International Journal of Software Engineering
and Knowledge Engineering
Vol. 25, No. 2 (2015) 277–305

DOI: 10.1142/S0218194015400100

World Scientific
www.worldscientific.com

Generating Reusable Visual Notations Using Model Transformation

Iman Avazpour* and John Grundy†
Centre for Computing and Engineering Software and Systems (SUCCESS)
Faculty of Science, Engineering and Technology
Swinburne University of Technology
Hawthorn, Victoria 3122, Australia
**iavazpour@swin.edu.au*
†*jgrundy@swin.edu.au*

Hai L. Vu
Centre for Advanced Internet Architecture (CAIA)
Faculty of Science, Engineering and Technology
Swinburne University of Technology
Hawthorn, Victoria 3122, Australia
hvu@swin.edu.au

Visual notations are a key aspect of visual languages. They provide a direct mapping between the intended information and set of graphical symbols. Visual notations are most often implemented using the low level syntax of programming languages which is time consuming, error prone, difficult to maintain and hardly human-centric. In this paper we describe an alternative approach to generating visual notations using by-example model transformations. In our new approach, a semantic mapping between model and view is implemented using model transformations. The notations resulting from this approach can be reused by mapping varieties of input data to their model and can be composed into different visualizations. Our approach is implemented in the CONVErT framework and has been applied to many visualization examples. Three case studies for visualizing statistical charts, visualization of traffic data, and reuse of a Minard's map visualization's components, are presented in this paper. A detailed user study of our approach for reusing notations and generating visualizations has been provided. 80% of the participants in this user study agreed that the novel approach to visualization was easy and 87% stated that they quickly learned to use the tool support.

Keywords: Visualization; visual notation; by-example transformation; notation composition.

1. Introduction

Using complex information in a visual format is more acceptable and effective for human beings in many circumstances, as visual representations use fuller capabilities of our powerful human visual system. They are particularly effective for graph-based

models or models with well-known and understood visual representations [1]. Visual notations are a key part of visualizations. They are an essential component in supporting interactions with the visualization. Generating these notations using programming languages has always been time consuming, error prone and difficult. It is also hardly a human-centric approach, requiring often detailed technical knowledge and a large distance between specification (in code) and visual notations.

This paper describes our new approach in generating visual notations to be used to generate complex visualizations. This approach has been inspired in part by the seminal Cornell Program Synthesizer (CPS) work [2]. CPS was designed to provide responsive and interactive program code editing. It used set of text-based templates that embodied the grammar of the programming language. Users would interact with these templates and generate code snippets and then compose them interactively to form a complete program code.

Similar to CPS, our approach considers a set of imported visual designs (the *view*) and user provided annotations as its *visual* templates. It automatically generates reusable visual notations from these templates. It then allows users to interactively map different data source elements to these notations using a highly human-centric drag-and-drop and by-example approach. Our toolset uses model transformations as the basis of implementing the mapping between the notation's intended information (that we call the notation's *model*) and the visual *view*. The defined notations can then be composed to generate complex visualizations. This approach is implemented in our CONcrete Visual assistEd Transformation (CONVErT) framework.[a]

This paper is organized as follows. The next section provides a brief overview [12] of closely related work. Section 3 describes our approach followed by three case studies in Sec. 4. Section 5 provides details of our user study evaluation and Sec. 6 provides a discussion. Finally, Sec. 7 concludes the paper and provides key areas of future work.

This paper is an extended version of the original paper presented at the International Symposium on Visual Information Communication and Interaction (VINCI 2014) [3]. It extends the original paper with modification to the approach with regards to notation generation where the notation data is generated automatically, support for visualization generation in Scalable Vector Graphics (SVG), updated annotation scripts to support attribute generation in SVG, and two new case studies to showcase the capabilities of the approach in visualization generation and reusability of visual notations.

2. Related Work

Many tools exist today for generating visualizations, e.g. Protovis [4] and CartoDB.[b] These tools provide conventional data visualizations for example charts and

[a] sites.google.com/site/swinmosaic/projects/convert
[b] cartodb.com

map-based visualizations. They do not however provide rich notation generation facilities and additional visualization capabilities are hard to define within these frameworks. For example, it is very hard to define a visualization for UML class diagrams, or flat designs like CAD drawings. Alternatively, powerful visualization scripting like D3 [5] require deep scripting knowledge. Our approach presented in this paper is focused on visual notations and how they can be incorporated and reused to generate visualizations.

One early approach to reuse notations for visualization was provided by Humphrey [6]. He introduced Relational Visualization Notation (RVN) for generating multi-dimensional visualizations. RVN is composed of three parts: semantic data models, graphics relations and design diagrams. The graphics relations provide binding between diagrams and informations using algebraic expressions. Design diagrams are directed, acyclic graphs that combine source relations to produce output graph relations. They combine multiple information and graphic relation into a visualization design specification.

Cerno-II is a visualization system capable of constructing graphical views of the execution state of object-oriented programs [7]. It uses display specification language to design new representations for displays. Each descriptor in this language is a functional expression specifying the general format of a type of display (boxes, lines, etc.). A specialization of Cerno-II for user interface component construction was provided in Skin [8]. Skin provides a visual functional language using icons and connectors. Visualizations are then formed by connecting these icons using connectors. Our approach is similar to these approaches, in that, the basis of notation design is on set of shapes defining the *view*, data descriptions (*model*), and mapping correspondences between them. However, these correspondences in our approach are provided using model transformations.

Ernst *et al.* provide visualizations for software application landscapes (software map) [9]. For each cluster map of the system, they have identified a semantic model and a symbol model and proposed to use transformations to link the gap between semantic model (the data to be visualized) and symbolic model (visualization) [9]. The approach provided by de Lara *et al.* also uses model transformations for manipulation of visual notations [10]. In their approach, syntax of notations is provided by meta models. Using these meta models, Domain Specific Visual Languages (DSVL) are defined and their manipulations are implemented using graph transformations [10]. The approach presented by Costagliola *et al.* also makes use of grammars for notation and visualization design [11]. In their approach diagramic notations are modelled using eXtended Positional Grammars (XPG). XPG is used for modelling both visual and textual notations. In their approach, visual notations are treated as visual languages where sentences are formed using set of visual symbols [11]. The Graphical Modelling Framework[c] (GMF) of Eclipse platform also helps modellers define a mapping from model elements to notational elements. In GMF

[c] http://www.eclipse.org/modeling/gmp/

however, data model, notation model and mapping model are all defined by meta-models and abstractions.

We take a very different approach to visual notation generation. In our approach, an existing visualization is imported and its visual fragments are used to create notations. This allows designers of visual languages and visualizations in general to adopt and use existing visual designs to suit their new or changing needs. The links between *model* and graphics and shapes (*view*) of the notations are provided by user-specified mappings. These mappings are implemented using model transformations. Once notations are defined, our approach allows us to use model transformations to map multiple input data to notation's *model*.

A key difference of our approach compared with current approaches is that by adding this transformation step, it is possible to reuse the generated notation specifications for wide varieties of different inputs and domains. In contrast, for most current approaches the visualizations are generated once for specific types of inputs. Also, using by-example transformation allows our approach to more readily incorporate user's domain knowledge in the visualizations process and hence provide a more user-centric visualization procedure. Similar to some current approaches, we use meta models for defining the syntax of visualizations. In our approach however, these meta-models are automatically reverse engineered from examples or defined in the background by the framework when users are composing notations.

The CONVErT framework used for implementation of this approach has been developed for generating model transformations using concrete visualizations. However, the visualizations in CONVErT were limited to set of predefined notations [12]. Using the approach presented in this paper, we demonstrate how reusable visual notations can be integrated into by-example transformation of CONVErT to expand its visualization and transformation capability. More specifically, we seek to address the following key research questions:

(1) Can we reuse existing visual designs to generate new visual notations?
(2) Can these visual notations be composed and linked together to generate more complex and complete visualizations?
(3) Can we reuse already defined visual notations for generating visualizations for different input models?

3. Approach

Our approach to generating visualizations is outlined in Fig. 1. This approach uses the already designed drawings as the starting point to generate visual notations. These drawings can be Scalable Vector Graphics (SVG) or Extensible Application Markup Language (XAML). Users import the drawings (or design new ones in the framework) and split them into different visual *views* depending on application of the visualization. For example, a bar chart in Fig. 1(a) is split into two *views*, an empty

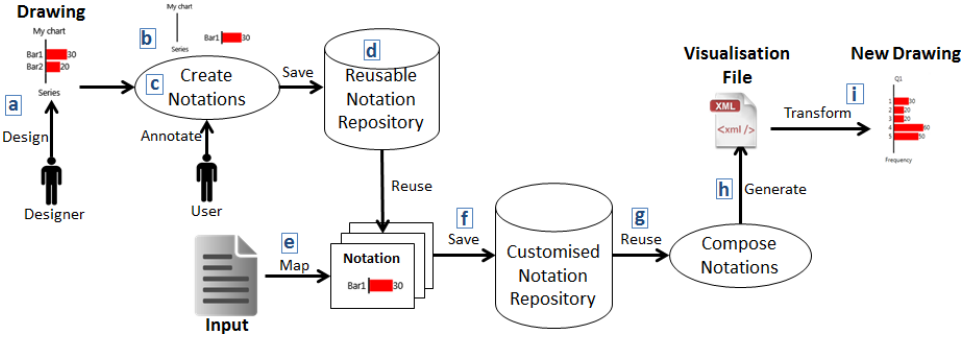


Fig. 1. Visualization procedure.

chart and a bar (Fig. 1(b)). This is because generally a bar chart visualization includes multiple bars and the bar notation needs to represent multiple data points.

The next step in the visualization procedure is to generate reusable visual notations from each *view*. The user (typically a visualization designer) needs to specify what each element (e.g. shapes, widths, heights, color, lines, etc.) of the *view* represents depending on application of the notation. For example, in our bar chart visualization, width of each bar represents a value. The designer therefore, needs to specify these element representations by a provided annotation language. Using these annotations, our approach generates a data part (we call it notation's *model*) and a mapping transformation between the data and the *view*. These three parts, the *view*, the *model* and the *model-to-view mapping* define a notation (Fig. 1(c)). The mapping transformation generates a *view* from a *model* by using the original *view* as a template and translating user-provided annotations to model transformation correspondences. If the *model* changes the *view* changes and hence the changes will be reflected on the visualization. For example if the values of bars in a bar chart are changed the mapping transformations reflect the new values to bar views to update bar views and refresh the visualization. Once defined, the resulting notation is saved in a notation repository to be (re)used for generating visualizations (Fig. 1(d)).

Notations in the repository can be reused by mapping elements of different inputs to their *model* elements. These mappings are provided using rule based model transformations. To visualise an input file using the defined notations, elements of input files are mapped to elements of the visual notation's *model* (Fig. 1(e)). This step defines how each part of the input file is to be represented using notational elements. Different inputs can be mapped to each notation that allows the notations to be reused for multiple visualizations. For example, bar notation of a bar chart could be mapped to sales data or population records or temperature listings. Once inputs are mapped, the already mapped notations (we call them *customized notations*) will be saved in a repository (Fig. 1(f)) for composition.

A visualization is generated by composing customized notations (Fig. 1(g)). This composition allows a visualization file's meta-model to be generated from the *models*

embedded in each notation. The notation-embedded *models* here play the role of meta-model's visual vocabulary. Using the meta-model resulting from the composition, a transformation is generated to transform the input file to the visualization file resulting from composition (Fig. 1(h)).

This visualization file is then transformed to visualizations using the mapping transformations embedded in notations. For example in Fig. 1, applying the transformation on the input file has resulted in the visualization marked by *i*.

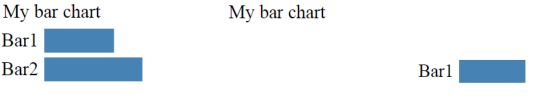
This new visualization approach has been implemented in CONVErT tool for generating interactive visualizations. CONVErT was initially developed for model transformation using concrete visualizations [12]. The visualization approach presented in this paper provides better flexibility in domain and types of visualizations that can be supported in pure model transformation tasks. On the other hand, by-example transformation and interaction mechanisms implemented in our CONVErT framework helped us to better incorporate a user-centric drag and drop approach into the visualization creation process. Also, the transformations used to transform input files to visualization files use the transformation code generator and engine of CONVErT. To better understand this approach, the following section provides case studies from some exemplar application domains.

4. Usage Examples

This section provides three case studies demonstrating the usage of our approach. The first is a simple example that demonstrates visualising input data as a bar chart using SVG graphics. This data can represent various domains from sales records to report generation. Our second case study demonstrates a more complex visualization of traffic data in XAML graphics, reusing these bar chart visualization elements. Our third case study is a Minard's Map visualization, with elements of its visualization reused to visualise traffic data as a complementary approach to the second case study.

4.1. *Case study 1: Generating a bar chart visualization*

Let us assume a bar chart visualization has been designed by a designer using SVG graphics similar to Fig. 2(a). Our intention is to reuse this design, generate reusable notations, and use them to visualize an input file representing set of sales records of a company.



(a) Bar chart (b) Empty chart view. (c) Bar view.

Fig. 2. Bar chart visualization design.

```
<svg xmlns="http://www.w3.org/2000/svg">
  <script type="text/ecmascript">
    <![CDATA[
      var yg = 10;
      function rect_load(evt) {
        yg = yg + 24;
        var mysvg = evt.target;
        var myrect = mysvg.getElementsByTagName("rect")[0];
        var mytext = mysvg.getElementsByTagName("text")[0];
        myrect.setAttribute("y", yg);
        mytext.setAttribute("y", yg+5);
      }]]>
  </script>
  <text x ="15" Y="5" width="100" dy="0.6em" text-anchor="center">My bar chart</text>
  <svg onload="rect_load(evt)">
    <text x ="60" width="40" dy="0.6em" text-anchor="end">Bar1</text>
    <rect width="59" fill="steelblue" x="65" height="20" />
  </svg>
  <svg onload="rect_load(evt)">
    <text x ="60" width="40" dy="0.6em" text-anchor="end">Bar2</text>
    <rect width="83" fill="steelblue" x="65" height="20" />
  </svg>
</svg>
```

Listing 1. Bar chart's *view*.

The representative code for the drawing provided by our designer in SVG is depicted in Listing 1. This Listing provides an SVG element representing the bar chart. It includes two rectangles with accompanying text as bars (each in a separate SVG element) and provides a function in Javascript for layout management of the bars. Both SVG and XAML allow additional processing and layouts using program code. For example an SVG graphic may use Javascript to perform layout management (like in Listing 1). XAML graphics may use additional coding in C# or Visual Basic as the code behind the visualization to perform similar tasks. Since our approach uses the initial *views* as templates, it will not interfere with any additional coding associated with the drawings that provides a strong capability for generating complex visualizations.

To generate notations for our bar chart, the drawing can be split into two *views*. A *view* as an empty chart (Fig. 2(b)) and a *view* for the bars (Fig. 2(c)). Accordingly, the SVG code for each view is provided in Listings 2 and 3. Two notations will be generated from these *views*. The various semantic constructs that these notations represent should be provided as the notation's *model*. In a bar chart visualization similar to Fig. 2(a), bars represent values of a certain category by visually depicting that value in the *view* using their width. Since multiple bars may exist in a bar chart, each bar is also accompanied by a name for the value it represents. Therefore the bar notation's *model* should provide these two elements. Consequently, it's notation's *view* should be annotated to reflect the correspondences between values of the *model* and and the *view*.

We have developed an annotation language to specify such *model-to-view* correspondences. These annotations specify correspondences between elements of the

```
<svg xmlns="http://www.w3.org/2000/svg">
  <script type="text/ecmascript">
    <![CDATA[
      var yg = 10;
      function rect_load(evt) {
        yg = yg + 24;
        var mysvg = evt.target;
        var myrect = mysvg... ;
        var mytext = mysvg... ;
        myrect.setAttribute("y", yg);
        mytext.setAttribute("y", yg+5);
      }]]>
  </script>
  <text ... >My bar chart</text>
</svg>
```

Listing 2. Barchart's SVG drawing code (some details are omitted to save space).

```
<svg onload="rect_load(evt)">
  <text ...>Bar1</text>
  <rect width="83" fill="steelblue" ... />
</svg>
```

Listing 3. Bar's SVG drawing code (some details are omitted to save space).

view, that could be XML elements or XML attributes, and name of the elements from the *model*. Table 1 provides a list of these annotations and a brief description. These annotations are different with regards to the type of correspondence relationship (one-to-one and one-to-many) and the type of output they will be generating in the *view*, e.g. contents for elements, contents for attributes or generating multiple elements.

As a rule of thumb in specifying notation *models*, variable characteristics of a notation should be specified by its *model*. In this example, the bar's length and name represent variables to be defined based on (to be visualized) input data. If color of the notations was also dependant on values of the input, a color would also have been considered as part of bar notation's *model*. Here bar notation uses default "steelblue"

Table 1. List of annotation used to generate mappings between notation's *model* and *view*.

Annotation	Description
linkto="Value"	A **one-to-one** relationship between *model* and *view*, generates element contents
@linkto=Value|value	A **one-to-one** relationship between *model* and *view*, generates attribute contents
callfor="Value"	A **one-to-many** relationship between *model* and *view*, generates multiple elements

```
<svg callfor="Bars" xmlns="..." >
  <script type="text/ecmascript">
    <![CDATA[
      var yg = 10;
      function rect_load(evt) {
        yg = yg + 24;
        var mysvg = evt.target;
        var myrect = mysvg... ;
        var mytext = mysvg... ;
        myrect.setAttribute("y", yg);
        mytext.setAttribute("y", yg+5);
      }
    ]]>
  </script>
  <text linkto="ChartName"... >My bar chart</text>
</svg>
```

Listing 4. Barchart's SVG drawing code.

color provided by the designer. To specify these correspondences, the visualization designer needs to specify annotations. For example, for the bar chart notation, the text to be represented on top of the chart needs to be provided by a model element, e.g. *ChartName* as in Listing 4. The linkto="ChartName" annotation specifies that the value to be represented by the text needs to come from a ChartName in bar chart notation's *model*.

Notations can incorporate, or host, other notations (e.g. a bar chart visualization will incorporate multiple bar notations. These one-to-many correspondences are specified by callfor annotations. For example in the Listing 4, the main SVG element has a callfor="Bars" annotation, which specifies a one to many relationship between the chart's main SVG element and the notation *models* included in Bars element of the *model*. The Bars element here is a placeholder in the chart *model* that defines the position which other notation models are to be placed in a host notation's model.

Similarly, Listing 5 provides the annotations used for bar notation. Here the value to be place as the bar's text is to be provided by a BName elements of the bar's *model* and the value to be provided to the width of the rectangle will be provided by a Value element in the chart *Model*. Note that since the width element is an attribute, and hence a one-to-one link to annotation for attributes has been used.

Annotated *views* are read by the transformation code generator in CONVErT and a mapping transformation script is generated for each *view*. In this transformation script one-to-one linkto annotations are translated to value fetch scripts and call for

```
<svg onload="rect_load(evt)">
  <text linkto="BName"...>Bar1</text>
  <rect width="@linkto=Value|20"... />
</svg>
```

Listing 5. Bar's SVG drawing code (some details have been omitted to save space).

```
<xsl:template match="HBarchartData" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <svg xmlns="http://www.w3.org/2000/svg">
  <text x="5" Y="5" width="100" dy="0.6em" text-anchor="center">
    <xsl:value-of select="ChartName" />
  </text>
  <xsl:apply-templates select="Bars" />
  </svg>
</xsl:template>
<xsl:template match="Bars" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <script type="text/ecmascript"><![CDATA[
        var yg = 10;
        function rect_load(evt) {
        yg = yg + 24;
        var mysvg = evt.target;
        var myrect = mysvg.getElementsByTagName("rect")[0];
        var mytext = mysvg.getElementsByTagName("text")[0];
        myrect.setAttribute("y", yg);
        mytext.setAttribute("y", yg+5);
        }
     ]]>
    </script>
    <xsl:apply-templates />
</xsl:template>
```

Listing 6. Barchart's *model-to-view* mapping transformation.

annotations are translated to call for templates (see Listings 6 and 7). As a result, when the mapping transformation script of bar chart notation is executed, it will fetch and copy the values provided to its *model* for the chart name to their corresponding text element in the *view*. It will also register a declarative call for templates to be applied on the data inside the "Bars" element of bar chart notation's *model*.

Additionally, the annotations will result in automatic generation of a *model* for each notation. In this paper, we use XML to represent *model* elements. However, our approach can be adopted to use other technologies as well. Listings 8 and 9 provide the automatically generated *model* for the bar chart and bar notations. Note the "isplaceholder" attribute in Bars element which have been automatically put there

```
<xsl:template match="HBarData" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <svg onload="rect_load(evt)">
    <text x="65" width="40" dy="0.6em" text-anchor="end">
      <xsl:value-of select="BName" />
    </text>
    <rect fill="steelblue" x="65" height="20">
      <xsl:attribute name="width">
        <xsl:value-of select="Value" />
      </xsl:attribute>
    </rect>
  </svg>
</xsl:template>
```

Listing 7. Bar's *model-to-view* mapping transformation.

```
<HBarchartData>
  <Bars isplaceholder="true">bars</Bars>
  <ChartName>My bar chart</ChartName>
</HBarchartData>
```

Listing 8. Barchart's *model*.

```
<HBarData>
  <BName>Bar1</BName>
  <Value>20</Value>
</HBarData>
```

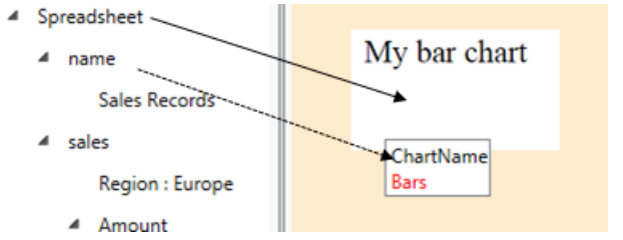
Listing 9. Bar's *model*.

when the system processes callfor annotations. Also, the values provided in the drawing have been used as default values. The attribute based linkto annotation can optionally provide a default value, in this case 20 was provided. Users can change these values according to their requirements. The name used for each notation's *model* is also provided by users once they build the notations, e.g. HBarchartData and HBarData.

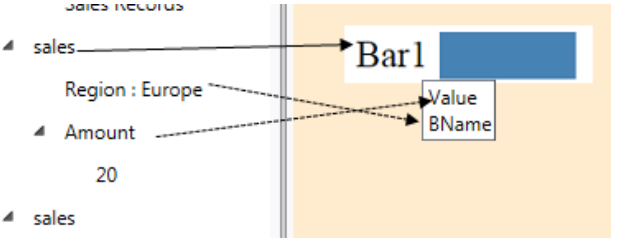
It is possible to configure the *model* to *view* mapping transformation to wrap each notation in a predefined interaction logic. For example it can be configured to wrap notations with drag and drop, and right click event handlers. As a result the notations generated can be dragged and dropped on a canvas or other notations to perform different tasks, or right clicking on each notation can be specified to reveal its *model* elements. This interaction can be altered according to application of notations. Examples of these wrapping mechanism have been used in CONVErT framework for model transformation where dragging and dropping notations performed model transformation tasks (e.g. see [13, 14]).

Once notations are generated, different input data can be mapped to their model elements. Although we could map input data directly to notation views using annotations, we have added this additional step so that the generated notations can be reused. This provides both an easier mechanism for (re)using notations in generating visualizations for multiple different datasets and an easier approach for novice users, given that the notations can be developed separately by a designer. This step provides the system with information required to create transformation rules for transforming specific parts of input data to the notation's *model* elements. Within our CONVErT framework, input data is shown using a default tree layout. Elements of the input model can be dragged and dropped on elements of notation's *model* to generate the input to notation's model transformation.

Figure 3 demonstrates how elements of input can be mapped to elements of both chart and bar notations using drag and drop. In this example, the input to be visualized represents the values of a sales record data in a spreadsheet. Each "sales"



(a) Mapping Spreadsheet element to chart notation.



(b) Mapping sales element to bar notation.

Fig. 3. Mapping input elements to notations.

is mapped to a bar with its region and sales amount representing BName and Value of the bar (see Fig. 3(b)). Similarly, the "Spreadsheet" element is mapped to bar chart notation with its name representing ChartName of the chart (see Fig. 3(a)). Note that "Bars" element placeholder is shown with different color to separate it from other elements of chart notation's *model*.

The defined customized notations are saved in a repository. These customized notations represent a transformation rule that transforms portions of input to notation's *model* (and its reverse where possible). As a result, a notation can be reused to create multiple customized notations for different inputs. Here for example, a transformation rule will be generated to transform each "Sales" element to a bar's *model*, and a transformation for transforming "Spreadsheet" to chart's *model*.

Once input data to notation transformation is complete, the defined customized notations need to be composed to generate a complete input to notation models transformation. An example of composing chart and bar notations to generate bar chart visualization is provided in Fig. 4. Linking a notation to a placeholder of

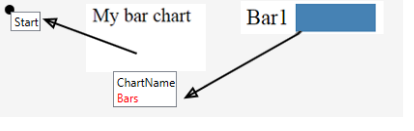


Fig. 4. Composing notations to create a visualization for a bar chart. Arrows are provided by framework to trace notation composition.

another notation presents the dragging notation's *model* to the placeholder element of host notations. This will specify that *model* of the dragging notation is to be copied inside the placeholder element of host notation. Since each customized notation also includes a transformation rule for transforming a portion of input data to the notation's *model*, this linking will also provide background logic so that the host notation knows that the transformation rule provided by the notation being dragged to it should be called. This is in order to effect the embedded input file to notation's *model* transformation and results in scheduling of input element-to-visual notation transformation rules. In our example, the bar customized notation's *model* is to be included in the "Bars" element placeholder of the chart notation, specifying that the chart may contain set of bars. This composition generates a grammar for bar chart where multiple bars can be provided inside the chart.

Linking a notation to a *start* element will define the root notation and hence the top-most (first to be run) transformation rule for the completed model transformation specification. This tells the transformation scheduler to start generating transformation code for transforming input file to notation *models* from the rule linked to this *start* element. For example, in Fig. 4, the transformation rule associated to chart customized notation (transforming Spreadsheet to chart notation's *model*) is the first rule to be called. It then calls the transformation rule associated with the bar notation accordingly to transform sales elements to bar notation models. The resulting model for this composition is provided in Listing 10. Note how *model* part of bar notations are copied into the Bars element of chart notation's *model* (due to space limitation, only one bar notations' model is shown). Also, the isplaceholder element previously present in the chart notation's model is removed after composition, as it will not be needed in the final result.

To render visual elements the resulting file from the input model to notation's models transformation needs to be transformed into a renderable visualization. This rendering in our approach reuses the *model* to *view* mapping transformation of notations available in the notation repository. When a resulting model file like Listing 10 is to be rendered (as a visualization), CONVErT checks the elements in the model file against the model part of the notations in the notation repository.

```
<HBarchartData>
  <ChartName>Sales Records</ChartName>
  <Bars>
    <HBarData>
      <BName>Europe</BName>
      <Value>20</Value>
    </HBarData>
    ...
  </Bars>
<HBarchartData>
```

Listing 10. Resulting model file from the composition in Fig. 4.

A visitor pattern is used to traverse this file for constructs similar to model part of notations where those *model* to *view* mapping transformations could be applied.

For example in Listing 10 the visitor will encounter a HBarChartData element that matches the bar chart notation in the repository, and a HBarData element that matches the bar notation's model. Accordingly, it will retrieve the mapping transformations embedded in these notations. These retrieved transformations will be put inside a transformation script and a declarative call for templates will initiate the chain of declarative calling of other transformation templates. This will result in transforming the resulted model file into renderable visualizations (in this case, SVG graphics). The resulting visualization will be rendered in the framework or can be exported to a browser-based visualization. For example, the resulting bar chart visualization of the visualization file in Listing 10 is depicted in Fig. 5.

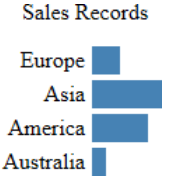
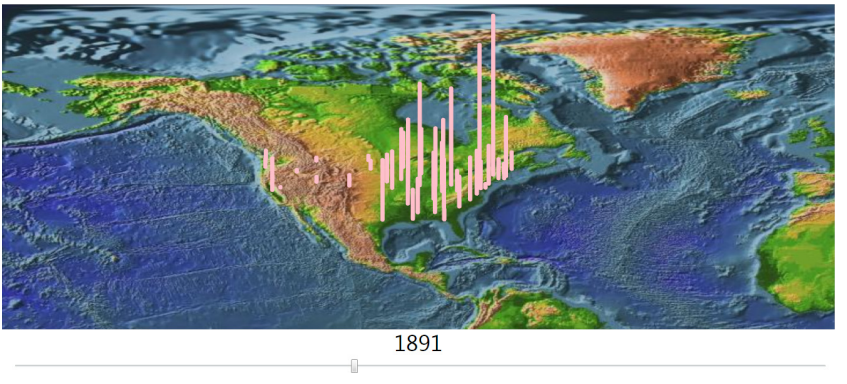


Fig. 5. Resulting bar chart visualization.

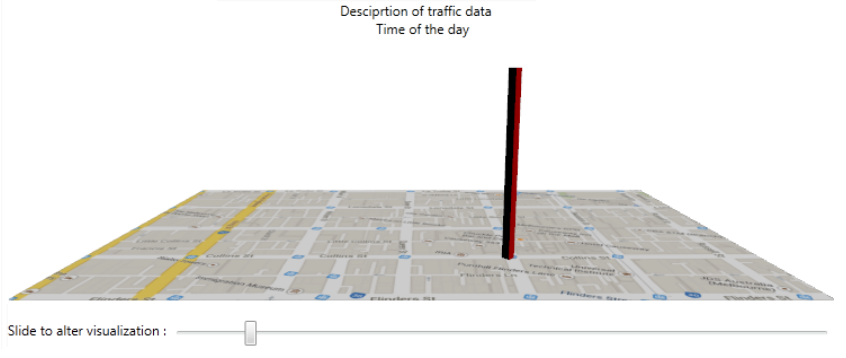
4.2. *Case study 2: Visualising traffic data*

Traffic congestion is an ongoing issue for modern cities around the world and it is important to understand how congestions form, monitor, and promptly take action against them. Analysis of congestion requires considerable knowledge of the network and is largely still based on the operator's past experience in dealing with local traffic. In this case study, we provide a visualization that enables the tracking of congestion through both temporal and spatial dimensions by displaying the number of cars passing a number of intersections (referred to as volume data) for set of particular intersections over time in a 3D map. This visualization will help to better monitor traffic volume and congestion.

Similar to our first case study, we assume a designer has generated a visualization of Melbourne CBD as in Fig. 6(b). This visualization is inspired by the 3D visualization of USA's population over time introduced by Petzold [15] (See Fig. 6(a)). It demonstrates population of each point of interest on a map using 3D bars. A slider is provided to navigate the visualization to depict data for different years. The 3D visualization provided by our designer exhibits similar features. It provides a slider bar to navigate between multiple frames where each frame demonstrates a traffic volume record at certain time. It also provides a 3D rendering of Melbourne CBD for each frame and a 3D bar. Our intention is to adopt this visualization for generating a time-lapse of traffic volume in Melbourne CBD for four intersections.



(a)

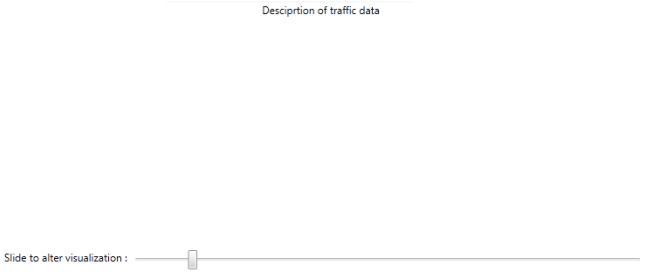


(b)

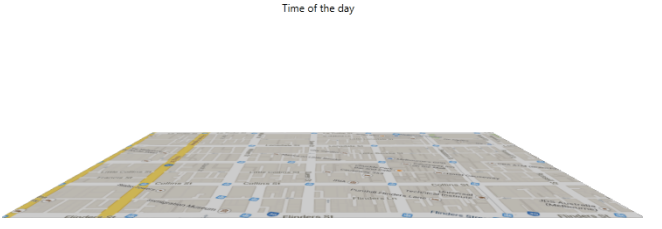
Fig. 6. Samples of 3D visualizations. (a) Visualization of USA population and (b) Traffic visualization provided by a designer.

To generate the required notations from the provided visualization design, we propose breaking it into three notations: 3D bar, map overview for a specific time, and a map host to review multiple congestion records on maps for different time frames as in Fig. 7. The combination of these three views is used to generate the complete visualization.

The *views* of Fig. 7 will reflect set of information provided to them on the visual shapes. For example, map host of Fig. 7(a) provides a description of the visualization and a horizontally laid out list that embeds maps like frames inside it. We intend to put Map *views* for each time frame in this list and as a result, the linked slider would provide navigation between frames. The Map host notation will then include other notations (in this case maps) as frames. Accordingly the map host notation would have a one-to-many relation (callfor annotation) for visuals and a one-to-one relation (linkto annotation) for description. Map notation provides a description for the map and will include multiple bars depending on the provided data. As a result it will



(a) Map host.



(b) Map.

(c) 3D bar.

Fig. 7. Traffic visualization notation *views*.

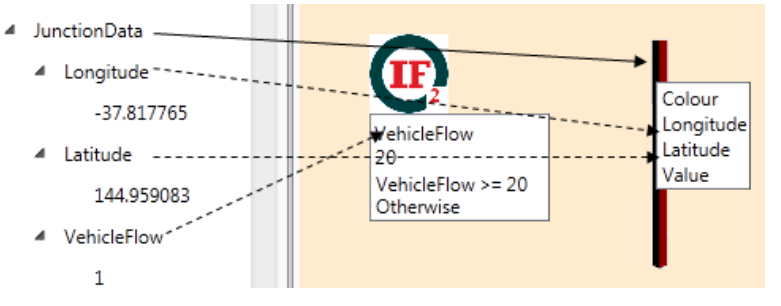
have a one-to-one relation (linkto annotation) with the description and one-to-many relation for bars. 3D bars to be laid out on the map take a value to be represented by their height, a color, and longitude and latitude to specify their position on the map. 3D bar notation therefore will have four one-to-one linkto annotations for these values. Listing 11 presents an example of these annotation for Map's *view*.

Using these annotations, the required *model* and the *model to view* mapping transformation for each notation is generated by the transformation code generator of our CONVErT framework. These notations will be saved in a notation repository and can be reused by mapping different input data to their *model* elements. In this example, we are assuming a traffic data file is provided which includes the volume data records of four intersections in Melbourne CBD. This data is provided using traffic sensors positioned in these intersections which record volume data for each five minutes. Using the transformation generation mechanism available in CONVErT,

```
<DockPanel ...>
  <TextBlock DockPanel.Dock="Top" ...>
   <TextBlock.Text linkto="MapDescription">desc</Text...>
  </TextBlock>
  <Canvas DockPanel.Dock="Bottom">
    <Viewport3D Canvas.Left="0" Canvas.Top="0"...>
      <Viewport3D.Camera>
        ...
      </Viewport3D.Camera>
      <ModelVisual3D>
              ...
      </ModelVisual3D>
      <Viewport2DVisual3D>
        ...
       <Viewport2DVisual3D.Geometry>
        ...
       </Viewport2DVisual3D.Geometry>
     </Viewport2DVisual3D>
   </Viewport3D>
   <Canvas callfor="TDBars">
   </Canvas>
 </Canvas>
</DockPanel>
```

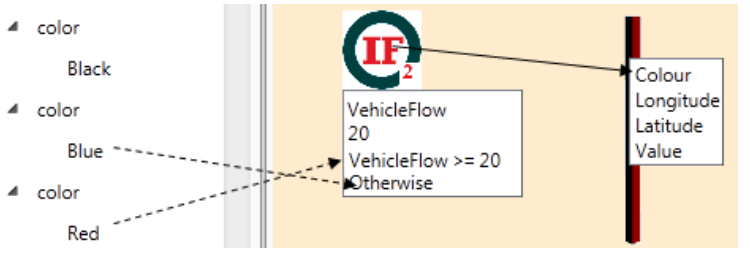
Listing 11. Map's annotated *view* (some details have been omitted to save space).

we can map these data records to our recently generated notations by drag and dropping each element on corresponding notational elements. In this example, we map each junction data to a bar representing its traffic volume as in Fig. 8. The color of the bar is calculated based on a threshold value. If traffic volume is more than or equal to 20 cars per five minutes, the bar should be red to indicate a warning, and blue otherwise. These colors are generated using the transformation conditions available in CONVErT framework. These conditions can be customized to suite every application. Similar to conditions, transformation functions are also available in CONVErT in case more complex correspondences needed to be defined, for example many-to-many, arithmetic, and string processing functions (for more information on these conditions and functions and how these transformations are



(a) Mapping elements to condition and notation.

Fig. 8. Mapping input data to 3D bar's notation. Arrows depict drag and drop.

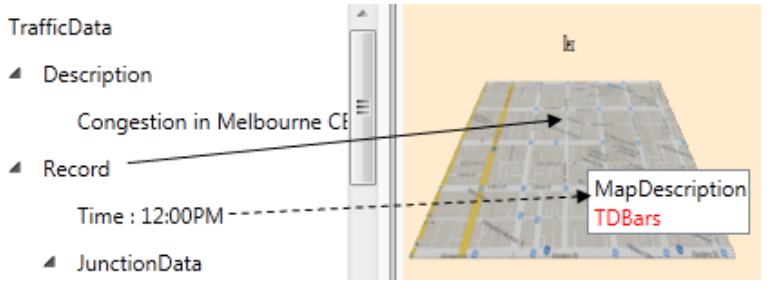


(b) Mapping colors to condition.

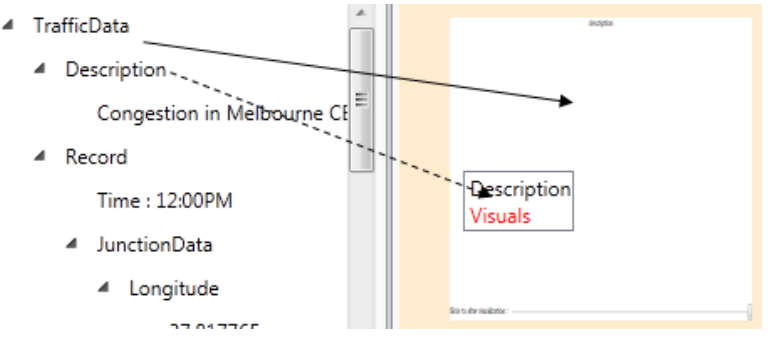
Fig. 8. (*Continued*)

generated, please refer to [16]). The VehicleFlow of the input also needs to be mapped directly to Value of bar's model (not displayed in the figure).

Each record element of the input data is to be mapped to a Map notation, with its time linked to Map's description as in Fig. 9(a). The traffic data element will be mapped to our Map host notation where its description is linked to host's description (see Fig. 9(b)).



(a) Mapping Record element to Map notation.



(b) Mapping TrafficData element to Map host notation.

Fig. 9. Mapping input data to notations. Arrows depict drag and drop.

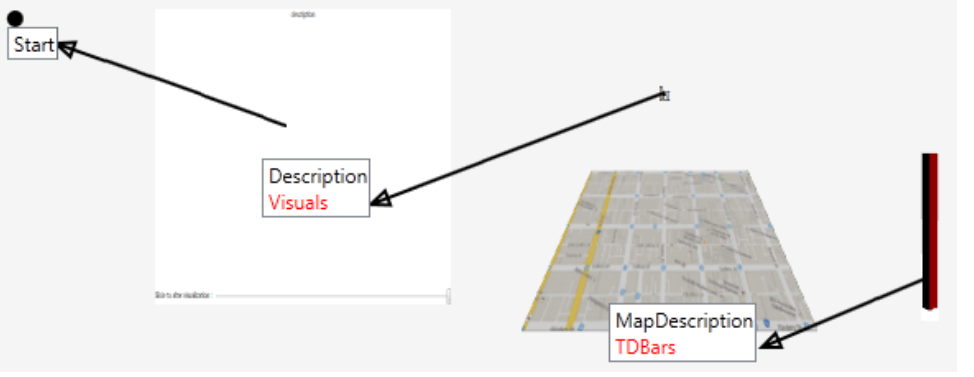


Fig. 10. Composing notations to create a 3D map visualization.

Saving the above specified mappings will result in generation of customized notations. Similar to our first case study, these notations can be composed to generate the meta-model for visualization file according to their placeholders (see Fig. 10). This composition results in a transformation from the input file to the visualization file conforming to the composition's meta-model.

The combinations of notation models resulting from this transformation will be transformed to a renderable visualization by reusing the *model* to *view* mapping transformations available in the notations of notation repository. The result of rendering the generated visualization file is presented in Fig. 11. Sliding the slider will result in an animated visualization of how traffic volume changes over time. In Fig. 11 the displayed frame represents traffic volume at 12:35 pm. It is interesting to note that if the input file is changed while conforming to the same meta-model (for example if data for other junctions is added to the input file), the same transformation can be used to generate updated visualizations. This is due to the fact that transformation rules for performing the input to visualization procedure are already defined.

Fig. 11. Resulting visualization of traffic data.

4.3. *Case study 3: Reusing Minard's map visualization for traffic data*

This case study shows an example of reusing the notations we used for regenerating a Minard's map for visualization of traffic information, complementing the approach above. Minard's map is a famous visual depiction of the French Grande Army's campaign for invasion of Russia in 1812 by Charles Joseph Minard. It is widely considered as one of the best statistical graphs by the visualization community [17], depicting number of troops, locations, campaign movements and status, and temperature information.

We have previously reproduced this map using two notations, a notation for troops movement and a notation for map visualization [14]. The troops movement notation (the blue shape in Fig. 12) is a shape constructed with two circles depicting number of troops at starting point and destination, and the lines connecting them. It represents number of troops at the starting city of the move (troopsEntered) and the number at destination city (TroopsLeft). Each city is represented by its X and Y (replacing Longitude and Latitude) and its name. The color of the shape also represents whether troops were advancing or retreating. This shape is generated using Windows Presentation Foundation and XAML. Based on the number of troops the radius of these circles is calculated in the accompanying C# code.

To reflect the data to be presented, this notation is mapped to troops records provided in a separate input file. User drag and drops the required elements of the input file on notation model elements and generates more complex correspondences using functions and conditions (see Fig. 12). Once these mapping are complete and the customized notations are generated, the composition of the troops movement notation and the map will result in the visualization show in Fig. 13.

We now show how we can easily reuse these visual notations (Map and Troops movement) used for Minard's map to represent traffic information on a map of city of

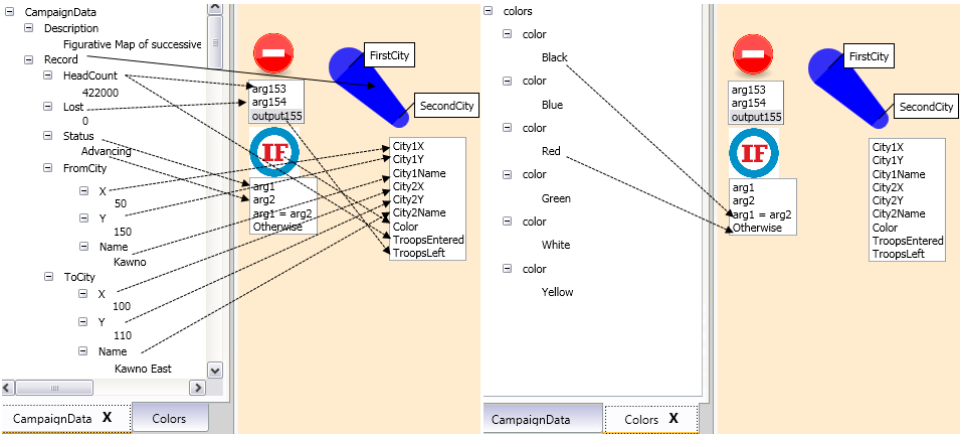


Fig. 12. Mapping input data to troops movement notation using functions and notations.

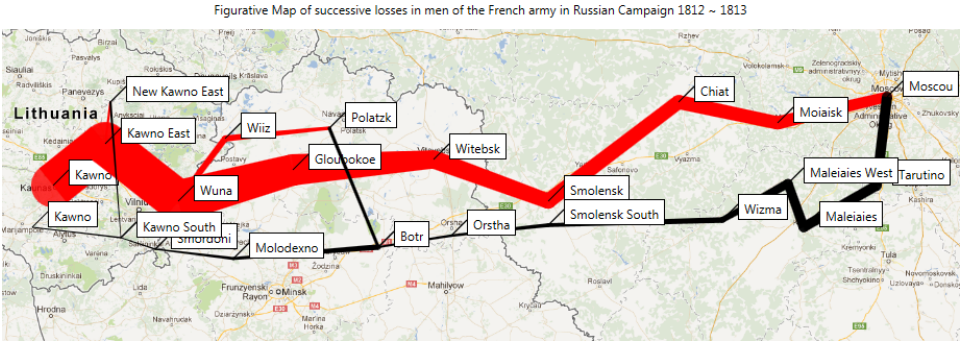


Fig. 13. Recreation of Minard's map using CONVErT.

Melbourne, Australia. For this visualization, each "movement" notation will be reused to represent the number of trips made between the Melbourne CBD and a selection of suburbs for a given day. We have collected this information using Household Traffic Survey data and now would like to generate a visualization to reflect on the collected information. Such a visualization would be used by urban geographers and local government to better understand travel choices of households in the Melbourne region. Such a visualization needs to show different colors based on the total number of trips made, i.e. if more than 10,000 trips are made, it should be shown in Red; between 5000 to 10,000 trips are shown in yellow; and less than 5000 should be shown in green.

To generate this visualization, we only need to make one change from the Minard map above — the image behind the map notation and its coordinates need to be changed to reflect the Melbourne area. The two notations (altered map and troops movement) can be now reused and mapped to the newly provided data. The "movement" visualization notation now represents people commuting from Melbourne CBD to various suburbs, instead of troop movements between cities.

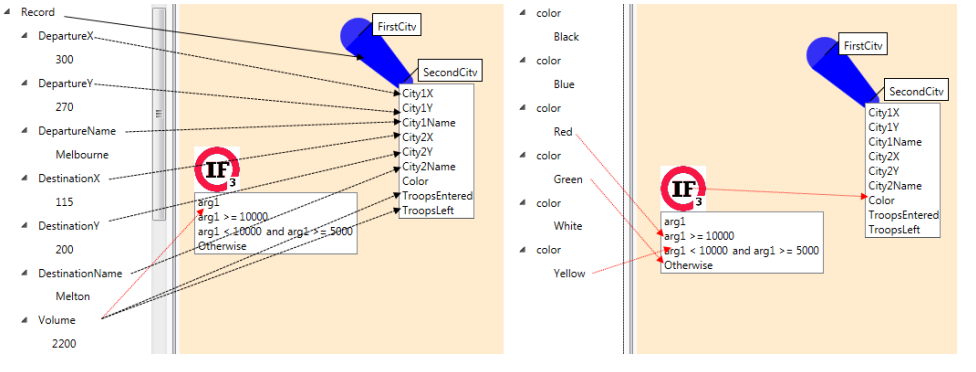


Fig. 14. Mapping trips survey data to our previously-defined troop movement visual notation. Arrows depict drag and drop directions.

Figure 14 demonstrates how the provided traffic survey information can be mapped to troops movement notation using drag and drop of input elements to notation model's elements and use of a condition. Note that in Fig. 14 the volume is mapped to both City1 and City2. This is due to the fact that unlike the case of troops movement were the number of troops declined during the movement, in this case the number of trips remains the same during the movement.

Once the mapping for both notations is done, they will be composed as shown in Fig. 15. The previously defined "map" visualization now represents Melbourne and

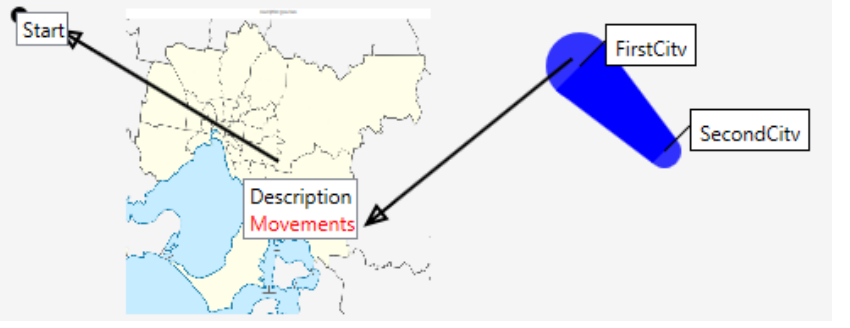


Fig. 15. Composing new customized notations for visualizing trips.

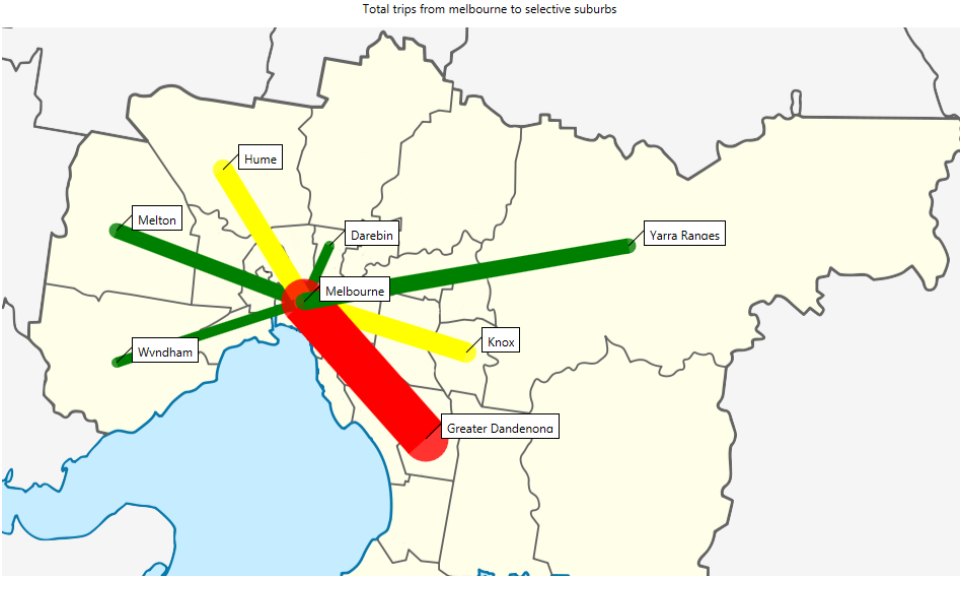


Fig. 16. Visualization of trips made between Melbourne CBD and its surrounding suburbs.

its surrounds. The "troop movement" notation now represents commuter trips between suburb ("SecondCity") and Melbourne CBD ("FirstCity"). The resulting visualization generated from this composition will transform the data provided by the Household Travel Survey dataset input file into the visualization shown in Fig. 16.

5. Architecture and Implementation

CONcrete Visual assistEd Transformation (CONVErT) [12], is our proof of concept framework to realise our approach to by-example, model-to-visualization and model-to-model transformations. CONVErT generates eXtensible Stylesheet Language Transformation (XSLT) implementations for both model-to-visualization and model-to-model transformations, and includes a transformation engine capable of running these transformations. CONVErT is implemented with C# and uses eXtensible Application Markup Language (XAML) and Scalable Vector Graphics (SVG) to render visual elements, making them interactable and providing polished, high-performance model renderings.

Each notation in CONVErT can incorporate a background logic that controls user interaction and notation's behavior. Model to view transformation of each notation can be configured to wrap the notation in a variety of interaction logics. For example, we have used this interaction logic to perform model transformations by drag and drop of visual notations [16]. This could be extended further to provide more fine grained information for example zoom-in and zoom-out functionality, or data wrangling and cleansing.

Since the model to view mapping of the notations is done using model transformations, notation views can use dedicated coding as well. For example, in our SVG bar chart, the layout was controlled by Javascript code embedded inside the SVG notation views. The logic for interaction with the map and the troops movement shapes in our second and third examples were provided in C# as code behind the notation views.

6. Evaluation

We have evaluated our approach and toolset in two ways. First, capability testing using various case study visualizations to show the effectiveness and applicability of the approach. We have implemented a wide range of visualizations and reused these in a wide range of target domains. To date, these include business data analytics [12], intelligent transport systems [3], a Minard's Map [14], CAD tool data integration [13], and code generation from various UML software models [16, 18].

Second, a user study to test target end user feedback on the usefulness of the approach and to examine how users react to such an approach, incorporating by-example visualization, drag and drop mapping specification and reuse and composition of visual notations. This section provides the details of our user study.

6.1. *User study method*

To perform this study, we recruited 19 users (including 4 controls for instrument testing). Users were selected from software engineering staff and students and were assigned into two groups. A 10 minute screencast was provided to each participant which described CONVErT framework's user interface and the visualization generation procedure. Participants were then asked to perform a set of given visualization tasks following think-aloud approach. The experimental setup comprized a laptop with an attached mouse. Screen captures were taken during the process and a matching questionnaire with 21 visualization related questions was handed to each participant at the end of the experiment. This questionnaire was designed using 5-point Likert scale (ranging from strongly disagree to strongly agree) and provided dedicated spaces to leave comments and optional feedback.

Participants were asked to choose between two experiments: one group would test application of our approach for business domain and the other group would evaluate the approach in software engineering domain. They were then asked to create a visualization with CONVErT. Both groups had the same settings but used different input models and visualizations. Our rationale for this was to test research question 3 with regards to using the approach for different domains and input models.

The first group was given an input file representing business sales data and were asked to create a bar chart visualization of their sales data. The second group were given a class diagram data (XML) and asked to generate a class diagram visualization. Task description hard copies were handed to the participants and did not describe instructional steps. Instead, they included the input file names and their locations, and a snapshot of the desired final visualization result. Users had to come up with steps required to get similar results. They were allowed to ask questions from the instructor if they had trouble understanding those steps.

6.2. *User study results*

Our first group consisted of ten participants (8 male, 2 female). The second group consisted of 5 participants (3 male, 2 female). In response to demographic question **D.4**: "How familiar are you with data visualization?", the participants had following options: **VF**: Very familiar, **SF**: Somewhat familiar, **HH**: Had heard of it, and **NF**: Not familiar. The percentage of responses are provided in Table 2. Table 2 also provides how participants describe their area of expertise (Question **D.5**). The

Table 2. Partial participant demographics. Numbers are percentages.

Question	**VF**	**SF**	**HH**	**NF**
D.4	13	60	13	13
Question	**SE**	**CS/IT**	**EC**	**OT**
D.5	47	40	0	13

Table 3. Sample questions of our user study questionnaire.

	Question
Q.1	I found it easy to visualise the given data as a bar chart/class diagram.
Q.2	I learned to use the tool quickly.
Q.3	In general I found the tool to be easy to use for visualization activities.
Q.4	I easily remember how to use the tool.
Q.5	Some things do require a lot of thought.

Table 4. User responses to questions of Table 3.

Question	5 (%)	4 (%)	3 (%)	2 (%)	1 (%)	Median	Mean	Mode
Q.1	73	7	20	0	0	5	4.53	5
Q.2	60	27	13	0	0	5	4.47	5
Q.3	47	40	13	0	0	4	4.33	5
Q.4	33	60	0	7	0	4	4.20	4
Q.5	7	47	20	20	7	4	3.27	4

options are **SE** for software engineering, **CS/IT** for computer science or information technology, **EC** for economics and **OT** for other areas.

Table 3 demonstrates a selection of five questions from our questionnaire targeted at ease of use and understandability of the approach and toolset. We have assigned scores of 1 (for perfect negative) to 5 (perfect positive) to each Likert point. Frequency of responses to sample questions based on these arrangements are summarized in Table 4. Full results can be found in CONVErT's website.[a] Please note that question Q.5 is a negative question and the responses have been accordingly altered to reflect this, i.e. those answering strongly agree have been mapped to strongly disagree and so on.

As the user study demonstrates, users of both groups positively liked the visualization approach and the majority of participants (60% strongly agree and 27% agree) agree that learning the tool was easy. However, we should note that since this user study was to evaluate usability of the notations and comprehension of our approach in visualization, users were provided with predefined notations. Given that users were required to annotate views to generate notations, we would anticipate to have slightly different results, since users would have to have basic understanding of XAML or SVG representations to understand the elements of visual *views*.

6.3. *Threats to validity*

There are certain threats to validity of our user study results with regards to participant number and affiliation.

Internal threats to validity impact the degree of *bias* of a study. As our participants were mostly recruited from staff and students of Swinburne university, their affiliation might have introduced bias in their responses to our questionnaire. As the demographics of Table 2 demonstrates, our users were mostly familiar with at least

one visualization technique (11 out of 15). This could have affected the results of our visualization evaluation. However, our participants are broadly representative of our target end user groups (visualization designers and visualization re-users).

External threats to validity impact the degree to which a study can be generalized. With our current number of participants to date, statistically significant and more generalizable inferences cannot be made. Hence, this user study is a work in progress and we intend to further evaluate this approach with more participants. We will update our online results as we recruit more participants. However, based on the user participants recruited to date, our tool and approach appear to have broad appeal to our target domain of end users and their likely expertise.

Construct threats to validity impact the degree a study measures its intended causes and effects. We chose to provide two subsets of our user participants two quite different kinds of visualizations and asked them to perform sets of tasks, to reduce potential domain-oriented influences on the outcomes. In this study we wanted to investigate if interactive concrete visualizations better support using users' domain knowledge in general, and not for specific domains. As a result, using two very different visualizations and visualization domains help us examine if the approach had similar effects on users of different domains (software engineering and business analysis in this case).

7. Discussion

The two very different case studies presented in this paper have helped to demonstrate how a separately designed visualization can be effectively reused to generate reusable visual notations. We have shown that our approach is effective in supporting complex visualization specifications using a by-example approach. We have shown that these visualization specifications can be effectively reused in multiple, diverse domains by mapping target domain input data elements to visualization specification elements. Our visualization support generates a variety of target renderable visualizations, optionally with interaction support. This includes WPF components, XAML, SVG, Javascript and C#. We have deployed our approach to specify a wide range of complex visualizations, many reusing the same pre-defined visualization components in very different domains. We have conducted a user study to gain feedback on the usability and likely acceptance of our approach and prototype toolset, using representative target end users. This has been largely positive to date.

We believe the approach presented here can be used with other technologies with minor alterations. This has been demonstrated by being able to readily generalise our original WPF and XAML-only approach to SVG, which in turn supports diverse rendering technologies. Hence, these technologies address our first key research question, and form our major research contribution — we can effectively reuse existing visual designs to generate new visual notations.

Our experiences to date also demonstrate that our second research question is also (partially) answered. Our user study shows that our target end user group appears to

be able to effectively use this concept in the CONVErT realization environment. Visual notations can be composed and linked together to generate more complex and complete visualizations, at least to the degree that our various case studies to date have shown. However, very complicated or novel visualization components may need to be implemented using C# or Javascript. CONVErT provides support for specifying novel visualizations as customized components in C# as well [14].

Our case studies, experiences with CONVErT to date, and our user study all partially answer our third research question, that we can reuse already defined visual notations for generating visualizations in different input models. Our case studies show the effectiveness of our visualization approach with regards to mapping input data to the already-designed notations, composing new customized notations, and generating concrete visualizations. These contributions were further investigated with our user study which examined target end user experience with our new visualization approach.

Considering that the mapping between input data and visual notations is performed using rule based model transformations, any data could have been used for the input to visual notation transformation step. For example, the traffic data of second case study could have been mapped to our bar chart visualization of case study one to form visual analytics charts. This also answers our third research question on reusing already defined visual notations for generating visualizations for different input models.

The case studies demonstrated in this paper were targeted to show capabilities of the approach and to clearly illustrate the capabilities of the proof of concept framework. This approach is capable of generating much larger scale and complex visualizations. However, certain considerations should be made. For example, where the visualization's dimension become very large, although the panels provided in the tool can stretch to house bigger renderings, one might use zoom-able panels in notation's view design. These panels can be embedded as part of top most notation's view by the designer of the initial visualization and are independent of the approach. Once required notations are composed, the embedded notations can be zoomed in and out accordingly.

Since the visualization generation in this approach relies on visual notations, the time spent to generate the visualization is a function of diversity of notations used in the visualization rather than the number of notations. For example, a bar chart has two types of notations; similarly a heat map has two notations as well (a map, and colored geographic boundaries). Therefore, the time spent to generate these two visualizations, although very different, are similar. Accordingly, generating visualizations with more diverse types of notations will be more time consuming. In large scale data visualization, the number of notations generally increases while their diversity remains the same. This is in fact strength of our approach where the complexity and computation time for visualization depends on notation types rather than their numbers.

8. Conclusions and Future Work

We have presented a new approach for generating visual notations and forming concrete visualizations. This approach allows reusable notations to be created from visual designs and provides a platform for linking varieties of input data to these notations. By composing these notations a meta-model for final visualization and a transformation from an input file to visualization is generated. This approach has been implemented in our CONVErT framework and applied on multiple visualization applications. We have evaluated our approach and its tool support in a user study and the results provide general acceptance of the approach and the use of drag and drop for visualization generation.

The approach provided here might be seen as an alternative to the Model View View Model (MVVM) provided in WPF. However, our approach does not limit use of MVVM since the MVVM can be still used in the provided views. As a result, the framework provides both MVVM and transformation-based model to view mapping. For example in our third case study, calculation of the circles for the troops movement shapes is done in their MVVM. Similarly, other external technologies can be integrated into notations as an extension. For example live refreshment of visualizations or external layout mechanisms [18].

Part of our future work will be focused on further evaluation of the approach perhaps with practitioners in the field of media and information visualization. We are also working on extensions of current approach to be able to customise and define interaction during notation generation. These interactions could include zoom-in and zoom-out functionalities, or provide drill-down or hide/show visual elements, or to embed further data relations in the visualizations. An example is where a pie chart has been visualized representing percentage of people who voted for certain product. By clicking on a pie piece in this visualization, it would be possible to show what percentage of them were male and what percentage were female.

It is common in visualization community to assume the data to be visualized is pristine and satisfies certain formatting and quality required [19]. However, it is not often the case and our visualization approach is not an exception. For example in the case study above, we have assumed that the input data is always complete and the records are sorted according to their representative time. We are working on data wrangling and cleaning approaches to catch data inconsistency and flaws before visualization. Similar functionality can however be provided using the transformation facilities within current version of the CONVErT framework.

Acknowledgments

This work is partially supported by the ARC Discovery Project (DP140102185) and ARC Future Fellowship (FT120100723) grants. Support for the first author from Swinburne University of Technology is gratefully acknowledged.

References

1. D. L. Moody, The physics of notations: Toward a scientific basis for constructing visual notations in software engineering, *IEEE Trans. Software Engineering* **35**(6) (2009) 756–779.
2. T. Teitelbaum and T. Reps, The cornell program synthesizer: A syntax-directed programming environment, *Commun. ACM* **24**(9) (1981) 563–573.
3. I. Avazpour, J. Grundy and H. Vu, Generating reusable visual notations using model transformation, in *7th International Symposium on Visual Information Communication and Interaction*, 2014, pp. 58–67.
4. M. Bostock and J. Heer, Protovis: A graphical toolkit for visualization, *IEEE Trans. Visualization and Computer Graphics* **15**(6) (2009) 1121–1128.
5. M. Bostock, V. Ogievetsky and J. Heer, D3: Data-driven documents, *IEEE Transactions on Visualization and Computer Graphics* **17**(12) (2011) 2301–2309.
6. M. C. Humphrey, Creating reusable visualizations with the relational visualization notation, in *Proceedings of the Conference on Visualization*, 2000, pp. 53–60.
7. S. Fenwick, J. Hosking and M. Warwick, A visualisation system for object-oriented programs, in *Technology of Object-Oriented Languages and Systems*, 1994, pp. 93–103.
8. J. Hosking, S. Fenwick, W. Mugridge and J. Grundy, Cover yourself with skin, Software Verification Research Centre, Department of Computer Science, University of Queensland, Technical Report, 94, 1994.
9. A. M. Ernst, J. Lankes, C. M. Schweda, A. Wittenburg and E. Denert-Stiftungslehrstuhl, Using model transformation for generating visualizations from repository contents, Technical report, Technische Universität München, 2006.
10. J. de Lara and H. Vangheluwe, Defining visual notations and their manipulation through meta-modelling and graph transformation, *Journal of Visual Languages and Computing* **15**(34) (2004) 309–330.
11. G. Costagliola, V. Deufemia and G. Polese, A framework for modeling and implementing visual notations with applications to software engineering, *ACM Trans. Software Engineering and Methodology* **13**(4) (2004) 431–487.
12. I. Avazpour and J. Grundy, CONVErT: A framework for complex model visualization and transformation, in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2012, pp. 237–238.
13. I. Avazpour, J. Grundy and L. Grunske, Tool support for automatic model transformation specification using concrete visualizations, in *IEEE/ACM International Conference on Automated Software Engineering*, 2013, pp. 718–721.
14. I. Avazpour and J. Grundy, Using concrete visual notations as first class citizens for model transformation specification, in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2013, pp. 87–90.
15. C. Petzold, *3D Programming for Windows* (O'Reilly, 2010).
16. I. Avazpour, Towards user-centric concrete model transformation, Ph.D. dissertation, Swinburne University of Technology, 2014.
17. E. R. Tufte, *Beautiful Evidence* (Graphics Press, Cheshire, 2006).
18. I. Avazpour, U. Rüegg and J. Grundy, CONVErT meets KIELER: Integrating advanced layout algorithms into by-example visualizations, in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2014, pp. 199–200.
19. S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck and P. Buono, Research directions in data wrangling: Visuatizations and transformations for usable and credible data, *Information Visualization* **10**(4) (2011) 271–288.