uVis Studio: An Integrated Development Environment for Visualization

Kostas Pantazos, Mohammad A. Kuhail, Soren Lauesen, and Shangjin Xu

IT University Of Copenhagen, Rued Langgaards Vej 7, Copenhagen, Denmark

ABSTRACT

A toolkit facilitates the visualization development process. The process can be further enhanced by integrating the toolkits in development environments. This paper describes how the uVis toolkit, a formula-based visualization toolkit, has been extended with a development environment, called uVis Studio. Instead of programming, developers apply a *Drag-Drop-Set-View-Interact* approach. Developers bind controls to data, and the Studio gives immediate visual feedback in the *Design Panel*. This is a novel feature, called *What-You-Bind-Is-What-You-Get*. The Studio also provides *Modes* that allow developers to interact and view the visualization from the end-user's perspective without switching workspace, and *Auto-Completion*; a feature of the *Property Grid* that provides suggestions not only for the formula language syntax but also for the tables, the table fields and the relationships in the database.

We conducted a usability study with six developers to evaluate if the Studio and its features enhance cognition and facilitate the visualization development. The results show that developers appreciated the *Drag-Drop-Set-View-Interact* approach, the *What-You-Bind-Is-What-You-Get*, the *Auto-Completion* and the *Modes*. Several usability problems were identified, and some suggestions for improvement include: new panels, better presentation of the *Modes*, and better error messages.

Keywords: Information Visualization, Integrated Development Environment, Visualization Development, What-You-Bind-Is-What-You-Get

1. INTRODUCTION

In the last two decades, several Information Visualization toolkits (InfoVis) have been developed to help developers construct visualizations in a programmatic way. Some of these toolkits are: Prefuse,[1] InfoVis Toolkit,[2] Protovis,[3] Piccolo,[4] D3,[5] etc. These toolkits provide frameworks and libraries for constructing state-of-the-art visualizations by writing code. Supporting developers to use InfoVis libraries is a success, but a development environment that provides immediate visual output during the construction phase can improve the development process. Further, it has a potential of engaging users with less IT skills in the visualization development process.

A development environment assists users in several ways: First, humans perform better with visual than textual representations.[6] Therefore, interacting with visual objects should be easier than coding. Second, in complex cases it is not easy to address the semantic gap between the written code and the visual outcome,[7] until you have executed the code. Third, visualization development is an iterative process, and immediately viewing the changes in the visualization (e.g. color selection, position) speeds up the process.

InfoVis toolkits[1,2,4,5] do not have a development environment where developers can directly interact with visual objects (building blocks) and get immediate feedback as they bind controls to data. These toolkits can be integrated in development environments (e.g. Eclipse, Visual Studio), but still developers have to use the code editor. Relating the code to the visual objects, and code to data can be challenging even for experienced developers. For example, it may be difficult to answer questions like "How can I create a shape? What does the shape represent? What does the shape's height, width, color, etc., mean?". These questions exist because of the gulf of execution (*How do I do something?*) and evaluation (*What happened?*) identified by Norman.[8] In the visualization area, developers must handle these two gulfs to construct visualizations.

Further author information: E-mail: [kopa, slauesen, moak, xush]@itu.dk

We developed uVis Studio (or simply the Studio), the development environment for uVis toolkit.[9,10] uVis Studio consists of 7 panels: *the Toolbox*, *the Solution Explorer*, *the Property Grid*, *the Modes*, *the Error List*, *the Entity-Relationship (E/R) Model*, and *the Design Panel*. It resembles a traditional development environment, but it has three novel features to enhance visualization development: *What-You-Bind-Is-What-You-Get* (*WYBIWYG*), *Modes* and *Auto-Completion*.

Visualizations are constructed in a *Drag-Drop-Set-View-Interact* approach. Developers *drag* and *drop* visual objects in the *Design Panel*, *set* properties in the *Property Grid*, immediately *view* how objects are bound to data in the *Design Panel*, and *interact* with the visualization as end-users without switching workspace.

The remainder of this paper is organized as follows. Section 2 provides a summary of the related work. Section 3 briefly describes the uVis toolkit, and in section 4 we present the uVis Studio. Section 5 presents a custom visualization developed with uVis Studio. The results of the usability study with six participants are reported in section 6. Finally, section 7 concludes with recommendations regarding the construction of development environments for visualizations and our future work.

2. RELATED WORK

2.1 Visualization Tools and Toolkits

Early 1990's, Roth and Mattis[11] presented SAGE, "an intelligent system which assumes presentation responsibilities for other systems by automatically creating graphical displays which presents the results they generate". This tool uses graphical techniques to express the application data characteristics and meet the presentation needs. SAGE creates a presentation in three steps: select, refine, and integrate the graphical technique taking into consideration the constraint of data, goal characteristics, and the properties of the graphical technique. This means that SAGE automatically suggests effective presentations to users, and allow them to refine it by specifying constraints on the layout. Other important studies, similar to SAGE,[11] are presented by Mackinlay[12] and Casner.[13] In contrast, users of the Studio define and customize how the presentation should look. Therefore, to create a visualization, they drag and drop visual objects, map data to objects, and immediately view the visualization.

SageBook[14] allows users to sketch, search and customize visualizations. The user creates a sketch using SageBrush,[15] and SageBook searches for suitable controls (data-graphics) in the SageBook's library. The results (one or more data-graphics) are shown in the SageBook browser. Users can select one data-graphic and modify it in SageBrush, but the visual objects are not bound to data. Further, SageBook limits users to construct visualizations that the data match a data-graphic in the library.

In the last decade, several visualization toolkits have been developed to support developers construct state-of-the-art visualizations. In Prefuse[1] developers use Java to develop visualizations using a set of fine-grained building blocks and specifying operators that define the layout and behavior of these blocks. The InfoVis Toolkit[2] is another Java-based visualization toolkit that utilizes a number of interactive building blocks to construct visualizations. In Piccolo[4] developers code in C# or Java and develop rich user interfaces. D3[5] is a web based visualization toolkit. Visualizations are constructed using SVG, HTML 5 and CSS. All these toolkits can be imported in an integrated development environments (IDEs), but developers create visualizations with the code editor. None of the aforementioned tools allow developers to construct visualizations by manipulating visual objects and immediately viewing the visualization without switching workspace.

Improvise[16] is a visualization toolkit for creating multi-view coordination visualizations for relational data. Developers create visualizations using expressions for simple shared-object coordination mechanism. Shared-objects in Improvise are graphical attributes such as color, font, etc. Improvise has a specialized development environment where developers create views by adding frames, controls, defining variables and attaching data using the lexicon work area (a central repository where information related to the data and database are saved). In Improvise, visualizations are created using the four editors and developers applies a step-by-step approach. Another visualization tool is Processing.[17] It has a development environment similar to a traditional IDE. Developers create visualization by writing code (Processing) in the code editor and view the visualization in a new window after having executed the code.

Google Chart Tools Library[18] is written in JavaScript and provides several predefined simple charts (e.g. line chart, scatter chart, etc.) and advanced chart types (e.g. Image multi-color bar chart, Motion Chart Time Formats, etc.). Developers can use the web-based development environment named Code Playground.[19] It consists of three panels: API, Code Editor and Output. Developers select the visualization type from the *API* panel, and the corresponding JavaScript code is automatically shown in the *Code Editor* where they can edit it. The *Output* panel shows the results when developers run the code. This development environment also supports debugging the code. Further, Google Chart Tools has the Live Chart Playground[20] to test charts already created in the Code Playground. In the Live Chart Playground, developers can change some parameters and see how the visualization changes. ProtoViewer[21] is similar to Code Playground of Google Visualization API, where developers can write the code, execute it and view the results. It uses Protovis,[3] the predecessor of D3.[5] The screen is divided into three panels: *Data*, *Design* and *Code*. Developers use the *Data* panel to choose a dataset from the *Data* panel and select a visualization. The code is automatically shown in the *Code* panel. Then, they execute the code to view the results in the *Design* panel. In Code Playground and ProtoViewer, developers write code to specify controls.

Industry has developed tools to help non-developers create visualizations. Some of them are: Spotfire,[22] Tableau,[23] Omniscope,[24] etc. All these tools come with a development environment, where users select the data, the visualization type, and then the visualization is automatically shown. These tools provide excellent interaction technologies, such as linking and brushing, zooming, sorting, searching etc. Users have to use the predefined chart-types supported by this tools. This means that creating new visualizations and implementing new interactions with these tools is not feasible as they are "black-box" systems.

2.2 Design Approaches

According to the reference model of information visualization (described in[25,26]), the development process consists of three steps: *Data Transformation*, *Visual Mapping*, and *View Transformation*. Card et al.[26] say: "The core of the reference model is the mapping of data table to visual structures". Therefore, we briefly summarize how the aforementioned tools and toolkits follow the reference model, and support the core principle.

In some tools,[11,12,22–24] developers construct visualization by selecting a dataset and a predefined visualization type. Then, the visualization is automatically created and shown to the user. This approach enhances accessibility (Do I know how?) and efficiency (How much time will it take?), but it affects expressiveness (Can I build it?) as users are limited to use only the supported visualization techniques. Further, the mapping of data to visual structures is automatically handled by the system.

Other tools[1,2,4,5] are more expressive than the previous ones and used by developers. Developers construct custom visualizations by writing code. However, mapping data to visual structures using code requires additional cognitive effort, which has an impact on efficiency. Also, it affects accessibility as good programming skills are required.

Finally, tools such as Protovis[3] and Improvise[16] attempts to balance expressiveness, efficiency and accessibility in different ways. Users of Protovis write simple specifications to construct visualizations. This attempts to achieve a good level of expressiveness and efficiency. Recently, Protovis was integrated in the ProtoViewer[21] to improve accessibility. However, users of ProtoViewer have to use the predefined visualizations suggested from the tool, or write Protovis specifications in the code editor. The mapping of data to visual objects is achieved through specifications, which requires additional cognitive effort. Improvise addresses expressiveness, efficiency and accessibility by supporting developers with a development environment. Users of Improvise use predefined controls and specify expressions to build coordinated visualization applying a step-by-step approach through several editors.

Our study[27] compared 13 InfoVis tools and toolkits from a user perspective, and also investigated the development environments of these tools. This study showed that developers develop custom visualizations by writing code, and lack external cognitive aids such as IDEs. As Norman[8] says, "the real power come from devising external aids that enhance cognitive abilities."

3. DRAG-DROP-SET-VIEW-INTERACT APPROACH

In the software engineering area, developers create user interfaces in IDEs such as Eclipse, Netbeans, Visual Studio, etc. Developers apply a *drag-drop-set-run* approach where they drag and drop controls, set properties (sometimes write code in the code editor), and run the program to view the results. These IDEs provide cognitive support such as auto-completion and immediate feedback. However, the feedback is static as controls in the *Design Panel* are not bound to data, and developers have to execute the program to view the data and interact with the visualization. Moreover, the auto-completion gives suggestions only for available variables and functions of a class.

As IDEs have proven to be a successful approach in constructing user interfaces,[28] we focused on applying the same principles in the InfoVis field. However, the existing cognitive support should be improved and more data-oriented as visualizations are all about the data. Therefore, we developed uVis Studio that has three novel features (described in detail in section 5).

1. **What-You-Bind-Is-What-You-Get (WYBIWYG)**: This feature of the *Design Panel* allows developers to obtain immediate feedback of controls bound to data without switching screens. Binding visual controls to data is one of the steps of the visualization development process,[26] and is not intuitive. *WYBIWYG* attempts to address it, and aims at improving efficiency and correctness during the construction process. Also, it may lead to the discovery of novel presentations because of the immediate feedback on the screen.
2. **Modes**: It consists of the *InteractionView* and the *DataView* mode. The *InteractionView* mode supports developers to try end-users interactions without switching workspace. As a result, it removes the redundant step where developers execute the code and wait for the results. Developers use the *DataView* mode to disable the *WYBIWYG* feature in case they feel overwhelmed by the data in the *Design Panel*.
3. **Auto-Completion**: This feature of the *Property Grid* helps developers to recognize rather than remember the syntax of the formula language, and write them correctly. In traditional IDEs, the auto-completion gives suggestions for available variables and functions. In addition, the *Auto-Completion* of uVis Studio provides suggestions for tables, table's fields and relationships in the database.

In uVis Studio, developers apply a *Drag-Drop-Set-View-Interact* approach to create visualizations. Below, we present the steps of this approach:

- Developers *drag* and *drop* controls in the *Design Panel*. To create controls (e.g. triangle, rectangle, etc.), drag and drop is a straight forward action and more obvious than writing code.
- For each control, developers *set* controls' properties. Developers change a property (e.g. `DataSource`, `Top`, `Left`, etc.) from the *Property Grid*. The *Auto-Completion* feature helps them in writing formulas.
- Developers *view* immediate visual results in the *Design Panel*, and if needed adjust control properties again. Developers do not need to switch workspace to view what happened as they change the properties. They can view the mapping of the data to controls in the *Design Panel* and observe the properties in the *Property Grid*.
- Developers *interact* with the visualization using the *Modes*. The *Design Panel* shows the running version of the visualization, and developers can use the *InteractView* mode to *interact* with the visualization as end-users and see what happens.

4. UVIS TOOLKIT

uVis is a formula based visualization toolkit that allows developers to construct visualization of relational data.[9,10] uVis supports construction of visualizations by means of primitive building blocks (controls) and spreadsheet-like formulas (Figure 1). However, the spreadsheet-formulas are more advanced than the traditional ones, when it comes to data accessing. uVis supports developers with simple and advanced formulas since it provides abstractions for data transformation. The following subsections briefly describe two key principles of uVis: controls and formulas. A more thorough description of the uVis formula language can be found in.[9,10]

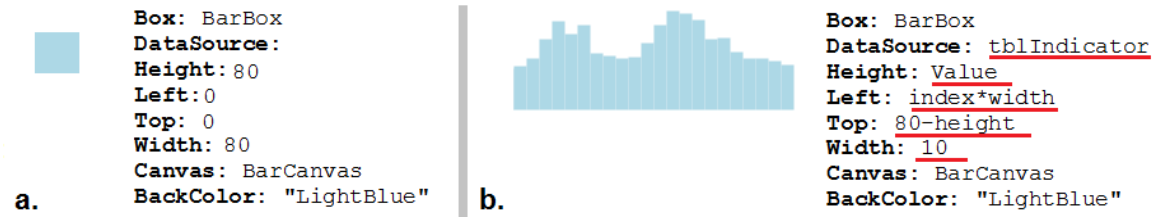

Figure 1. (a) A box control with its default values. (b) A a bar chart that uses a box control. The developer specifies the *DataSource* property, uVis connects to the database, and generates a control for each row in the table *tblIndicator*. The *Height* is computed as the table field *Value*. Left uses the keyword *Index* (it corresponds to the row index) and the control's `Width`.

4.1 Controls

Controls are the building blocks of a visualization. A control can be bound to a table and uVis kernel generates a control instance for each row in the table. A control has 3 types of properties (uVis, designer and event properties) which are specified by formulas. Formulas are used to bind controls to data, set the appearance and specify the behavior of controls.

4.2 Formulas

The formula is the most important concept in uVis toolkit. To build a visualization you need the controls, but formulas are the ones responsible for visually mapping these controls. A control property can have a formula that specifies how the property value is computed. An example of uVis formulas is presented in Figure 1. uVis formulas can refer to controls, controls properties, tables and table fields in the database.

In other programming languages, or domain specific languages, the sequence of declarations is crucial. uVis takes away the burden of determining the sequence. Developers can specify formulas of a control in a free sequence. However, when a cyclic reference exists (similar case as in spreadsheets), uVis detects it and notifies developers.

5. UVIS STUDIO

uVis Studio (Figure 2) extends the uVis toolkit with an integrated development environment that is written in C#, and all panels are Windows Presentation Foundation(WPF) windows. All panels can be arranged by developers; they can resize, dock, hide, and open them as separate windows. Developers construct visualizations applying the *Drag-Drop-Set-View-Interact* approach. Figure 2 shows the Studio and a custom visualization similar to the Lifelines.[29] In this case, the LifeLines shows the medical orders (white boxes aligned to the timescale) and intakes (colored bars inside the white boxes) for patient Lise B. Hansen. To create this visualization, developers use 5 types of controls: the label, the textbox, the button, the timescale, the panel, and the box. They drag and drop controls in the *Design Panel*, set the formulas for each control in the *Property Grid*, view immediate results as the properties are changed, and interact as end-users to see how the visualization behaves.

5.1 The Toolbox

The *Toolbox* (Figure 2.1) contains all controls that uVis toolkit supports. Developers drag and drop controls into the *Design Panel* and the Studio set the default property values.

5.2 The Explorer

The *Explorer* (Figure 2.5) lists the visualization file (*Vis*) that contains the formulas for each control in the visualization, and the visualization mapping file (*Vism*) that contains the information related to the database. The *Vis* file is created as developers use the Studio. While the *Vism* file is written by a developer or a database expert who can specify the database connection, and table relationships. Developers click on the *Vism* file and the visualization opens in the *Design Panel*.

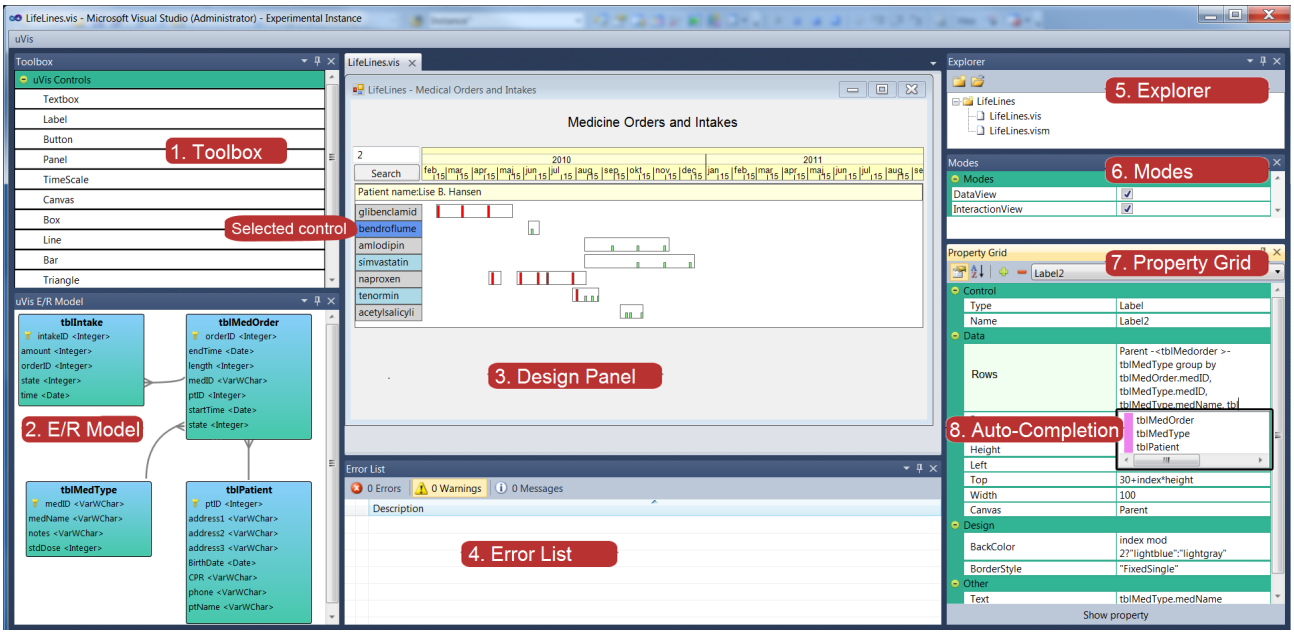

Figure 2. uVis Studio[30] consists of seven panels: 1. Toolbox, 2. E/R Model, 3. Design Panel, 4. Error List, 5. Explorer, 6. Modes and 7. Property Grid. The *Design Panel* shows a simple version of LifeLines created in uVis Studio. Notice that the selected control is highlighted in dark-blue, and properties are shown in the *Property Grid*. The developer is specifying the `DataSource` of the selected control. As the developer is specifying the fields in the *group by* clause, the *Auto-Completion* (8) does not suggest `tblIntake` because the developer is using only `tblPatient` (represented by `Parent`), `tblMedOrder` and `tblMedType`.

5.3 The Design Panel

Visualizations are shown in the *Design Panel* (Figure 2.3). The *Design Panel* addresses the gulf of execution, as developers can directly manipulate controls and obtain immediate feedback as they change control properties. Developers drag and drop, move, and resize controls. However, developers cannot move a control using the mouse when the position (`Left` or `Top`) is bound to data using a formula. For example, in Figure 2 the `Top` property of the selected control is set to a formula. In this case, developers can only change the position of the control from the *Property Grid*.

The *Design Panel* is coordinated with the *Property Grid*. Whenever developers select a control, it becomes highlighted in dark-blue color and the corresponding formulas are shown in the *Property Grid*. When developers interact with the Studio, the visualization formulas are updated. For example, when a new control is added in the form, then formulas with default values, which represent a control, are auto-generated.

5.3.1 What-You-Bind-Is-What-You-Get

The *Design Panel* is a "live" one. It supports direct manipulation and provides continuous feedback during development. When developers change a formula, the *Design Panel* updates immediately. Developers do not need to execute the program to view controls bound to data. We call this feature *What-You-Bind-Is-What-You-Get*. For example, developers bind a box control to table *tblIndicator* that has 22 rows (by specifying a formula, as shown in Figure 1, for the `DataSource` property in the *Property Grid*). uVis kernel creates 22 bars automatically, and the *Design Panel* shows them. Also, developers can map a field of a table to the `Text` property of a label, and the value of the field is immediately shown in the label.

5.4 The Property Grid

The *Property Grid* (Figure 2.7) is the area where developers specify the uVis formulas for control properties. A row in the *Property Grid* consists of the *property-name* and the *formula*. Whenever a control in the *Design*

Panel is selected, the *Property Grid* shows the properties, and their corresponding formulas. Properties can be sorted alphabetically, or shown in groups (e.g. layout properties) to allow developers find them faster. Changes in the *Property Grid* are immediately reflected in the *Design Panel*. Also, changes in the *Design Panel* (e.g moving a control) automatically update the corresponding properties in the *Property Grid*. Further, developers can add and remove properties, by clicking on the + and - button respectively.

5.4.1 Auto-Completion

Writing formulas can be challenging, as developers must remember the syntax, and it is common to misspell words. To help them, the *Property Grid* has an *Auto-Completion* feature that suggests what can follow. The suggestions can be available variables and functions of the formula language, but also suggestions for tables, table fields and relationships in the database. Figure 2.8 shows an example.

The *Auto-Completion* in the *Property Grid* aims at reducing typing errors, misunderstandings, and provide suggestions of what can follow. This information is extracted from the uVis kernel, which uses several path states defined for the formula language. Let us assume that a developer starts typing. The uVis kernel examines what has been typed, and returns a path state. This path state is used from the *Auto-Completion* algorithm in the Studio to show suggestions that the developer can use at this point. For example, after a control name (`lblPatient`), a property (`Text`) may follow, but not a field name. After a table name (`tblPatient`) a field name (`ptName`) may follow, but not a property name. Inspired by the auto-completion feature of Visual Studio, the items in the list are sorted and grouped in categories based on developer's input, and color-coding icons are used to distinguish easier suggestions.

5.5 The E/R Model

The uVis kernel extracts the tables, the fields and the relationship names from the database. The Studio uses this information and visually presents it as an E/R diagram (Figure 2.2). Each entity shows information regarding the table, the fields, and the field types. Developers are able to drag the entities using the mouse. As the database may contain many tables, developers can detach the *E/R Model* panel from the Studio, and obtain a better overview by enlarging the window.

5.6 The Error List

The *Error List* panel (Figure 2.4) shows a list of possible errors whenever a formula is wrong (e.g. developers misspelled a word, misplaced an operator, etc.). After developers have changed a formula in the *Property Grid* (pressed `ENTER` or lost focus to confirm the change), the uVis kernel compiles the formula and reports to the Studio the errors. Whenever an error is shown, developers can select an error by double-clicking, and the wrong formula in the *Property Grid* is highlighted in red. Once developers correct the formula, the *Error List* is updated.

5.7 The Modes

The *Modes* panel (Figure 2.6) aims at making the development process faster by means of the *InteractionView* and *DataView* mode.

1. The *InteractionView* mode allows developers to try end-users interactions without any need for running the visualization. In the *Design Panel* developers can interact as a designer (e.g. move a button) with the controls when the *InteractionView* is active. Otherwise, developers are in end-user's mode. Enabling the *InteractionView* might be considered as the *Run* button in traditional development environments. However in these environments, the developer has to wait for the result, which is shown in a different workspace. For example in the LifeLines (Figure 2), developers can zoom in and out using the timescale.

2. The *DataView* mode allows developers to enable or disable the *WYBIWYG* feature of the *Design Panel*. When the *DataView* is disabled the screen shows only one instance of the control. In this case, the screen is similar to the *Design Panel* in Visual Studio or Eclipse. We provide this feature to allow developers to decide what fits them better during development (e.g they might be overwhelmed by data).

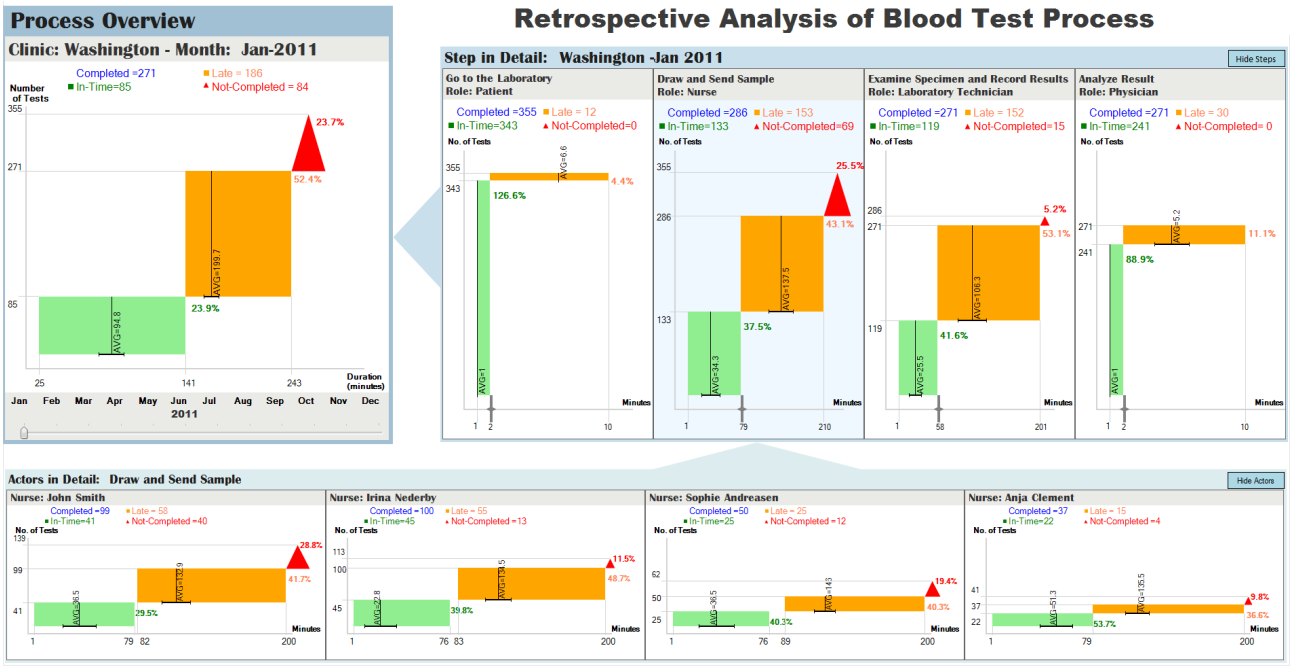

Figure 3. This visualization consists of 3 views (*Process Overview, Step in Detail*, and *Actors in Detail*), allowing managers to explore process, step, and actors' performances at a glance. Each view uses one or more PCDs to visualize the in-time, late and not-completed completions. It is an interactive visualization where changes in *Process Overview* or *Step in Detail* view are reflected in all views.

6. THE MULTI-STEP PROCESS VISUALIZATION

This section presents a custom visualization created with the *Drag-Drop-Set-View-Interact* approach by one of the authors, who is experienced with uVis formulas and the Studio. The final version presented in Figure 3 is a result of an iterative design, where different representation aspects (e.g. color-coding) were considered. The formula language enhanced the development of visualization, but in combination with the Studio, and especially using the *WYBIWYG*, the *Auto-Completion* and the *Modes*, allowed him to rapidly construct several prototypes and conclude to the Multi-Step Process Visualization (MSProVis) (Figure 3).

MSProVis is a multi-view visualization for retrospective analysis, and aims at exposing delays in multi-step processes and supporting the comparisons between steps or between actors executing those steps.[31] MSProVis is an interactive visualization composed of three views: *Process Overview*, *Step in Detail*, and *Actors in Detail*. Each view uses one or more Process Completion Diagrams (PCDs). The PCD aggregates and categorizes event-logs, and visualizes information using shapes and colors. Shapes are placed in a time series plot, where the X-axis presents the duration and the Y-Axis the number of completions. Users start by looking at the Overview PCD, then click on it to see the completion diagrams for each step individually, then click on those steps as needed, to compare the performance of individual actors. In the header of each view the font size decreases with the depth in the hierarchy, and the header color becomes paler. MSProVis computes default thresholds that define lateness based on analysis of historical performance data for each step, and allows managers to adjust those thresholds interactively.

7. USABILITY STUDY

This section presents a usability study with six developers, who were asked to construct a simple bar chart and a simplified version of LifeLines using the uVis Studio. No participant had prior knowledge of the uVis formulas or the Studio.

The purpose of the usability study was to investigate if developers were able to understand uVis formulas and construct visualizations with uVis Studio. This study focused on getting feedback on uVis Studio, and gathering opinions regarding its special features: *WYBIWYG*, *Modes*, and *Auto-Completion*. Further, it aimed at identifying usability problems for further improvement.

7.1 Procedure and Tasks

Each usability test was conducted in three hours on average. One of the authors conducted all the studies, kept notes, and recorded the sessions. The documentation used in this study can be found at[32]

Part 1: Introduction to uVis: This part took 30 minutes on average, where the participant was introduced to uVis. The author had predefined a structure of the concepts he was going to explain, and used that as a guide to present an overview of the tool to the participant. Also, the author showed the participant how to create a bar chart. On the way, the author explained the concepts in detail.

Part 2: Construct a bar chart: The participant was asked to create a bar chart, similar to what the author did in Part 1. The author encouraged the participant to carry out the task in a think-aloud manner. The author also provided the participant with a reference card that showed the bar chart and the formulas for each control. At the end of this part the participant was asked some questions using a 5-point Likert scale. This part lasted one hour on average. The purpose of this part was to introduce the participant to the uVis formula language and the Studio by means of a simple example. Further, it aimed at identifying usability problems.

Part 3: Construct the LifeLines: The author showed the participant an already implemented version of the LifeLines that uses data from a database with 4 tables (Figure 2). He explained some advanced concepts (e.g. the control-join operator "`-=`" that aligns controls vertically, and the `HPos` function of the timescale that aligns controls horizontally). Then, the author asked the participant to create the LifeLines, and think aloud during the process. He provided the participant with the uVis reference card, a reference card where the visualization was shown without any formulas, and some hints in another page. The hint page showed the participant how to write a complex `DataSource` formula with a group by, how to align controls vertically and horizontally, how to convert a `String` type to `Integer` type, and described the `Refresh()` function. At the end, the participant was asked to reply again to the same questions used in part 2. This part of the study lasted one hour on average and aimed at evaluating uVis and the Studio in constructing a custom visualization. At the end of this part, the author showed the participant an initial prototype of MSProVis.[32] The participant was asked to estimate the development time using the Studio and other tools the participant had used.

7.2 Participants

The participants of this study were all experienced developers, who had developed visualizations and user interfaces. In order to evaluate the uVis Studio and it features, we decided to ask for developers that have been using a development environment, and would be able compare the features of the Studio versus what they had used before.

Participant 1: Male; 30 years old; working as a PhD student in Computer Science since 2008; has been coding since 1999; has a very good knowledge of E/R databases; has a good knowledge of MS Excel spreadsheet formulas; has been working for the last two years with visualizations; has coded visualizations from scratch using Python, and has used Prefuse, and D3.

Participant 2: Male; 23 years old; master student in Human Computer Interaction for 6 months; coding since 2007; started using E/R databases some years ago, but does not have a strong background as he does not write his own SQLs; has barely used spreadsheet formulas in MS Excel; has not worked with visualizations, but has done work related to the Human Computer Interaction (HCI) field; has never developed a visualization.

Participant 3: Male; 31 years old; working as a PhD student in Computer Science since 2005; has been coding for the last 12 years; started using E/R databases since 2009, but does not use it very often; started using spreadsheet formulas in MS Excel 8 years ago, but does not use them very often; took only a class on visualization, and has participated in several usability studies about visualizations; coded a TreeMap for his class using Piccolo.

Participant 4: Female; 22 years old; graduated student in Computer Science; has been coding for the last 4 years; took a class in E/R databases in Spring 2011; has basic knowledge of MS Excel spreadsheet; introduced to visualization area in 2011; coded a visualization using ActionScript in Flex, and has used Prefuse.

Participant 5: Male; 26 years old; working as a PhD student in Computer Science since 2007, has used E/R databases since 2004; has used spreadsheet formulas for the last 9 years; has been researching the visualization field since 2008; first uses Spotfire to construct a visualization, and if it is not possible he would code it.

Participant 6: Male, 24 years old; PhD student in Computer Science for one and a half years; has been coding since 2006; started using E/R databases 1 year ago; has been using spreadsheet formulas in MS Excel; worked with visualization for 3 months while he took a class for his studies; created a visualization for his class project using JavaScript and D3.

7.3 Results

The system used in the usability study was a functional prototype that frequently failed. Whenever the author had to assist, he recorded it as a usability problem. Lauesen[33] classifies the usability problems into six types: *Bug, Missing Functionality, Task Failure, Cumbersome, Medium Problem* and *Minor Problem*. We used this classification to classify usability problems. Based on the root problem, we group the usability problems in two categories: uVis Formula Language and uVis Studio. Table 1 shows a summary of the problems identified in this study with programmers. Some problems (e.g. specify a formula that used a *group by* for the `DataSource` property) were encountered by all participants, other problems by only one (e.g. resize a control). Several problems were caused by uVis kernel bugs. For example, after specifying the control-join operator the kernel did not compile the formulas correctly and no error messages were shown in the *Error List*. As a result the participant had to close and re-open the visualization. Another bug was related with the algorithm that the *Auto-Completion* used. The following subsections summarize our observations during the usability study.

uVis Formula Language

Participants could easily specify simple formulas that bind controls to data and map properties to table fields. In complicated cases, they were more skeptical on how to specify a formula that uses the join operator (-<). Some of them solved the confusion by referring to the *E/R Model* and using *Auto-Completion*. This study showed that in order to use uVis Studio, users need to have some database knowledge, and understand SQL concepts such as: joining tables, group by, etc. All of them appreciated the simplicity of the control-join operator. One of them looked at the operator and managed to figure out the algorithm. The others were more skeptical and the author had to explain the operator.

`Parent` refers to another control so that the formulas can use its data row. `Canvas` means that the control is attached to the `Canvas` control and scrolls with it. These were not easily understood by the participants. Two of the participants suggested that a new panel that presents the `Parent` and `Canvas` hierarchy could improve understandability. This window would help them distinguish which controls were bound to what data, and which control contained what controls. Furthermore, improving the functionalities that automatically sets the `Parent` and `Canvas` property may reduce the confusion.

Referring to control properties was easier. They could rapidly write the formulas using the `bang` (used to access control properties) and `dot` operator (used to access fields in tables). One of them used the `dot` operator all the time, while another used only the `bang`. In both cases, uVis compiled the formulas correctly, but the participant had not clearly understood the difference. Making uVis correct the formula to `bang` or `dot`, might help.

uVis Studio

All participants believed that a mature version of uVis Studio can support them better than other tools they had used to develop custom visualizations. Getting familiar with the uVis formula language and having proper documentation might boost their performance. Table 2 presents comments during the debriefing regarding the uVis Studio and it features.

The panels in the Studio allowed them to view information from different perspectives. Changes in the *Property Grid* were immediately reflected in the *Design Panel*, and viewing the *E/R Model* helped them specify

Area	Problem Type	Description	Participant	Possible Solutions
uVis Formula language	Task Failure	The *Parent* and the *Canvas* concept	1,3,6	Better training. Investigating for solutions.
	Minor	The *Dot* and the *Bang* difference	2,5	Improve the uVis Kernel, so it finds out which operator should be used, and automatically corrects the formula. Better training.
	Minor	The use of double quotes	2	Better training
	Medium	Specify a simple Rows formula	1,5,6	Introduce formula suggestions in the E/R model, and allow users to set a property from the E/R model.
	Task Failure	Specify a complex Rows formula that uses group by	1,2,3,4,5,6	Introduce formula suggestions in the E/R model, and allow the user to set a property from the E/R model.
	Cumbersome *, Medium **	The use of the Control-Join (-=) operator	2*, 6 *, 1**	Better training
	Cumbersome	The use of the horizontal position function	2,6	Better training
	Bug	Failed to align controls horizontally	3,4,5	Improve uVis kernel
	Missing Functionality	Transfer the parent control's properties to a child control	2	Improve uVis kernel
	Missing Functionality	Use *This* keyword instead of *Me*	2	Better training, and may be introduce *This* in uVis formula language
	Cumbersome	Refer to a control property	2,6	Better training
	Bug	Delete a control from the form	1,2	Improve uVis kernel
uVis Studio	Missing Functionality	Use the CSS paradigm for positioning controls	1,4	Implement the CSS paradigm
	Minor	Use the top-down approach to position controls	1,2,6	Improve uVis kernel and the Studio. Allow users to set the Bottom and Right property as well.
	Missing Functionality	Missing details in the Design Panel	1,5	Improve the Design Panel. Add tooltips that will show more information can solve this problem.
	Missing Functionality	The position of a control when the position is set to a formula	1,3,4	A new algorithm that will check the references in the formulas, and allow users to move a control.
	Minor	Auto-Completion hides the suggestion that match the typed word.	1, 5	Change the algorithm of the Auto-Completion.
	Bug	In a few cases the Auto-Completion did not suggested anything	1,2,3,4,5,6	Improve the algorithm of the Auto-Completion.
	Minor	Double-click on an error in the Error-List	4	Automatically highlight incorrect formulas in the Property-Grid.
	Cumbersome	Understand the error-messages	1,2,3,4,5,6	Better error descriptions
	Missing Functionality	Debugging feature in the Studio	1,3,4	New panel in the Studio
	Missing Functionality	Control-Data Hierarchy window to show the Parent and Canvas hierarchy of controls	1,4	New panel in the Studio
	Minor	The combo-box in the Property Grid	1,4	The Control-Data Hierarchy window may solve this problem.
	Cumbersome	Keep track of parent-child relationship	4,6	The Control-Data Hierarchy window may solve this problem.
	Minor	The position of the Modes window in the Studio	1,4	Re-design the layout using a slider, and place it to top.
	Minor	Names of the modes in the Modes window	1,2,3,4,5,6	Re-design the layout using a slider, and place it to top.
	Minor	Setting automatically the Left property to *Index*2* was not enough to see that there were several controls	2	Set automatically the Left property to index*5
	Missing Functionality	Set the *Group By* automatically	2	May be group by primary keys from the E/R model using the formula suggestion feature. uVis Kernel
	Minor	Add *Refresh()* by default to the Timescale	2	Set *Refresh()* as default property when the control is created.
	Minor	*Name* and *Text* in the Property were ambiguous	3	Better training
	Missing Functionality	Data-View window to show the row-data from the database	1,3	New panel in the Studio
	Bug	Some cases the Studio failed to set *Parent* and *Canvas* automatically	3,4	Improve the algorithm that sets the Parent and Canvas automatically in the Studio
	Minor	Find a control in the Toolbox	4	Add icons in for each control in the Toolbox
	Minor	Resize a control	4	Increase the offset when the user mouse-over and clicks on the borders of a control.
	Minor	*TAB-Key* to confirm a formula change in the Property Grid	5	Use the TAB-Key, and consider other cases based on other development environments
	Minor	Show default values of a property when the property is empty	5	Introduce a new column in the Property-Grid that shows the values.
	Task Failure	Bind controls to data not from the Rows property	2,3	Better training. The formula suggestion in the E/R model may help.
	Missing Functionality	Missing tooltips in the Studio	1	Add tooltips for all the panels, also for the suggestions in the Auto-Completion list.

Table 1. A summary of the usability problems encountered by each participant in this study. For each problem we present a potential solution. The definition of the problems according to Lauesen:[33] **Bug** = The system is supposed to support the functionality, but it did not; **Missing Functionality**= the system cannot support the user's task; **Task Failure**= The user cannot complete the task on his own or he erroneously believes that it is completed; **Cumbersome**= The user complains that the system is cumbersome; **Medium Problem**= the user find the solution after lengthy attempts; **Minor Problem**= The user finds the solution after a few short attempts.

	Comments
uVis Studio	"The studio helps me see and keep the general goal. When I am programming, I go to deep in details, and after a couple of hours I lose what I started with.", "The studio helps me create better visualizations, as I can see the results when I am creating it, but more controls are needed.", "The Studio fills comfortably the space in the middle between MS Excel, where you can create fast simple visualizations, and the other toolkits used for more complex visualizations.", "I cannot keep track of the controls, and especially the *Parent* . I could not keep the hierarchy straight.", "It is better than writing code, but there may be visualizations that might not be possible to implement, for example animations.",
WYBIWYG	"I have never seen a design panel that shows immediate feedback, that's good.", "It was cool to see the thing I was building as I was writing the formulas.", "The immediate feedback makes a difference. However, not being able to move the boxes after I specified one formula for, say, "left", was a little clumsy.", "It's cool. You don't need to compile. It makes the development easier, but I would like more details, for instance adding warnings that there are hidden instances.", "It's nice to see what you are making."
Auto-Completion	"Auto-Completion is helpful.", "Auto-Completion is good, but in some cases failed.", "The auto-completion tells you what you could use, and this is great."
Modes	"I liked the Modes, you don't need to run, but a better way of showing the Modes is needed", "The Modes are helpful. The naming is not intuitive.", "Modes make sense, but I would position the window over the Design Panel.", "The Modes were helpful as well, I would like a keyboard shortcut for that."
E/R Model	"It facilitated my work, but I would have liked to see the real data as well", "The E/R helped me a lot in specifying the datasource, but still I have to think."
Error List	"Error messages were not easy to understand.", "Trying to find out an error is difficult, as there is no indication where it could have been", "Showing errors better would make it easier."

Table 2. Participants' comments during the study.

the formulas. They could identify the relationships, and the *Auto-Completion* confirmed their assumptions. Two participants suggested that viewing the data in the database, would ensure them that the visualization showed the correct data. The *Error List* panel was less helpful than the other panels, because all participants found some of the errors hard to understand, and asked for better error-messages. However, they appreciated the fact that they could select an error and see the wrong formula colored in red. In this usability study, the *Toolbox* contained a number of controls that allowed them to carry out the tasks. However, they commented that they need more controls. Also, one participant suggested that small icons in the *Toolbox* would have helped her in locating the controls faster.

WYBIWYG: All participants found the *WYBIWYG* feature of the *Design Panel* useful. They could drag and drop controls, bind them to data and get immediate feedback on the screen. From the observations and their comments, we can say that all participants were looking at the *Design Panel* while they changed formulas. Further, they had the possibility to turn this feature off but none of them did. This confirms that they found it useful. Some of the participants asked for more detailed information in the *Design Panel* (e.g. warning the user in case of hidden controls) and being able to move controls after they had specified a formula for their position.

Auto-Completion: From the observations and the participant's comments the *Auto-Completion* assisted them in most of the cases. They appreciated the fact that it was showing what can follow, and this ensured them that they were writing the syntactically correct formula. *Auto-Completion* reduced the cognitive effort participants had to remember the syntax. However, in some cases it failed (e.g. no suggestions after a control function). Further, one participant suggested that introducing tool-tips on mouse over, could provide them more information about a suggestion.

***Modes*:** Participants liked the *InteractView* mode, although it was only used in the second task because of the simplicity of task 1. They were confused as the names (*InteractionView* and *DataView*) were misleading. The names in the *Modes* should explicitly describe the states. One of them suggested using a slider rather than a check-box, to improve understandability. Two of them suggested placing the *Modes* over the *Design Panel* to improve visibility.

Question	Participant 1		Participant 2		Participant 3		Participant 4		Participant 5		Participant 6		Average	
	Task 1	Task 2	Task 1	Task 2	Task 1	Task 2	Task 1	Task 2	Task 1	Task 2	Task 1	Task 2	Task 1	Task 2
How easy was it to use uVis formulas?	4	3	4	4	4	4	2	3	4	1	3	2	**3,5**	**2,8**
How easy was it to understand the uVis properties: Rows, Parent and Canvas.	2	4	5	4	4	4	3	4	4	3	2	2	**3,3**	**3,5**
How easy was it to use the uVis operators (., !, -<, -=)?	2	3	3	3	5	4	4	4	5	2	4	2	**3,8**	**3,0**
How easy was it to bind controls to data?	4	4	4	4	3	5	4	4	2	1	2	2	**3,2**	**3,3**
How easy was it to use uVis Studio?	3	4	4	4	4	4	3	3	3	2	3	3	**3,3**	**3,3**
How useful was the Auto-Completion in the property grid?	3	5	4	4	5	4	5	5	5	5	4	4	**4,3**	**4,5**
How useful was it to preview "live" the visualization in the Design Panel?	4	5	4	4	5	5	5	4	5	5	3	4	**4,3**	**4,5**
How useful was it to interact with the Design Panel?	3	3	5	4	5	4	5	5	3	5	4	4	**4,2**	**4,2**
How useful was it to see the E/R model?	5	5	5	5	5	5	2	3	5	5	4	4	**4,3**	**4,5**
How useful was it to see the Error List window?	2	2	5	5	4	2	2	3	1	4	5	4	**3,2**	**3,3**
How useful was to have the modes?	N/A	3	N/A	4	N/A	4	N/A	4	N/A	5	N/A	4	**N/A**	**4**

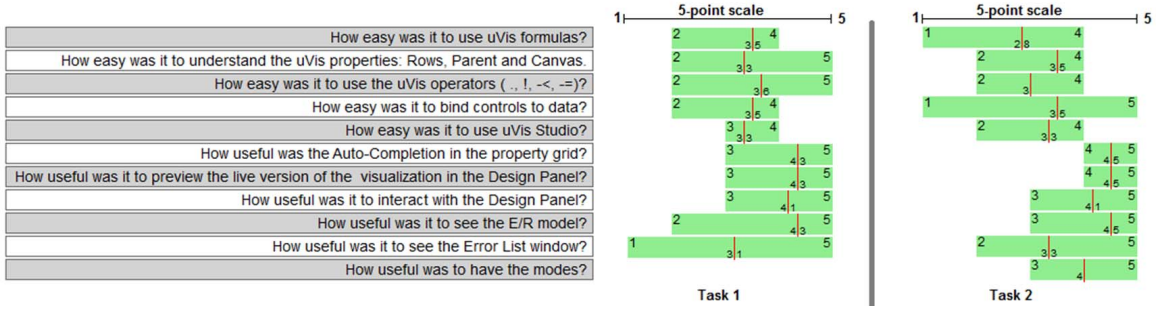

Figure 4. Participants' rating for Task 1 and 2 (top). Minimum, Maximum and Average rating for Task 1 and 2 (bottom).

After each task, participants were asked to fill a questionnaire. Figure 4 show the ratings for task 1 and 2, and the minimum, the maximum and average ratings for each task. Note that in the first questionnaire they were not asked about the *InteractView* mode, because they did not use the feature. Although the sample is small, the average values confirm our observations and their comments during the usability study and the interview. The features of the Studio (*WYBIWYG*, *Modes* and *Auto-Completion*) were important and useful to all of them. The uVis formula language was rated lower as some of them had difficulties understanding it.

All participants looked at the initial version of MSProVis and except for participant 6, they estimated the time it would had taken them to construct the visualization with the Studio versus other tools. Using other tools, they would have spent 2-3 weeks on average. While, using a mature version of uVis Studio, their estimations varied from one hour to less than a week. Although this has to be proven, it still indicates the perceived power of the tool.

8. CONCLUSION AND FUTURE WORK

This paper presents how the uVis toolkit, a formula-based visualization toolkit, has been extended with a development environment, called uVis Studio. Instead of programming, developers construct visualization applying a *Drag-Drop-Set-View-Interact* approach. This environment introduces three new features to facilitate visualization development: *WYBIWYG*, *Modes*, and *Auto-Completion*. The results of our evaluation showed that participants appreciated uVis Studio and its features. Further, there were statements that uVis Studio would save them much time. Based on our experience and evaluation, we present some recommendations to the InfoVis community and industry that should be considered in the future.

- As visualizations are all about the data, visualization tools should have a *WYBIWYG* feature. This feature makes development process more transparent.
- The development environment should allow developers to test end-user interactions without switching workspace. Switching the context from development to run-time is perceived as a barrier.
- Auto-Completion is known for its usefulness, but introducing database related suggestions is a big improvement. It helps with misspellings and using the right syntax of the language.

- Assisting developers with several panels reduces the cognitive effort during the development process, and as a result, it can improve the final result. For example a panel that shows the data from the database is important.

The usability study with developers highlighted the difficult uVis concepts, Studio's usability issues, and suggested improvements. These are being addressed in the new version as uVis Studio targets not only developers, but also users with domain knowledge and limited IT skills. Usability studies will be conducted with these users to prove the effectiveness and efficiency of our approach.

9. ACKNOWLEDGMENTS

The authors would like to thank Prof. Ben Shneiderman, Dr. Catherine Plaisant, and the participants of this study. Authors also thank Mads B. Andersen for his contribution in the uVis project.

REFERENCES

[1] Heer, J., Card, S. K., and Landay, J. A., "prefuse: a toolkit for interactive information visualization," in [*Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*], *CHI '05*, 421–430, ACM, New York, NY, USA (2005).

[2] Fekete, J.-D., "The infovis toolkit," in [*Proceedings of the IEEE Symposium on Information Visualization*], *INFOVIS '04*, 167–174, IEEE Computer Society, Washington, DC, USA (2004).

[3] Bostock, M. and Heer, J., "Protovis: A graphical toolkit for visualization," *IEEE Transactions on Visualization and Computer Graphics* **15**, 1121–1128 (Nov. 2009).

[4] Bederson, B. B., Grosjean, J., and Meyer, J., "Toolkit design for interactive structured graphics," *IEEE Trans. Softw. Eng.* **30**(8), 535–546 (2004).

[5] Bostock, M., Ogievetsky, V., and Heer, J., "D3 data-driven documents," *IEEE Transactions on Visualization and Computer Graphics* **17**(12), 2301–2309 (2011).

[6] Myers, B. A., "Visual programming, programming by example, and program visualization: a taxonomy," *SIGCHI Bull.* **17**(4), 59–66, ACM, New York, NY, USA (1986).

[7] Miyashita, K., Matsuoka, S., Takahashi, S., Yonezawa, A., and Kamada, T., "Declarative programming of graphical interfaces by visual examples," in [*Proceedings of the 5th annual ACM symposium on User interface software and technology*], *UIST '92*, 107–116, ACM, New York, NY, USA (1992).

[8] Norman, D. A., [*The Design of Everyday Things*], Basic Books, Inc., New York, NY, USA (2002).

[9] Lauesen, S., "Vistool for unified data visualization." www.itu.dk/people/slauesen/S-EHR/UnifiedDataVisualization.pdf (2009).

[10] Kuhail, M. A., Pantazos, K., and Lauesen, S., "Customizable time-oriented visualizations," in [*Advances in Visual Computing*], *Lecture Notes in Computer Science* **7432**, 668–677, Springer Berlin Heidelberg (2012).

[11] Roth, S. and Mattis, J., "Automating the presentation of information," in [*Artificial Intelligence Applications, 1991. Proceedings., Seventh IEEE Conference on*], **i**, 90 –97 (feb 1991).

[12] Mackinlay, J., "Automating the design of graphical presentations of relational information," *ACM Trans. Graph.* **5**, 110–141 (Apr. 1986).

[13] Casner, S. M., "Task-analytic approach to the automated design of graphic presentations," *ACM Trans. Graph.* **10**, 111–151 (Apr. 1991).

[14] Chuah, M. C., Roth, S. F., and Kerpedjiev, S., "Intelligent multimedia information retrieval," ch. Sketching, searching, and customizing visualizations: a content-based approach to design retrieval, 83–111, MIT Press, Cambridge, MA, USA (1997).

[15] Roth, S. F., Kolojejchick, J., Mattis, J., and Chuah, M. C., "SageTools: an intelligent environment for sketching, browsing, and customizing data-graphics," in [*Conference Companion on Human Factors in Computing Systems*], *CHI '95*, 409–410, ACM, New York, NY, USA (1995).

[16] Weaver, C., "Building highly-coordinated visualizations in improvise," in [*Proceedings of the IEEE Symposium on Information Visualization*], *INFOVIS '04*, 159–166, IEEE Computer Society, Washington, DC, USA (2004).

[17] Processing. http://www.processing.com/ (Accessed August, 2011).

[18] Google, "Google chart tools." http://code.google.com/ apis/chart/ (Accessed August, 2011).

[19] Google, "Google code playground." http://code.google.com/apis/ajax/playground/ (Accessed August, 2011).

[20] Google, "Live chart playground." https://developers.google.com/chart/image/docs/chart playground (Accessed August, 2011).

[21] Akasaka, R., "Protoviewer: a web-based visual design environment for protovis," in [*ACM SIGGRAPH 2011 Posters*], *SIGGRAPH '11*, 85:1–85:1, ACM, New York, NY, USA (2011).

[22] Spotfire. http://spotfire.tibco.com/ (Accessed August, 2011).

[23] Tableau. http://www.tableausoftware.com/ (Accessed August, 2011).

[24] Omniscope, V. http://www.visokio.com/ (Accessed August, 2011).

[25] Chi, E. H.-H. and Riedl, J., "An operator interaction framework for visualization systems," in [*Information Visualization, 1998. Proceedings. IEEE Symposium on*], 63 –70 (oct 1998).

[26] Card, S. K., Mackinlay, J. D., and Shneiderman, B., eds., [*Readings in information visualization: using vision to think*], Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999).

[27] Pantazos, K. and Lauesen, S., "Constructing visualizations with infovis tools - an evaluation from a user perspective," in [*GRAPP/IVAPP*], 731–736 (2012).

[28] Myers, B., Hudson, S. E., and Pausch, R., "Past, present, and future of user interface software tools," *ACM Trans. Comput.-Hum. Interact.* **7**, 3–28 (Mar. 2000).

[29] Plaisant, C., Heller, D., Li, J., Shneiderman, B., Mushlin, R., and Karat, J., "Visualizing medical records with lifelines," in [*CHI 98 conference summary on Human factors in computing systems*], *CHI '98*, 28–29, ACM, New York, NY, USA (1998).

[30] Pantazos, K., "Video Demostration - uVis Studio." https://dl.dropbox.com/u/5614860/uVis1.0/Studio.wmv (2012).

[31] Pantazos, K., Tarkan, S., Plaisant, C., and Shneiderman, B., "Promoting timely completion of multi-step processes - a visual approach to retrospective analysis," Tech. Rep. HCIL-2012-27, University Of Maryland, Human Computer Interaction Lab (2012).

[32] Pantazos, K., "Usability Study - Documenation." https://dl.dropbox.com/u/5614860/uVis1.0/UsStudy.pdf (2012).

[33] Lauesen, S., [*User Interface Design: A Software Engineering Perspective*], Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2005).