Generating Reusable Visual Notations using Model Transformation

Iman Avazpour
Centre for Computing and Engineering Software and Systems (SUCCESS)
Swinburne University of Technology
Hawthorn 3122, VIC, Australia
iavazpour@swin.edu.au

John C. Grundy
Centre for Computing and Engineering Software and Systems (SUCCESS)
Swinburne University of Technology
Hawthorn 3122, VIC, Australia
jgrundy@swin.edu.au

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

ABSTRACT

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 visualisations. Our approach is implemented in the CONVErT framework and has been applied to many visualisation examples. Two case studies for visualising statistical charts and visualisation of traffic data are presented in this paper. A detailed user study of our approach for reusing notations and generating visualisations has been provided that shows good reusability and general acceptance of the novel approach.

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 visualisations. They are an essential component in supporting interactions with the visualisation. Generating these notation visualisations 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 a new approach that allows users to generate complex visualisations using a drag and drop, visual and interactive approach which is much more human-centric. In our new approach a variety of visualisations can be imported or designed to be used as a notation's *view* representation. Our new approach uses model transformations as the basis of implementing the mapping between notation's intended information (that we call the notation's *model*) and the *view*. Once notations are defined, elements of input files to be visualised are mapped to notation's *model* using rule based transformations. These rule based transformations allow use of functions and conditions to provide high flexibility for mapping wide varieties of inputs to a notation's *model*, and hence improve reusability of defined notations. Composition of these notations is used to generate full model visualisations. Once a visualisation is generated for one example set of data, another data set conforming to the same meta-model can be visualised by reusing the defined visualisation. This approach is implemented in our CONcrete Visual assistEd Transformation (CONVErT) framework[1].

This paper is organised as follows. The next section provides a brief overview of closely related work. Section three describes our approach followed by two case studies in section four. Section five provides details of our user study evaluation and section six provides a discussion. Finally section seven concludes the paper and provides key areas of future work.

RELATED WORK

One early approach to reuse notations for visualisation was provided by Humphrey [2]. He introduced Relational Visualisation Notation (RVN) for generating

[1] sites.google.com/site/swinmosaic/projects/convert

VINCI 2014, August 5–8, 2014, Sydney, Australia.

multi-dimensional visualisations. 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 visualisation design specification.

Cerno-II is a visualisation system capable of constructing graphical views of the execution state of object-oriented programs [3]. 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 specialisation of Cerno-II for user interface component construction was provided in Skin [4]. Skin provides a visual functional language using icons and connectors. Visualisations 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 visualisations for software application landscapes (software map) [5]. 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 visualised) and symbolic model (visualisation) [5]. The approach provided by de Lara et al. also uses model transformations for manipulation of visual notations [6]. 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 [6]. The approach presented by Costagliola et al. also makes use of grammars for notation and visualisation design [7]. 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 [7]. The Graphical Modelling Framework[2] (GMF) of Eclipse platform also helps modellers define a mapping from model elements to notational elements . In GMF however, data model, notation model and mapping model are all defined by meta-models and abstractions.

We take a different approach for notation generation. In our approach, an existing visualisation is imported and its visual fragments are used to create notations. This will allow designers of visual languages and visualisations in general to adopt and use any visual design to suit their needs. The links between *model* and graphics and shapes (*view*) of the notations are provided by mappings 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 with current approaches is that by adding this transformation step, it is possible to reuse the once generated notations for varieties of different inputs and domain. Whereas in current approaches, the visualisations are generated once for specific types of inputs. Also, using by-example transformation allows our approach to incorporate user's domain knowledge in the visualisations process and hence provide a more user centric visualisation procedure. Similar to some current approaches, we use meta models for defining the syntax of visualisations. 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.

CONVErT framework that is used for implementation of this approach, has been developed for generating model transformations using concrete visualisations. The visualisations in CONVErT however were limited to set of predefined notations. 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 visualisation and transformation capability. More specifically, we seek to address following key contributions:

1. Can we reuse already existing visual notational element designs to generate new visual notations?
2. Can we effectively define correspondence links between data and visual notation following a by-example and user-centric approach?
3. Can these customised visual notation definitions be composed and linked together to generate more complex and complete visualisations?
4. Can concrete model visualisations be effectively generated using a visual and interactive by-example approach?
5. Can we reuse already defined visual notations for generating visualisations for different input models?

APPROACH

Our approach to generating visualisations is outlined in Figure 1. We use three different artefacts: 1. renderable visual materials or visualisations (Figure 1.1 and 1.8), 2. the file underlying each visualisation or visualisation file, and 3. input file to be visualised (Figure 1.5). As the figure demonstrates, our model visualisation procedure starts by reusing an already existing visualisation (Figure 1.1). We assume a designer has designed the shapes to be used in the visualisation, in this case a house. The visualisation designer, who is also our target end user of the visualisation, splits the visual design into different visual *views* depending on application of

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

the visualisation. For example, in Figure 1 the house has been split into four *views*: roof, door, surrounding walls and window (Figure 1.2). The collection of these parts defines a house's visualisation.

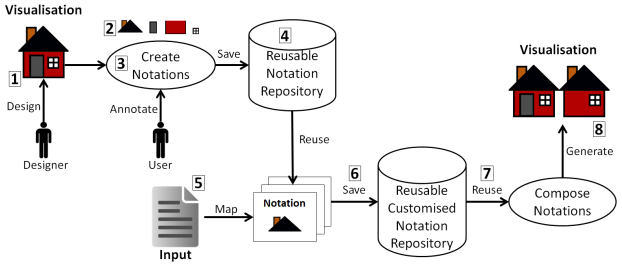


Figure 1. Visualisation procedure.

The next step in the visualisation procedure is to generate reusable visual notations from each *view*. Visual (or indeed textual) notations in our approach are represented by a collection of shapes, text and graphics that define the *view* of the notation (provided separately by a designer). A data part specifies what is to be represented by the notation (the *model*). model needs to be mapped to the *view*; therefore, a direct mapping specification between them should be provided. In our visualisation procedure, user defines these mappings by providing model element correspondences to the *view* using annotations (Figure 1.3). This mapping transformation generates the *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*, and hence the visualisation, can be refreshed to reflect the changes. For example if the values of bars in a bar chart are changed refreshing the visualisation will result in running the mapping transformations again and hence updating the bar height. The resulting notation is saved in a notation repository to be (re)used for generating visualisations (Figure 1.4).

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 predefined notations, elements of input files are mapped to elements of the visual notation's *model* (Figure 1.5). 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 visualisations. 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 customised notations) will be saved in a repository (Figure 1.6).

A visualisation is generated by composing customised notations (Figure 1.7). This composition allows a visualisation 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 visualisation file resulting from composition. This visualisation file is then transformed to visualisations using the mapping transformations embedded in notations. For example in Figure 1, applying the transformation on the input file has resulted in the visualisation marked by 8.

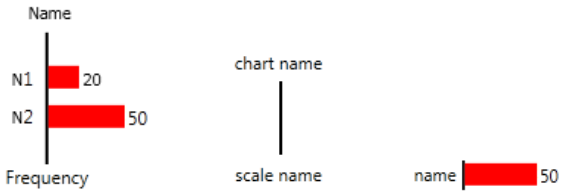
This new visualisation approach has been implemented in CONVErT tool for generating interactive visualisations. CONVErT was initially developed for model transformation using concrete visualisations [8]. The visualisation approach presented in this paper provides better flexibility in domain and types of visualisations that can be supported in model transformation tasks. On the other hand, by-example transformation and interaction mechanisms implemented in CONVErT framework helped us better incorporate user-centric drag and drop approach in the visualisation process. Also, the transformations used to transform input files to visualisation file 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.

USAGE EXAMPLES

This section provides two case studies demonstrating the usage of our approach. The first is a simple example that demonstrates visualising input data as a bar chart. This data can represent various domains from sales records to report generation. Our second case study demonstrates a more complex visualisation of traffic data using 3D shapes and animation.

Case study 1: Generating a reusable bar chart visualisation

Let's assume a bar chart visualisation has been designed by a designer similar to Figure 2(a). Our intention is to reuse this design, generate reusable notations, and use them to visualise an input file representing records of a user study.



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

Figure 2. Bar chart visualisation design.

To generate notations for our bar chart, the design can be split into two *views*. A *view* for chart (Figure 2(b)) and a *view* for bars (Figure 2(c)). Two notations will be generated from these *views*. The various semantic constructs that these notations represent should be provided as a notation's *model*. In a bar chart visualisation similar to Figure 2(a), bars represent values of a certain

category by visually depicting that value in the *view* using their length. Since multiple bars may exist in a bar chart, each bar is also accompanied by a name for the value it represents. Therefore a bar's *model* should specify the value and the name of the bar as in Listing 1. Similarly, since the chart notation needs to provide a name for the chart, and a name representing scale element, it's *model* should provide two names as in Listing 2.

Listing 1. Bar's *model*

```
<HorizontalBar>
  <Value>50</Value>
  <Name>name</Name>
</HorizontalBar>
```

Listing 2. Bar chart's *model*

```
<HorizontalBarchart>
  <Name>chart name</Name>
  <ScaleName>scale name</Scale...
  <Bars>bars</Bars>
</HorizontalBarchart>
```

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 visualised) input data. If colour of the notations was also dependant on values of the input, a colour should also have been specified in bar notation's *model*. Here bar notation uses default red colour provided by the designer. In this paper, we present these *model* elements by XML and *views* are provided via Windows Presentation Foundation (WPF) and eXtensible Application Markup Language (XAML) [9]. As a result, our target visualisations are also XAML and WPF based visualisations. However, the approach can be adopted to use other technologies as well.

Notations can incorporate, or host, other notations (e.g. a bar chart visualisation will incorporate multiple bar visualisations). To clearly define the position which notations are to be placed in a host notation, a placeholder should be provided in the host notation's *model*. For example, in listing 2, a "Bars" element is provided as a placeholder specifying that the bar chart may include multiple bars. These placeholders specify the linking and inclusions of a notation in another.

The mapping between *model* and *view* in our approach is a transformation which transforms the domain values defined in the *model* into the *view*. Any updates to the *model* will be applied to the *view* by the mapping, which itself is created by user-provided annotations in the *view* (See Figure 1.3). These annotations define correspondence relationships between *model* and *view* and include one-to-one, and one-to-many and iterative correspondence relations. To specify these correspondences, a simple annotation scripting language is used in our approach. This consists of a linkto="element" for specifying one-to-one correspondences, and callfor="element" for specifying one-to-many correspondences. For example in the bar chart notation, the name of bar chart and the scale name should be annotated by "linkto" as in Listing 3. The placeholder for bars should be annotated in the *view* where bar notations are to be inserted. In this case, bar notations should be inserted as children of a StackPanel element in the design. As a result, callfor="bars" has been annotated in the StackPanel in Listing 3.

Listing 3. Char notation's view annotation

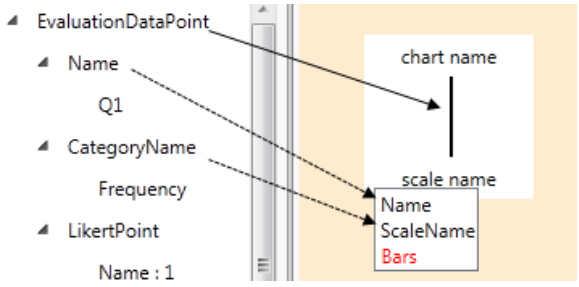
```
<DockPanel xmlns="http://schem..." Background="White">
<TextBlock ... DockPanel.Dock="Top" Margin="5">
<TextBlock.Text linkto="Name">Name</TextBlock.Text>
</TextBlock>
<TextBlock ... DockPanel.Dock="Bottom" Margin="5">
<TextBlock.Text linkto="ScaleName">Scale</TextBlock..>
</TextBlock>
<StackPanel>
<StackPanel Orientation="Horizontal">
<Border BorderBrush="Black" BorderThickness="0,0,2,0">
<Label>
<Label.Background>White</Label.Background>
</Label>
</Border>
<Label Width="50"/>
</StackPanel>
<StackPanel callfor="Bars">
</StackPanel>
</StackPanel>
</DockPane>
```

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 linkto annotations are translated to value fetch scripts and callfor annotations are translated to call for templates. 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 names to their corresponding TextBlocks in the *view*. It will also register a declarative call for templates to be applied on the data provided to the "Bars" element of bar chart notation's *model*.

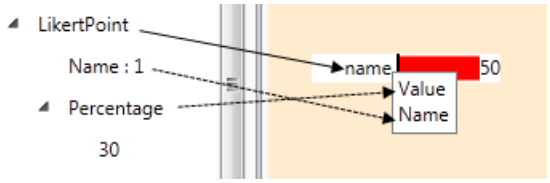
The mapping transformation is designed in a way that can wrap each notation in a predefined interaction logic. This is done in the transformation code generator that translates the annotations. In this example the code generator is configured to wrap notations with drag and drop, and right click event handlers. As a result the notations generated can be dragged and dropped on a canvas or other notations to perform different tasks. Also right clicking on each notation reveals its *model* elements. This interaction can be altered according to application of notations. For example, one could provide events to right click on a notation and receive additional information regarding its data. The events to be performed in this wrapping are implemented in event handling mechanism of WPF. If alternative visualisation mechanisms are being used, the wrapping can be altered to use other event handling mechanisms. 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 [10, 11]).

Step two involves transforming example input data to the specified notational elements. This step is used to provide the system with information in order to create transformation rules for transforming specific parts of input models to the notation's *model* data. Input

data is shown using a tree layout provided by CONVErT framework by default. Elements of input can be dragged and dropped on elements of notation *model* to generate the transformation from input to notation's *model*. 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 visualised represents the values of a Likert based user study. Each Likert point is mapped to a bar with its Name and Percentage representing Name and Value of the bar (see Figure 3(b)). Similarly, "EvaluationDataPoint" element is mapped to bar chart notation with its Name and Category representing Name and ScaleName of the chart (see Figure 3(a)). Note that "Bars" element placeholder is shown with different colour to separate it from other elements of chart notation's *model*.



(a) Mapping EvaluationDataPoint to chart notation.



(b) Mapping LikertPoint elements to bar notation.

Figure 3. Mapping input elements to notations.

The defined customised notations are saved in repository. These customised 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 customised notations for different inputs. Here for example, a transformation rule will be generated to transform each "LikertPoint" element to a bar's *model*, and a transformation for transforming "EvaluationDataPoint" to chart's *model*.

Once input data to notation transformation is complete, the defined customised notations need to be composed to generate a meta-model for the visualisation file and a complete input to visualisation file transformation. An example of composing chart and bar notations to generate bar chart visualisation is provided in Figure 4. Linking a notation to a placeholder in 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 placeholder element of host notation. Since each customised 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 inside 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 customised 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.

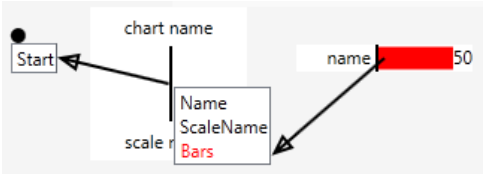


Figure 4. Composing notations to create a visualisation for a bar chart. Arrows are provided by framework to trace notation composition.

Linking a notation to a *start* element will define the parent 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 input file to visualisation file's transformation code from the rule linked to this *start* element. For example in figure 4, the transformation rule associated to chart customised notation (transforming EvaluationDataPoint 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 LikertPoint elements to bar notation Models. Meta-models resulting from composition are also used for model validation purposes. The resulting visualisation file of this composition is provided in Listing 4. 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).

Listing 4. Resulting visualisation file of the composition in Figure 4

```
<HorizontalBarchart>
  <Name>Q1</Name>
  <ScaleName>Frequency</ScaleName>
  <Bars>
    <HorizontalBar>
      <Value>30</Value>
      <Name>1</Name>
    </HorizontalBar>
    ...
  </Bars>
</HorizontalBarchart>
```

To render visual elements the visualisation file needs to be transformed into a renderable visualisation. This rendering in our approach reuses the *model* to *view* mapping transformation of notations available in the notation repository. When a visualisation file is to be rendered (as visualisation), it is checked against the

mapping transformations of the notations in the notation repository. A visitor pattern is used to search the visualisation file for constructs similar to *model* part of notations where those *model* to *view* mapping transformations could be applied. The mapping transformations of the matching notations will be returned by this visitor. From these retrieved transformations, a transformation script is generated to transform the visualisation file into renderable visualisations (in this case XAML graphics). This transformation script is executed using the transformation engine of CONVErT and the resulting XAML code will be rendered in the framework or can be exported as browser based visualisation. For example, the resulting bar chart visualisation of the visualisation file in Listing 4 is depicted on Figure 5.

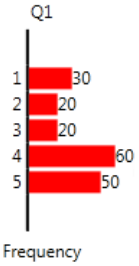


Figure 5. Resulting bar chart visualisation.

As stated above, user can interact with the notations of this visualisation. For example, they can drag each bar into other notations or other canvases. The next section provides a more complex example using 3D visualisations.

Case study 2: Visualising traffic data

Traffic congestion is an ongoing issue for modern cities around the globe 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 visualisation 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 visualisation will help to better monitor traffic volume and congestion.

Similar to our first case study, we assume a designer has generated a visualisation of Melbourne CBD as in Figure 6(b). This visualisation is inspired by the 3D visualisation of USA's population over time introduced by Petzold [12] (See Figure 6(a)). It demonstrates population of each point of interest on a map using 3D bars. A slider is provided to navigate the visualisation to depict data for different years. The 3D visualisation 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 visualisation for generating a time-lapse of traffic volume in Melbourne CBD for four intersections.

To generate the required notations from the provided visualisation, 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 Figure 7. The combination of these three views is used to generate the complete visualisation.

(a) Map host. (b) Map. (c) 3D bar.

Figure 7. Traffic visualisation notation *views*.

Given the *views* in Figure 7, the required *model* data to be represented by them is provided in Listings 5 to 7. Map host of Figure 7(a) provides a description of the visualisation and a horizontally laid out list that embeds visual elements 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's *model* will then have to include multiple other notations (in this case maps). Therefore the map host notation's *model* in Listing 5 provides a placeholder for visuals and a description. Map notation provides a description for the map and will include multiple bars depending on the provided data. As a result its *model* should include the description and a placeholder for bars. Listing 7 provides map notation's *model*. 3D bars to be laid out on the map take a value to be represented by their height, a colour, and longitude and latitude to specify their position on the map. 3D bar notation's *model* is provided in Listing 6.

Listing 5. Host's *model*

```
<VisualizationHost>
 <Description>desc</De...
 <Visuals>maps</Visuals>
</VisualizationHost>
```

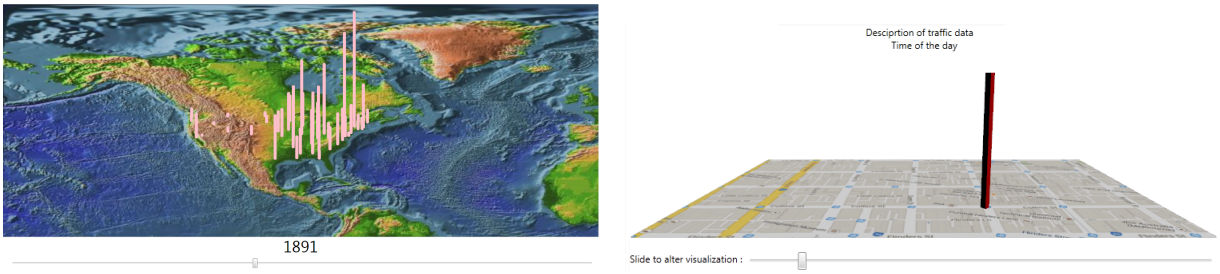
Listing 6. 3D Bar's *model*

```
<CubeData>
 <Colour>Blue</Colour>
 <Longitude>−37.814790</Lon...
 <Latitude>144.969018</Latit...
 <Value>10</Value>
</CubeData>
```

Listing 7. Map's *model*

```
<MapData>
 <MapDescription>Desc</MapDescription>
 <TDBars>bars</TDBars>
</MapData>
```

To link these Models to their *view*, correspondence links between elements of *view* and *model* should be annotated in the *view*. In this example, elements of the bar's *model* are in one to one relationship with their corresponding *view* elements. As a result they should be annotated by linkto annotations (not provided here due to space limitation). A map may include multiple bars, as a result, callfor annotation should be provided in its *view* to reflect the one to many relationship be-



(a) Visualisation of USA population.

(b) Traffic visualisation provided by a designer

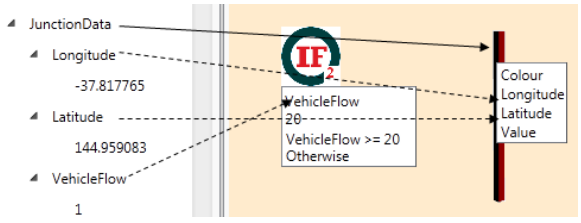
Figure 6. Samples of 3D visualisations.

tween the map and the bars. Map description, should be viewed on top of the map, therefore the linkto annotation is provided to link the description provided by map *model* to the text block representing the description on top of the map. Listing 8 presents annotated *view* of our map notation. Similar to the map notation, map host notation includes multiple maps. It therefore represents one to many relationship between the host and multiple maps ("visuals") to be inserted inside it using a callfor annotation. It also has a description that allows the visualisation description to be displayed on top, this description should be provided by linkto annotation similar to Map notation.

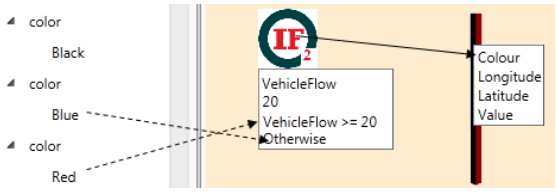
Listing 8. Map's annotated *view* (Some details have been omitted to save space)

```
<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>
        <PerspectiveCamera FarPlaneDistance="200".../>
      </Viewport3D.Camera>
      <ModelVisual3D>
        <ModelVisual3D.Content>
          <DirectionalLight Color="White" Direction="..."/>
        </ModelVisual3D.Content>
      </ModelVisual3D>
      <Viewport2DVisual3D>
        <Viewport2DVisual3D.Material>
          <DiffuseMaterial>
            <DiffuseMaterial.Brush>
              <ImageBrush>
                <ImageBrush.ImageSource>MelbBing.bmp</I...>
              </ImageBrush>
            </DiffuseMaterial.Brush>
          </DiffuseMaterial>
        </Viewport2DVisual3D.Material>
        <Viewport2DVisual3D.Geometry>
          <MeshGeometry3D Positions="-55,0,-30 -55,0,..."/>
        </Viewport2DVisual3D.Geometry>
      </Viewport2DVisual3D>
    </Viewport3D>
    <Canvas callfor="TDBars">
    </Canvas>
  </Canvas>
</DockPanel>
```

Using these annotations, the required *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, we can map these data records to our recently generated notations by drag and dropping each element on corresponding notation element. In this example, we map each junction data to a bar representing its traffic volume as in Figure 8. The colour 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 colours are generated using the transformation conditions available in CONVErT framework. These conditions can be customised 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 generated, please refer to [13]).



(a) Mapping elements to condition and notation.



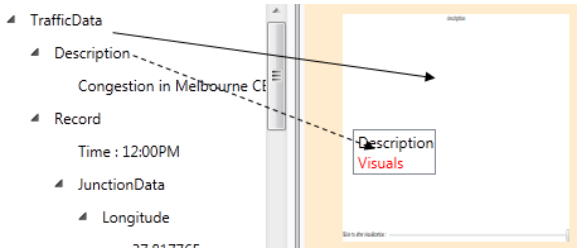
(b) Mapping colours to condition.

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

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 Figure 9(a). The traffic data element will be mapped to our Map host notation where its description if linked to host's description (see figure 9(b)).

Saving the above specified mapping will result in gen-

(a) Mapping Record element to Map notation.



(b) Mapping TrafficData element to Map host notation.

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

eration of customised notations. Similar to our first case study, these notations can be composed to generate the meta-model for visualisation file according to their placeholders (see Figure 10). This composition results in a transformation from the input file to the visualisation file confirming to the composition's meta-model.

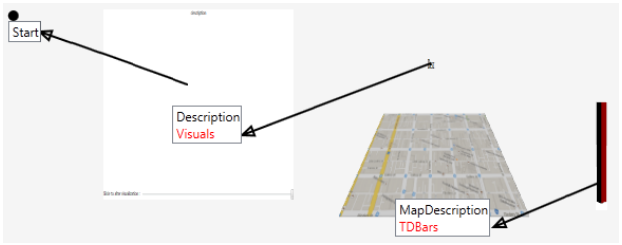


Figure 10. Composing notations to create a 3D map visualisation.

The visualisation file resulting from this transformation will be transformed to a renderable visualisation by reusing the *model* to *view* mapping transformations available in the notations of notation repository. The result of rendering the generated visualisation file is presented in Figure 11. Sliding the slider will result in an animated visualisation of how traffic volume changes over time. In Figure 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 visualisations.This is due to the fact that transformation rules for performing the input to visualisation procedure are already defined.

EVALUATION

The evaluation of our approach is divided to two parts. First, capability testing using the case study visualisations to show the effectiveness and applicability of the

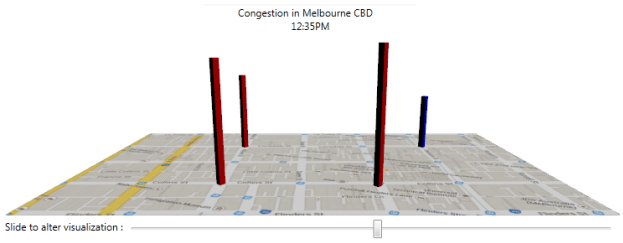


Figure 11. Resulting visualisation of traffic data.

approach. Second, a user study to test usefulness of the approach and to examine how users react to such by-example drag and drop visualisation. This section provides the details of our user study.

For this study, 19 users were recruited (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 participants which described CONVErT framework's user interface and the visualisation generation procedure. Participants were then asked to perform a set of given visualisation tasks following think-aloud approach. The experiment setup comprised a laptop with an attached mouse. Screen captures were taken during the process and a matching questionnaire with 21 visualisation 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 visualisation with CONVErT. Both groups had the same settings but used different input models and visualisations. Our rationale for this was to test research questions 1, 2 and 5.

The first group were given an input file representing business sales data and were asked to create a bar chart visualisation of their sales data. The second group were given a class diagram data (XML) and asked to generate a class diagram visualisation. Task description hard copies which were handed to the participants did not describe instructional steps. Instead, they included the input file names and their locations, and a snapshot of the desired final visualisation 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. Our rationale for this was to test research questions 3 and 4.

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 questions **D.4**:"How familiar are you with data visualisation?",

the participants had following options: **VF**: Very familiar, **SF**: Somewhat familiar, **HH**: Had heard of it, and **NF**: Not familiar. The frequency of responses are provided in Table 1.

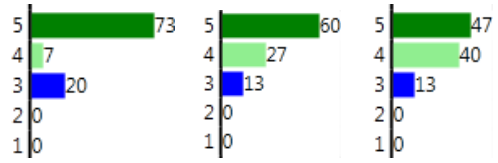
Table 1. Partial participant demographics.

Question	**VF**	**SF**	**HH**	**NF**
D.4	2	9	2	2

Table 2. 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 visualisation activities.

Table 2 demonstrates a selection of three key 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 summarised in Figure 12. Full results can be found in CONVErT's website[1].Note that the visualisations of Figure 12 are generated with a modified version of bar chart visualisation in our first case study.



(a) Q1. (b) Q2. (c) Q3.

Figure 12. Frequency (%) of answers to questions in Table 2.

As the user study demonstrates, users of both groups positively liked the visualisation 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 visualisation, 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 representations to understand the elements of visual *views*.

DISCUSSION

The two very different case studies provided above demonstrated how a separately designed visualisation can be reused to generate reusable visual notations. Although the visualisations used XAML and WPF as proof of concept technologies, we believe the approach presented here can be used with other technologies with minor alterations. And hence, they address our first key contribution. The case studies also partially address our second to forth contributions on effectiveness of our visualisation approach with regards to mapping input data to the designed notations, composing the customised notations and generating concrete visualisations. These contributions are further investigated with our user study which tries to examine user experience with the new visualisation approach.

Considering that the mapping between input data and visual notations are performed using rule based model transformations, any data could have been used for input to visual notation transformation step. For example, the traffic data of second case study could have been mapped to our bar chart visualisation of case study one to form visual analytics charts. This addresses our fifth contribution on reusing already defined visual notations for generating visualisations for different input models.

There are certain threats to validity of our user study results with regards to participant number and affiliation. Our participants were mostly recruited from staff and students of Swinburne university and therefore their affiliation might have introduced bias in their responses to the questionnaire. As the demographics of Table 1 demonstrates, our users were mostly familiar with at least one visualisation technique (11 out of 15). This could have affected the results of our visualisation evaluation. Also, with current number of participants, statistically significant and more generalisable inferences cannot be made. Hence, this user study is a work in progress and we will update our online results as we recruit more participants.

CONCLUSIONS AND FUTURE WORK

We have presented a new approach for generating visual notations and forming concrete visualisations. 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 visualisation and a transformation from an input file to visualisation is generated. This approach has been implemented in our CONVErT framework and applied on multiple visualisation 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 visualisation 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 MVC. For example in our second case study visualistion, processing of longitude and latitude of bars is done in their MVVM in a way that each bar checks the registered

coordinates of its parent (the map) and positions itself accordingly. Similarly, other external technolgies can be integrated into notations as an extenssion. For example live refreshment of visualisations or external layout mechanism [14].

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 visualisations. An example is where a pie chart has been visualised representing percentage of people who voted for certain product. By clicking on a pie piece in this visualisation, it would be possible to show what percentage of them were male and what percentage were female.

It is common in visualisation community to assume the data to be visualised is pristine and satisfies certain formatting and quality required [15]. However, it is not often the case and our visualisation 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 visualisation. Similar functionality can however be provided using the transformation facilities within current version of the CONVErT framework.

Acknowledgment

This work is partially supported by an ARC Discovery Project and ARC Future Fellowship. 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 Transactions on Software Engineering*, vol. 35, no. 6, pp. 756–779, 2009.
2. M. C. Humphrey, "Creating reusable visualizations with the relational visualization notation," in *Proceedings of the Conference on Visualization '00*, ser. VIS '00. Los Alamitos, CA, USA: IEEE Computer Society Press, 2000, pp. 53–60.
3. S. Fenwick, J. Hosking, and M. Warwick, "A Visualisation System for Object-Oriented Programs," in *Technology of object-oriented languages and systems TOOLS 15*, C. Mingins and B. Meyer, Eds. Sydney, Australia: Prentice Hall, 1994, pp. 93–103.
4. J. Hosking, S. Fenwick, W. Mugridge, and J. Grundy, "Cover yourself with Skin," Software Verification Research Centre Department of Computer Science The University of Queensland, Queensland 4072 Australia, Tech. Rep. 94, 1994.
5. 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, Munich, Tech. Rep., 2006.
6. J. de Lara and H. Vangheluwe, "Defining visual notations and their manipulation through meta-modelling and graph transformation," *Journal of Visual Languages and Computing*, vol. 15, no. 34, pp. 309 – 330, 2004, domain-Specific Modeling with Visual Languages.
7. G. Costagliola, V. Deufemia, and G. Polese, "A framework for modeling and implementing visual notations with applications to software engineering," *ACM Transactions on Software Engineering and Methodology*, vol. 13, no. 4, pp. 431–487, Oct. 2004.
8. I. Avazpour and J. Grundy, "CONVErT: A framework for complex model visualisation and transformation," in *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL-HCC)*, 2012, pp. 237–238.
9. L. MacVittie, *XAML in a Nutshell*. O'Reilly Media, Inc., 2006.
10. I. Avazpour, J. Grundy, and L. Grunske, "Tool support for automatic model transformation specification using concrete visualisations," in *2013 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 718–721.
11. I. Avazpour and J. Grundy, "Using concrete visual notations as first class citizens for model transformation specification," in *2013 IEEE Symposium on Visual Languages and Human-Centric Computing (VL-HCC)*, Sept 2013, pp. 87–90.
12. C. Petzold, *3D Programming for Windows®*. O'Reilly, 2010.
13. I. Avazpour, "Towards user-centric concrete model transformation," Ph.D. dissertation, Swinburne University of Technology, 2014.
14. I. Avazpour, U. Rüegg, and J. Grundy, "CONVErT meets KIELER: Integrating advanced layout algorithms into by-example visualisations," in *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL-HCC)*, 2014.
15. 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*, vol. 10, no. 4, pp. 271–288, Oct. 2011.