Science of Computer Programming 232 (2024) 103037

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Original software publication

Exploring complex models with picto web

Alfa Yohannis *, Dimitris Kolovos, Antonio García-Domínguez

University of York, United Kingdom

A R T I C L E I N F O

Keywords:
Complex models
Model exploration
Picto web
Visualisation

A B S T R A C T

Picto Web is a multi-tenant web-based tool for complex model exploration. It can transform different types of models into a variety of transient web-based views in formats such as HTML, Graphviz and PlantUML using rule-based model-to-text transformations. Picto Web implements a lazy view computation approach to support large models and complex transformations efficiently, and includes model and transformation template monitoring and push notification facilities to automatically recompute views when either are modified and deliver updated views to clients. The tool is packaged as a Docker container for ease of deployment.

Code metadata

Code metadata description	Please fill in this column
Current code version	0.1.1-alpha
Permanent link to code/repository used for this code version	https://github.com/ScienceofComputerProgramming/SCICO-D-22-00358
Permanent link to Reproducible Capsule	https://doi.org/10.24433/CO.1899025.v1
Legal Code License	Eclipse Public License (EPL-2.0)
Code versioning system used	git
Software code languages, tools, and services used	Java, Javascript, HTML, CSS, Graphviz
Compilation requirements, operating environments and dependencies	Java 11, Maven
If available, link to developer documentation/manual	https://github.com/epsilonlabs/picto-web/blob/main/README.md
Support email for questions	Online forum https://github.com/epsilonlabs/picto-web/discussions

1. Motivation and significance

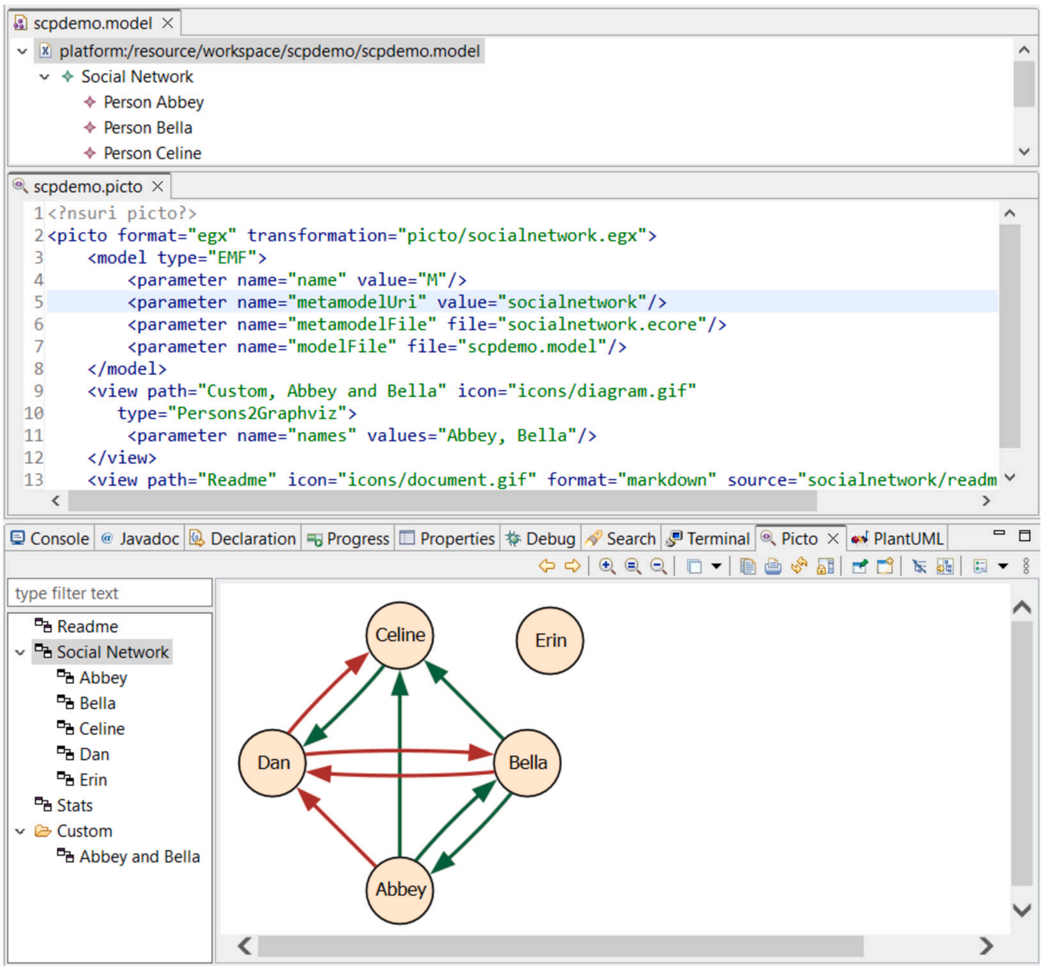
Models of complex systems can contain a large number of heterogeneous elements with non-trivial relationships between them [1, 2]. One way to help stakeholders comprehend and communicate such models is through visualisation. For large models, it is typically undesirable to show all model elements and their relationships in a single graphical representation since it can overwhelm users. In terms of computation, this approach is also inefficient for large models since a potentially very large and complex diagram (or similar visual representation) has to be rendered in one go.

The code (and data) in this article has been certified as Reproducible by Code Ocean: https://codeocean.com/. More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

* Corresponding author.
E-mail address: alfa.ryano@gmail.com (A. Yohannis).

https://doi.org/10.1016/j.scico.2023.103037
Received 19 December 2022; Received in revised form 5 October 2023; Accepted 5 October 2023

Available online 10 October 2023



```
<?nsuri picto?>
<picto format="egx" transformation="picto/socialnetwork.egx">
    <model type="EMF">
        <parameter name="name" value="M"/>
        <parameter name="metamodelUri" value="socialnetwork"/>
        <parameter name="metamodelFile" file="socialnetwork.ecore"/>
        <parameter name="modelFile" file="scpdemo.model"/>
    </model>
    <view path="Custom, Abbey and Bella" icon="icons/diagram.gif"
       type="Persons2Graphviz">
        <parameter name="names" values="Abbey, Bella"/>
    </view>
    <view path="Readme" icon="icons/document.gif" format="markdown" source="socialnetwork/readm
```

Fig. 1. Picto plugin for Eclipse environment [7].

Moreover, in a multi-user scenario, the model visualisation system also has to handle many repeated requests, whether a user is refreshing a view several times or multiple users send requests to the same view over the network. Thus, novel techniques are needed to enable developers and other stakeholders to construct and carry out contextual exploration of large and complex models from multiple viewpoints and at different granularity levels. The visualisation must also be capable of handling frequent requests from many users and delivering the desired views promptly.

To facilitate complex model exploration, the open-source diagramming framework Sprotty [3] renders graphical views using web technologies. In addition to being utilised on the client side for efficient, scalable SVG rendering with Eclipse Layout Kernel [4] layout, Sprotty also offers a headless component that can be used with the Xtext framework to generate and auto-layout node-edge diagrams from textual models. By itself, Sprotty does not support diagram editing. If diagram editing is required, additional frameworks such as the Graphical Language Server Platform (GLSP) [5] need to be used.

The KIELER Lightweight Diagram (KLD) framework [6] employs Xtend, Piccolo2D, and EMF to support on-demand model visualisation. It also uses the KIELER Infrastructure for Meta Layout (KIML) to specify diagram layouts. Users of KLD can define a diagram using one of three EMF-based models (KLayoutData, KRendering, and KGraph). Java/Xtend interfaces can also be used to extend KLD to transform and map a semantic model to KGraph and KRendering models. KLD only supports rendering views in node-edge diagrams, like Sprotty.

Another tool for visualising models is Picto [8], an Eclipse plugin that can create transient graphical views from heterogeneous models using lazy model-to-text transformation. A screenshot of Picto in Eclipse is displayed in Fig. 1. In the figure, Picto shows a social network model. The nodes represent people, and the edges indicate 'like' and 'dislike' relationships between the people (see Section 2.2 for a detailed description of the model).

Picto's user interface has two main parts to display the model: a tree panel on the left side and a viewport – an embedded web browser – on the right side. The tree panel shows the views/sub-views of the model in a tree form with the names and icons. When a view is selected, Picto performs a sequence of transformations to generate and render the view's content on the viewport. For example, Picto displays the complete Social Network model as a Graphviz-based node-edge diagram of the selected `Social Network` view in

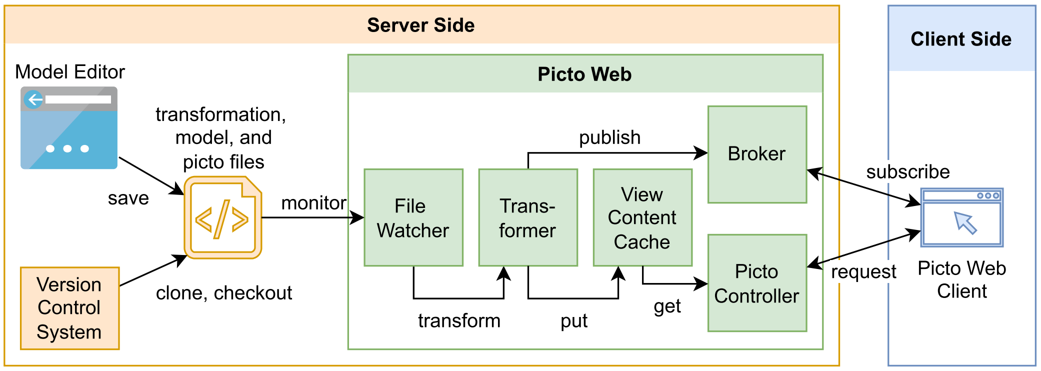


Fig. 2. The architecture of Picto Web [7].

Fig. 1. In addition to Graphviz, Picto can visualise models in PlantUML and SVG/HTML formats. It is also extensible, which enables users to add other diagram-as-code formats.

Nevertheless, as an Eclipse IDE plugin, Picto can also bring some hurdles to users since they have to install Java and Eclipse to use it. They must also locally check out the models they want to explore, including the corresponding visualisation transformations. For software developers who are already familiar with Eclipse, this is not an issue, but for other engineers and stakeholders, it can be a substantial challenge.

In this paper, we introduce Picto Web,[1] a web-based implementation of the Picto model visualisation tool. In contrast to Picto, Picto Web does not need to be installed locally, making it more appropriate for a broader range of stakeholders and developers who would benefit from having access to graphical representations of complex software and system models that are generated dynamically. In contrast to Sprotty and KLD, Picto Web and Picto support rendering views in multiple formats, such as table/form views in HTML, PlantUML and Graphviz diagrams, and views generated using JavaScript graphical libraries, e.g. Three.js.

Picto Web has been presented at the MODELS 2022 Tool and Demo track [7], and most of the content presented in this tool companion manuscript is based on that paper.

2. Software description

Picto Web is implemented as a multi-tenant client-server web application that can be accessed through a standard web browser to perform interactive complex model exploration. It retains the core features of the Eclipse-based version of Picto, such as lazy generation of views, hierarchical organisation of views, and support for multiple model and view formats. To provide a real-time user experience, Picto Web continuously checks model files and visualisation model-to-text templates for modifications and immediately propagates any changes to produced views to the viewers' web browsers.

2.1. Software architecture

Fig. 2 shows the two components that make up Picto Web's architecture: a server-side component that runs visualisation transformations against models and produces hierarchies of lazily-computed transient views, and a browser-based client that displays these views.

The server side of Picto Web consists of the `FileWatcher`, `Transformer`, `ViewContentCache`, `PictoController`, and `Broker` components. `FileWatcher` monitors the directory containing the models that Picto Web visualises (see Section 2.3) and the transformation and configuration files that define the transformation of the models to their visual representations. It detects any changes made to the files (added, removed, or updated) and notifies the `Transformer`, which subsequently creates, updates, or invalidates the contents of the corresponding views.[2]

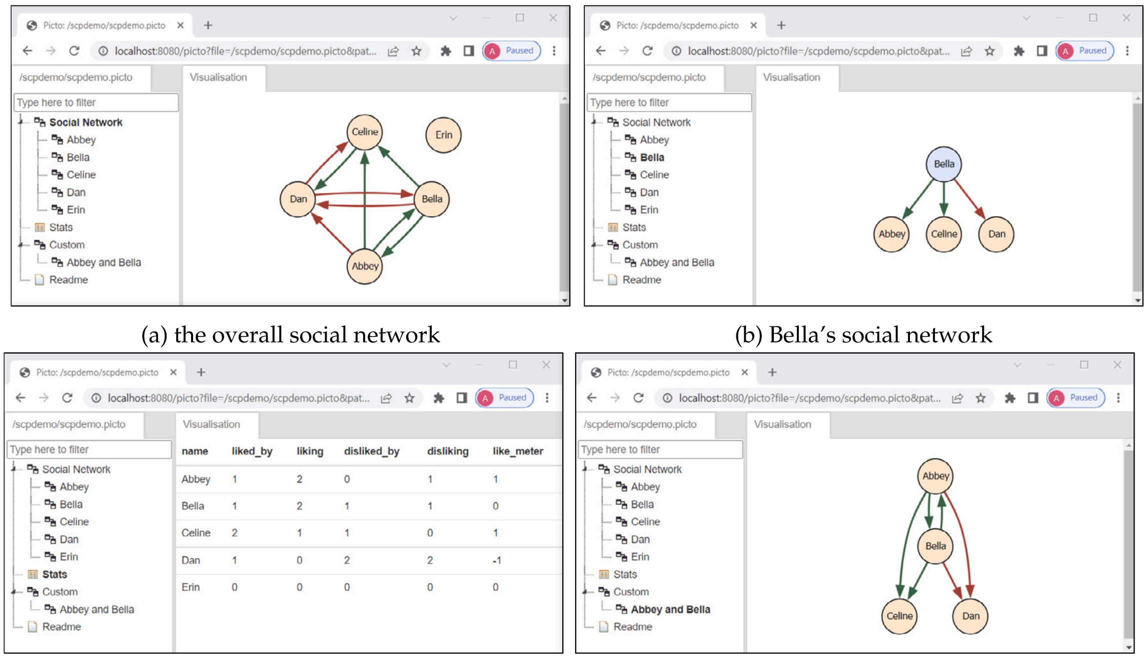
Picto Web then stores the created/updated views in the `ViewContentCache`, which maps the paths of the views in the tree view to their actual contents. By using the cache, unnecessary regeneration of views that have not changed can be avoided. This enables Picto Web to respond efficiently to recurring requests for the same views.

Every time a client asks for the content of a particular view, `PictoController` receives the request and responds by locating the requested view's content in the `ViewContentCache` using the view's path as the key. Picto Web then sends the view content to the client to display.

The `Broker` is responsible for sending changes to Picto Web clients. Every time a Picto Web client requests the results of a visualisation transformation, it also subscribes to the `Broker` to receive notifications about future changes to its results (e.g. due to

[1] https://github.com/epsilonlabs/picto-web.

[2] The work in [9] served as the foundation for the processes utilised in accomplishing incremental view invalidation and regeneration upon changes in models and templates. However, a further explanation would be outside the scope of this paper.



name	liked_by	liking	disliked_by	disliking	like_meter
Abbey	1	2	0	1	1
Bella	1	2	1	1	0
Celine	2	1	1	0	1
Dan	1	0	2	2	-1
Erin	0	0	0	0	0

(a) the overall social network (b) Bella's social network

(c) an HTML table summarising the social network (d) custom Abbey and Bella's social network

Fig. 3. Picto Web client displays different views [7]. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

changes in the underpinning models/templates of the views). In addition, the `Transformer` generates the model's content views and sends them to the `Broker` after transformation.

To simplify the deployment and testing of the Picto Web server, we have packed it as a Docker container[3] and can be run using the following command.

```
docker run --rm -it -v $PWD:/workspace -p 8080:8080 picto-web
```

The running Picto Web server can be accessed via a web browser using the address http://localhost:8080. The `-v $PWD:/workspace` option allows the Docker container to access the transformation and model files in the current directory from its internal `/workspace` directory. In this way, the internal `FileWatcher` in the Picto Web Docker image can keep watching the file system for any modifications that can cause view creation or invalidation.

2.2. Software functionalities

In this illustration, we demonstrate using Picto Web to produce transient views from a model that conforms to a contrived social network metamodel from [8]. The models contain five individuals (Abbey, Bella, Claire, Dan, and Erin), and their likes/dislikes relationships. From the model, we create one node-edge diagram of the entire social network and a diagram for each individual that includes anyone the person likes or dislikes.

In a view, a person is shown as a circle while the *dislike* and *like* relationships are presented as red and green directed edges, respectively, as illustrated in Fig. 3a. The node-edge view displays the social network model described in Listing 1.

Picto Web's user interface is shown in Fig. 3, and has two main parts. The right side is a panel that shows the content of the active, selected view, and the left side is a tree view that lists the views generated from the underlying model(s) hierarchically. In our example, Picto Web displays Abbey, Bella, Claire, Dan, and Erin's overall social network when a user chooses the *Social Network* view on the left panel (Fig. 3a).

Viewers can also use the left panel to see each person's local social network. For instance, Fig. 3b shows Bella's social network. Besides being able to display view contents in SVG/HTML, as seen in Figs. 3a and 3b, Picto Web can also render model views in other formats. As an illustration, Fig. 3c shows a social network summary as a HTML table.

As seen in Fig. 3d, developers can also create custom views in addition to the views produced through rule-based transformation (such as a view for each user in the network). For example, the custom view only shows Abbey and Bella's social network.

[3] The link to the container can be found at Picto Web's GitHub repository 1.

```
<?xml version="1.0" encoding="ASCII"?>
<SocialNetwork xmlns="socialnetwork" xmi:id="0">
  <people xmi:id="1" name="Abbey" likes="2 3" dislikes="4"/>
  <people xmi:id="2" name="Bella" likes="1 3" dislikes="4"/>
  <people xmi:id="3" name="Celine" likes="4"/>
  <people xmi:id="4" name="Dan" dislikes="2 3"/>
  <people xmi:id="5" name="Erin"/>
</SocialNetwork>
```

Listing 1: A social network model as the input file for the lazy transformation [7]. The format of the identifiers is simplified.

```
rule Network2Graphviz
transform n : socialnetwork::SocialNetwork {
  template : "socialnetwork2graphviz.egl"
  parameters : Map{
    "path" = Sequence{"Social Network"},
    "format" = "graphviz-circo",
    "people" = n.people
  }
}
```

Listing 2: The Network2Graphviz EGX rule.

```
digraph G {
  node[shape=rectangle, fontname=Tahoma, fontsize=10, style="filled", gradientangle="270", fillcolor="bisque:
       floralwhite"]
  edge[penwidth=3, style=tapered, arrowhead=none]
  [%for (p in people){%]
    [%=p.name%]
    [%for (l in p.likes){%]
      [%=p.name%] -> [%=l.name%] [color=green]
      [%}%]
    [%for (l in p.dislikes){%]
      [%=p.name%] -> [%=l.name%] [color=red]
    [%}%]
  [%}%]
}
```

Listing 3: An EGL template that generates a Graphviz representation of a social network [7].

```
digraph G {
  node[shape=rectangle, fontname=Tahoma, fontsize=10, style="filled", gradientangle="270", fillcolor="bisque:
       floralwhite"]
  ...
  Abbey
  Abbey -> Bella [color=green]
  ...
  Dan
  Dan -> Bella [color=green]
}
```

Listing 4: A view generated by the EGL template in Listing 3 [7].

2.3. Transformation

Picto Web requires at least three types of files: model files (in any format supported by Epsilon[4]), visualisation transformation files in the Epsilon Generation Language [10] (EGL, *.egl) and the EGL Co-Ordination Language[5] (EGX, *.egx), and configuration files (*.picto) which map models to visualisation transformations and define custom views. The XMI-based representation of the social network model of our running example is displayed in Listing 1. EGL templates produce individual views, and EGX programs specify how the EGL templates should be applied to various model elements. An example of the EGX transformation coordination for this scenario can be seen in Listing 2. Every `SocialNetwork` element in the source model is processed through rule `Network2Graphviz` to create a Graphviz-based view using the `socialnetwork2 graphviz.egl` template in Listing 3. The logic for creating the view content is defined by the EGL template of Listing 3. Finally, Listing 4 displays the Graphviz view produced by the transformation.

Configuration (*.picto) files define the transformation coordination and model files used to generate the contents of the target views. For example, Picto Web transforms the social network model in Listing 1 into the social network and person views displayed in Fig. 3 using the rules contained in the transformation coordination file `socialnetwork.egx` in Listing 2. The figure also includes a custom view showing only Abbey and Bella's social network (Fig. 3d) and a view showing a Markdown readme file.

[4] https://eclipse.org/epsilon/doc/emc.
[5] https://www.eclipse.org/epsilon/doc/egx.

```
<?nsuri picto?>
<picto format="egx" transformation="picto/socialnetwork.egx">
  <view path="Custom, Abbey and Bella" icon="diagram-ff00ff" type="Persons2Graphviz">
    <parameter name="names" values="Abbey, Bella"/>
  </view>
  <view path="Readme" icon="document" format="markdown" source="socialnetwork/readme.md" position="0"/>
</picto>
```

Listing 5: The Picto file that binds the model and the visualisation transformation [7].

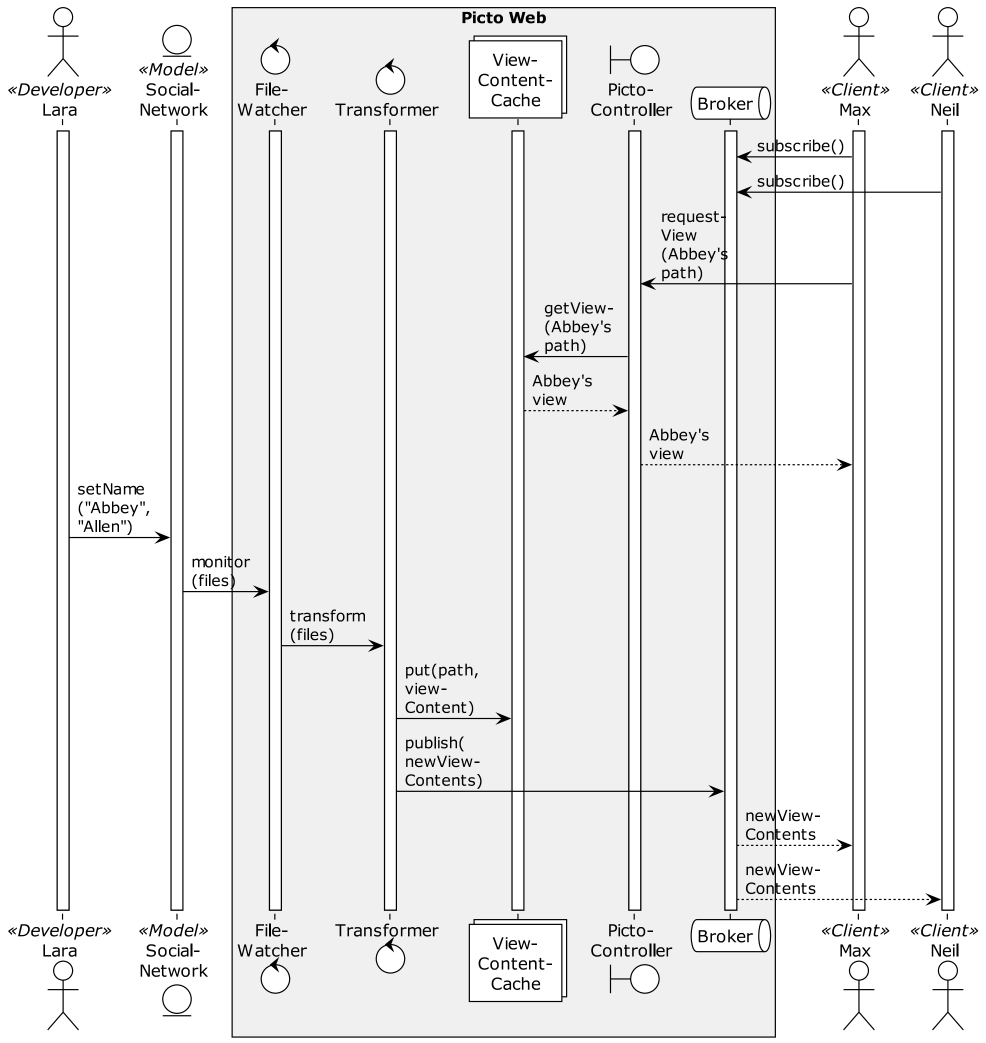


Fig. 4. Requests and Updates of Picto Web's view contents [7].

3. Illustrative examples

The internal control flow of Picto Web is illustrated in this section using the sequence diagram in Fig. 4. The first transformation occurs when Lara, a developer, constructs a social network model. The transformation produces view contents and stores them in the `ViewContentCache`. Next, other users (Max and Neil) want to explore Lara's model and launch Picto Web clients. These clients subscribe to the `Broker` of the Picto Web server to receive updates when the model is modified.

Max wishes to view Abbey's social network in the model. As a result, he makes a view request to Picto Web with the argument `path=/Social Network/Abbey`, which is Abbey's view path in the tree view (Fig. 3). `PictoController`, the component that handles the request, obtains Abbey's view content from the `ViewContentCache` since the path's view content is already available in the cache from the first transformation and sends it back to Max's client so that it can be shown.

At a later time, Lara changes the name "Abbey" to "Allen" and saves the modification to the model file. When a file is updated, the `FileWatcher` detects it and notifies the `Transformer`, which then creates new view contents from the modified file and updates the `ViewContentCache`. The `Transformer` also publishes the new content views to the `Broker` so that Max and Neil, the subscribers, can retrieve the new content views and refresh the views that are currently displayed to them.

4. Impact

Picto Web can facilitate efficient multi-user exploration of complex models in model-driven engineering settings. The following are some advantages it offers over existing tools [3,6,8]:

1. It enables users to carry out contextual exploration of complex models at various granularities and viewpoints.
2. It allows for the visualisation of models in various representation formats (e.g., HTML, Graphviz, SVG, PlantUML, etc.).
3. It offers a multi-user environment and supports immediate propagation of model modifications to viewers.

5. Conclusions

In this manuscript we presented Picto Web, a web-based implementation of the Picto model visualisation tool first developed as an Eclipse IDE plugin. In contrast to Picto, Picto Web does not need to be installed locally, making it usable to a broader range of developers and users who would benefit from using a tool capable of generating graphical representations of complex software and system models dynamically.

6. Future plans

In the future, we aim to further enhance the performance of visualisation transformations in Picto Web by using the element/property access traces of Picto Web transformations to enable more precise regeneration and invalidation of generated views. Another topic that merits investigation is extending the visualisation to allow model differencing so that it can inform viewers which parts of the models have been changed.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The work in this paper has been funded through the HICLASS InnovateUK project (contract no. 113213).

References

[1] N. Boccara, Modeling Complex Systems, Springer New York, New York, NY, 2010, pp. 1–23.
[2] R.E. Klosterman, Simple and complex models, Environ. Plan. B, Plan. Des. 39 (1) (2012) 1–6, https://doi.org/10.1068/b38155.
[3] Sprotty, eclipse/sprotty: a diagramming framework for the web, https://github.com/eclipse/sprotty, 2022. (Accessed 22 June 2022).
[4] Eclipse, Eclipse layout kernel (elk), https://www.eclipse.org/elk/, 2022. (Accessed 11 December 2022).
[5] Eclipse Foundation, Documentation: Eclipse graphical language server platform, https://www.eclipse.org/glsp/documentation/, 2022. (Accessed 23 June 2022).
[6] C. Schneider, M. Spönemann, R. von Hanxleden, Just model!—putting automatic synthesis of node-link-diagrams into practice, in: 2013 IEEE Symposium on Visual Languages and Human Centric Computing, IEEE, 2013, pp. 75–82.
[7] A. Yohannis, D. Kolovos, A. García-Domínguez, C.J.F. Candel, Picto web: a tool for complex model exploration, in: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 56–60.
[8] D. Kolovos, A. de la Vega, J. Cooper, Efficient generation of graphical model views via lazy model-to-text transformation, in: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 12–23.
[9] B. Ogunyomi, L.M. Rose, D.S. Kolovos, Incremental execution of model-to-text transformations using property access traces, Softw. Syst. Model. 18 (1) (2019) 367–383, https://doi.org/10.1007/s10270-018-0666-5.
[10] L.M. Rose, R.F. Paige, D.S. Kolovos, F.A.C. Polack, The epsilon generation language, in: I. Schieferdecker, A. Hartman (Eds.), Model Driven Architecture – Foundations and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 1–16.