Systems*, Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 480–496.

[2] Bastien Amar, Hervé Leblanc, Bernard Coulette, and Clémentine Nebut. 2010. Using Aspect-Oriented Programming to Trace Imperative Transformations. In *Proceedings of the 14th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2010, Vitória, Brazil, 25-29 October 2010*. IEEE Computer Society, 143–152. https://doi.org/10.1109/EDOC.2010.12

[3] Hugo Brunelière, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. 2014. MoDisco: A model driven reverse engineering framework. *Inf. Softw. Technol.* 56, 8 (2014), 1012–1032. https://doi.org/10.1016/j.infsof.2014.04.007

[4] R. Ian Bull, Casey Best, and Margaret-Anne D. Storey. 2004. Advanced widgets for Eclipse. In *Proceedings of the 2004 OOPSLA workshop on Eclipse Technology eXchange, ETX 2004, Vancouver, British Columbia, Canada, October 24, 2004*, Michael G. Burke (Ed.). ACM, 6–11. https://doi.org/10.1145/1066129.1066131

[5] Eclipse Foundation. [n.d.]. *Eclipse Graphical Language Server Protocol (GLSP)*. https://www.eclipse.org/glsp/

[6] Eclipse Foundation. [n.d.]. *Eclipse Sprotty*. https://projects.eclipse.org/projects/ecd.sprotty

[7] E.R. Gansner, E. Koutsofios, S.C. North, and K.-P. Vo. 1993. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering* 19, 3 (March 1993), 214–230. https://doi.org/10.1109/32.221135

[8] Carlos A. González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. 2012. EMFtoCSP: a tool for the lightweight verification of EMF models. In *Proceedings of the First International Workshop on Formal Methods in Software Engineering - Rigorous and Agile Approaches, FormSERA 2012, Zurich, Switzerland, June 2, 2012*, Stefania Gnesi, Stefan Gruner, Nico Plat, and Bernhard Rumpe (Eds.). IEEE, 44–50. https://doi.org/10.1109/FormSERA.2012.6229788

[9] Steven Kelly, Kalle Lyytinen, and Matti Rossi. 2013. MetaEdit+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In *Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE*, Janis A. Bubenko Jr., John Krogstie, Oscar Pastor, Barbara Pernici, Colette Rolland, and Arne Sølvberg (Eds.). Springer, 109–129. https://doi.org/10.1007/978-3-642-36926-1_9

[10] Steven Kelly, Kalle Lyytinen, Matti Rossi, and Juha-Pekka Tolvanen. 2013. MetaEdit+ at the Age of 20. In *Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE*, Janis A. Bubenko Jr., John Krogstie, Oscar Pastor, Barbara Pernici, Colette Rolland, and Arne Sølvberg (Eds.). Springer, 131–137. https://doi.org/10.1007/978-3-642-36926-1_10

[11] Dimitrios S. Kolovos, Nicholas Matragkas, and Antonio García-Domínguez. 2016. Towards Flexible Parsing of Structured Textual Model Representations. In *Proceedings of the 2nd Workshop on Flexible Model Driven Engineering co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2016), Saint-Malo, France, October 2, 2016 (CEUR Workshop Proceedings, Vol. 1694)*. CEUR-WS.org, 22–31. http://ceur-ws.org/Vol-1694/FlexMDE2016_paper_3.pdf

[12] Rainer Koschke. 2002. Software Visualization for Reverse Engineering. In *Software Visualization (International Seminar Dagstuhl Castle, Revised Papers)*, Stephan Diehl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 138–150.

[13] Daniel Moody. 2009. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Trans. Softw. Eng.* 35, 6 (Nov. 2009), 756âĂŞ779. https://doi.org/10.1109/TSE.2009.67

[14] PlantUML Team. [n.d.]. *PlantUML*. https://plantuml.com

[15] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona Polack. 2008. The Epsilon Generation Language. In *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany,*

June 9-13, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5095), Ina Schieferdecker and Alan Hartman (Eds.). Springer, 1–16. https://doi.org/10.1007/978-3-540-69100-6_1

[16] Beatriz Sánchez, Athanasios Zolotas, Horacio Hoyos Rodriguez, Dimitris S. Kolovos, and Richard F. Paige. 2019. On-the-Fly Translation and Execution of OCL-Like Queries on Simulink Models. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019, Munich, Germany, September 15-20, 2019*, Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño (Eds.). IEEE, 205–215. https://doi.org/10.1109/MODELS.2019.000-1

[17] C. Schneider, M. Spönemann, and R. von Hanxleden. 2013. Just model! – Putting automatic synthesis of node-link-diagrams into practice. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. 75–82.

[18] Ricardo Solmi. [n.d.]. The Whole Platform. https://whole.sourceforge.io/

[19] Miro Spönemann, Christoph Daniel Schulze, Christian Motika, Christian Schneider, and Reinhard von Hanxleden. 2013. KIELER: Building on automatic layout for pragmatics-aware modeling. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing, San Jose, CA, USA, September 15-19, 2013*, Caitlin Kelleher, Margaret M. Burnett, and Stefan Sauer (Eds.). IEEE Computer Society, 195–196. https://doi.org/10.1109/VLHCC.2013.6645265

[20] Oskar van Rest, Guido Wachsmuth, Jim R. H. Steel, Jörn Guy Süß, and Eelco Visser. 2013. Robust Real-Time Synchronization between Textual and Graphical Editors. In *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7909)*, Keith Duddy and Gerti Kappel (Eds.). Springer, 92–107. https://doi.org/10.1007/978-3-642-38883-5_11

[21] Markus Voelter and Sascha Lisson. 2014. Supporting Diverse Notations in MPS' Projectional Editor. In *Proceedings of the 2nd International Workshop on The Globalization of Modeling Languages co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems, GEMOC@Models 2014, Valencia, - Spain, September 28, 2014 (CEUR Workshop Proceedings, Vol. 1236)*, Benoît Combemale, Julien DeAntoni, and Robert B. France (Eds.). CEUR-WS.org, 7–16. http://ceur-ws.org/Vol-1236/paper-03.pdf

[22] Guido Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *IEEE Software* 31, 5 (2014), 35–43. https://doi.org/10.1109/MS.2014.100