Uvis: Visualization and Interaction with a Drag-Drop-Formula Tool

Soren Lauesen, Mohammad A. Kuhail, Kostas Pantazos, Shangjin Xu, Mads B. Andersen

The IT-University of Copenhagen, Denmark
slauesen@itu.dk, moak@itu.dk, kopa@itu.dk, xushangjin@gmail.com, mban@itu.dk

Abstract. Popular tools for constructing user interfaces use the *drag-drop-set-property* principle. The developer drops visual components (boxes, labels, etc.) on the screen and defines their properties, e.g. position, color and text. However, data presentation and user interaction are very limited. To exceed this limit, programming is needed. Many end-user developers are familiar with the popular tools, but are uncomfortable with programming. Could we improve the drag-drop-set-property principle so that they can make non-standard data visualization and interaction without programming? This paper presents a tool (uVis) that takes a long step in this direction. The principle is to allow each property to be a spreadsheet-like formula that computes position, color, etc. A formula can combine data from several database tables with data about components and dialog data provided by the end-user. Formulas can also handle events and provide interaction.

Keywords: Data visualization, database, interaction, user interface, end-user development.

1 Introduction

A user interface consists of general-purpose components (e.g. boxes, labels, curves, and scales) each with its own property values (position, size, color, etc.) and its own event handlers.

In order to create a specific user interface, we must somehow put the necessary components on the screen and define their properties and event handlers. *Programmatic* tools require that the developer writes a kind of program that creates the components and sets their property values. *Drag-drop-set-property* tools allow the developer to manually drag and drop the components and set their properties. *Charting tools* contain a program that creates a pattern of components (a predefined visualization, e.g. a pie chart) and lets the developer specify some of the properties. Combinations of these principles exist too, of course.

Our focus in this paper is *non-standard data visualizations* where predefined visualizations cannot support the end-users adequately. Figure 1 shows two examples from the medical area. The Lifeline screen shows the patient's notes, diagnoses and medicine on a time scale with multiple zoom areas. Icon shapes and color indicate the

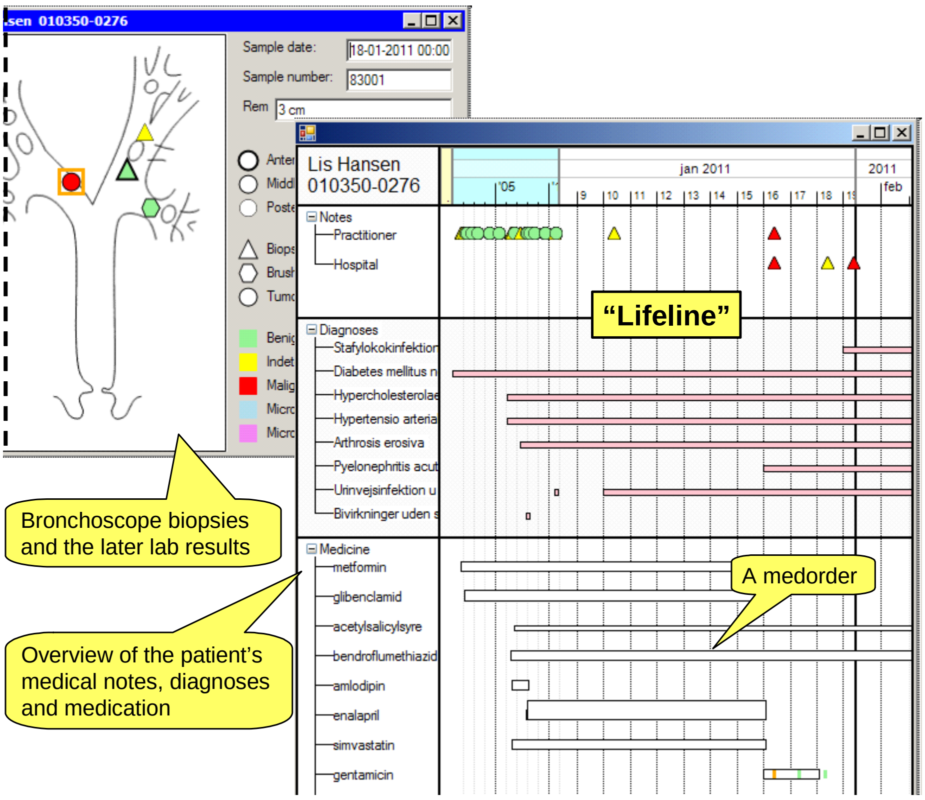


Figure 1. Two screens with custom visualization of medical data

kind of note, the height of boxes indicate the medicine dose, etc. You can see at a glance which medicines the patient gets now, how long the patient has got them, and how they time-wise relate to diagnoses and notes. The data exist already in the data-base; we just show them in a different way. As far as we know, no real-life health record system uses such screens today.

The bronchial screen shows where biopsies have been taken, how they were taken, and what the lab results tell. Clinicians also use the screen to record the biopsies and lab results. The diagram of the bronchia is a simple drawing made in the department.

From a theoretical perspective these visualizations are not novel, just variations of known themes such as Lifelines (Plaisant et. al. [17]) and geographical maps. Yet they are highly useful in their domain, and at present they can only be made with programmatic tools.

Our goal is that end-user developers could make these visualizations in coopera-tion with ordinary end-users. We will use the term *local developers* or just *developers*, to mean these end-user developers. They are often familiar with popular tools such as MS Access and Excel, and they sometimes make small applications for their own use or for use in the department. They are rarely comfortable with programming and with present tools they cannot make data visualizations such as Figure 1.

Myers et al. [12] gave an excellent overview of user interface tools in 2000 and explained why drag-drop-set-property tools (called *interface builders* and *interactive graphical tools*) were much more successful with local developers than program-based tools. They and other authors [3] report that spreadsheets are the only kind of "programming" widely accepted by end-users. When we say "programming" in this paper we do not include spreadsheet-like formulas. We define a "real" program as a text that generates visual components and sets their properties. It runs in a context that is different from the user screen itself. It may use loops and invisible objects such as variables and methods.

Why are drag-drop-set-property tools so popular? Most people give up programming. They cannot see the connection between the program and the visible result, and they cannot write a program themselves. Norman describes this as the gulf of evaluation and the gulf of execution [13].

What is the situation today? A recent study (Pantazos [15]) showed that local developers (called *savvy users* in the paper) still need better tools and more attention from the information visualization community.

Uvis is a drag-drop-set-property tool that allows local developers to build tailor-made user interfaces without real programming. Developers put together visual components and specify their properties and event handlers with spreadsheet-like formulas. A formula corresponds to a spreadsheet formula and uses a similar notation, but it is able to combine data from databases, visual components and end-user input.

Using uVis we developed the application in Figure 1 in 10 hours. End-users can use it to see real patient data, enter data about the biopsies, etc. The database existed already. With a bit of training, local developers in the hospital could have made the application. Only formulas were needed, no real programming.

2 Related Work

Industry tools such as Microsoft Access, Microsoft Visual Studio, Eclipse and NetBeans allow developers to construct user screens with drag-and-drop of text boxes, buttons and other components. For each component the developer sets size, position, color and other properties to a constant. This approach can quickly generate a mockup that looks right, but shows only dummy data. In simple cases you can connect a database table to a component, for instance to make a combo box or a data grid component. However, to make something like the Lifeline screen, "programming behind" is needed and only professional programmers can do this.

Charting tools such as pie charts and bar charts are provided in Excel, Google Spreadsheets [7] and as separate packages. These tools don't require programming skills and are widely used. However, to integrate them with a production application, the end-user has to copy and paste data from the application to the tool (or a programmer has to write code that does it). Without programming there is no way to create visualizations beyond what is predefined. For instance, something like the Lifeline couldn't be made. Further the end-user has little interaction with the data and cannot feed data back to the existing application.

Data analysis tools such as Tableau [20], Polaris [19], Spotfire [18] and Omniscope [14] integrate well with existing data and help users explore the data. They don't require programming skills. Here too there is no way to create visualizations beyond what is predefined and something like the Lifeline couldn't be made. Further there is only predefined interaction with the data and the end-user cannot feed data back to the existing application.

Graphics libraries such as GDI+ and Java 2D are available for many programming languages. They provide basic components such as line, polygon, and ellipse. By means of a program you can make them create any visualization, bind to any data and perform any interaction. However, to accomplish this, you must be a professional programmer.

Visualization Toolkits allow you to construct traditional and new visualizations in a programmatic way that is simpler than using graphics libraries, for instance by means of a domain-specific programming language. Examples are Protovis [1], D3 [2], Prefuse [8], Improvise [22] and Infovis [6]. These toolkits don't use a drag-and-drop approach, they are not easy to integrate with existing relational data, and you cannot feed data back to the existing application without programming.

Spreadsheet-based graphical tools allow you to construct graphical user interfaces. A component corresponds to a named spreadsheet cell that can be positioned anywhere on the screen. The properties of the components are specified with formulas. Examples are Forms/3 [3], KidSim/Cocoa [5], NoPumpG [11]. Like visualization toolkits, these tools don't use a drag-and-drop approach, they are not easy to integrate with existing relational data, and you cannot feed data back to the existing application without programming. Further, screens with advanced visualizations that require cross referencing are hard to implement with these tools.

3 Uvis Principles

The uVis development environment is similar to traditional tools where the developer drags components to the user screen and defines the properties of the components. In traditional tools, most of the properties can be constants only, and this limits what can be done without programming.

The basic uVis idea is that any property can have a formula that computes the value of the property. The challenge is to provide a formula language that is sufficiently strong to generate complex user interfaces based on general-purpose components such as boxes and labels. Figure 1 is an example. To achieve this, the formulas must be able to address data from several sources:

1. External data (e.g. in a database)
2. Property values in the same or other components.
3. Dialog data provided by the end user, e.g. the position of a scroll bar or the contents of a data entry textbox.

It must be possible to combine these data with traditional operators (e.g. +, -, *, <, =) and traditional functions (e.g. Sin, Format, Choose). This gives us a declarative formula language as powerful as spreadsheet formulas, but able to address many kinds of objects, not just spreadsheet cells.

Operands in a formula must be able to walk from object to object to get data. As an example, the operand *compA.compB.tableC.fieldD* will walk to visual component compA, from there to the related compB, to its related record in tableC and then get its fieldD. This *walk principle* is used extensively in uVis.

A spreadsheet has circular references when a formula in a cell refers to cells that refer back to the cell we started with. This may occur for uVis formulas too when they address properties in other components. As for spreadsheets, it is detected and reported as an error.

Formulas must be able to do more than computing a value:

4. Generate several instances of a component. A special property *Rows* can do this.
5. Perform actions, for instance setting a value, refreshing the screen, and opening a user screen. Event handler properties can do this in response to end-user actions and timer events.

The screens in Figure 1 use these mechanisms in many places. Here are some examples.

Generate several instances. The colored icons in the bronchial screen are generated by a single *Glyph* component, dragged and dropped by the designer. He gave it a *Rows* formula that retrieves all the patient's bronchial data rows from the database and generates a glyph instance for each of them. As a result, each instance is connected to a data row.

Refer to external data. For each glyph instance, the Top and Left formulas compute the position by means of fields in the connected row. In the same way, each glyph instance computes its color and shape from other fields in the row.

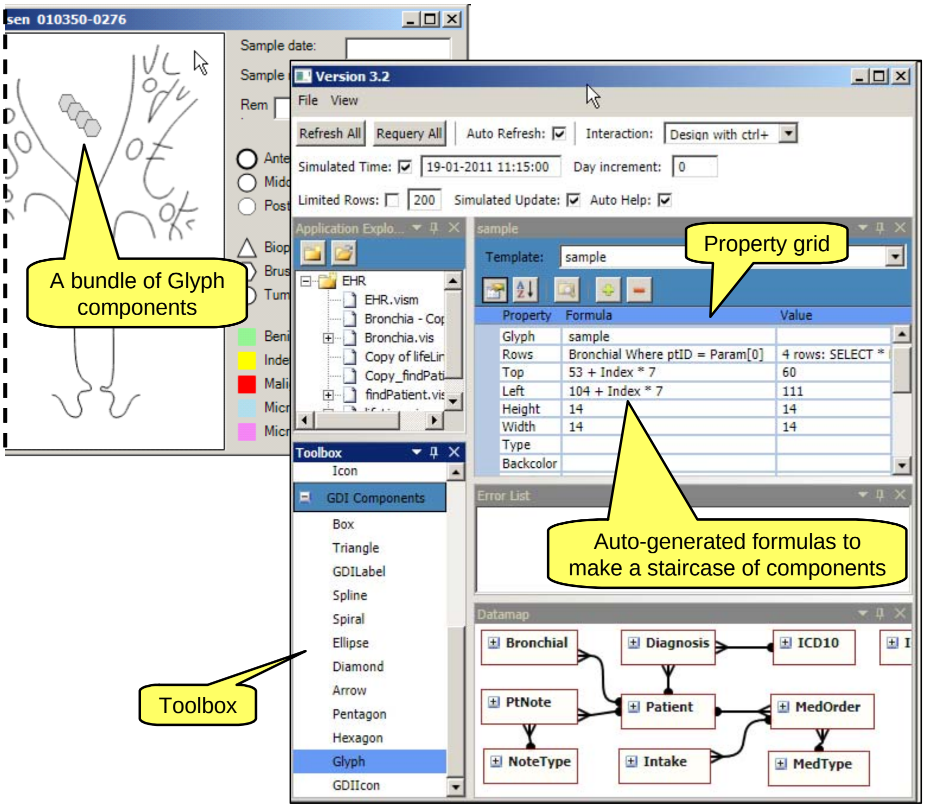
Refer to property values and end-user data. The bronchial screen uses a box component to mark the selected glyph instance. The box must know which glyph to mark, so the designer added his own property, *Selected*, to keep track of the selected glyph. The box's Top formula walks to Selected, then to the selected glyph, gets its Top value and uses it to compute its own Top. The Left formula does the same.

Perform actions. When the end user clicks a glyph, the marker box must show it. To achieve this, the designer specified a formula for the glyph's *Click* property. When the end user clicks a glyph, the Click formula is executed. It sets the marker's *Selected* so that it refers to the glyph that was clicked. Next, it asks uVis to *refresh* the screen, i.e. re-compute all formulas and update the components as needed.

3.1 Uvis Development Environment

In this section we present the uVis development environment and give detailed examples of formulas. Figure 2 shows uVis Studio when the local developer has constructed the trivial part of the bronchial screen: the components that occur only once. He has for instance dropped an icon component in the left part of the screen and has set its File property to show the bronchial diagram. This is quite similar to how popular tools work today.

In order to access external data, he needs a *data map* file that has a connection string to the database and specifies the available tables and relationships. For now, we assume that the local IT department has provided it. The developer told uVis Studio to



Auto Refresh: ☑ Interaction: Design with ctrl+
Simulated Time: ☑ 19-01-2011 11:15:00 Day increment: 0
Limited Rows: ☐ 200 Simulated Update: ☑ Auto Help: ☑

sample
Template: sample

Property	Formula	Value
Glyph	sample	
Rows	Bronchial Where ptID = Param[0]	4 rows: SELECT *
Top	53 + Index * 7	60
Left	104 + Index * 7	111
Height	14	14
Width	14	14
Type		
Backcolor		

Figure 2. Setting the Rows property to create a bundle of components

use this data map file, and as a result uVis shows the data map as an E/R diagram (lower right). The developer can detach and enlarge the diagram, expand a box to see a list of the fields, and double-click a box to see the table with data.

During construction the local developer sees the bronchial screen exactly as it will look to the end-user. He can also interact with it as the end-user would do. In order to change formulas, he holds *Control* and clicks a component. As a result, the property grid (middle right) shows the formulas and also the result of the formulas for this component instance. He can edit the formulas and as soon as he types Enter, uVis updates the user screen accordingly.

Generate several instances. The developer's next task is to make uVis create an icon for each biopsy. The number of icons is dynamic; it depends on the database contents.

He has dragged a Glyph component from the toolbox and dropped it on the bronchial diagram. A glyph can appear as different shapes: circle, triangle, etc. Initially it appears as a gray hexagon. The property grid shows the properties of the glyph. The developer has set the name of the component to *Sample*. Now he sets the Rows property in this way:

```
Rows: Bronchial Where ptID = Param[0]
```

Bronchial is a table in the database that has a row for each biopsy. It has a *ptID* field that identifies the patient. The bronchial screen was opened with the current *ptID* in the parameter property, *Param*. The Param property is a list, and Param[0] is the first element in the list. The result of the Rows formula is the set of Bronchial rows for the current patient. As soon as the developer had typed the Rows formula, uVis generated a glyph component for each row and connected it to the row. Furthermore, uVis set the Top and Left formulas so that the glyphs appear a staircase of gray icons to visualize how many there are.

If the developer had to express the Rows formula in the usual way as an SQL statement, it would look like this:

```
SELECT Bronchial.ptID, Bronchial.y, Bronchial.x, Bronchial.kind, Bronchial.result,
Bronchial.ant_post, Bronchial.splDate, Bronchial.splNumber, Bronchial.remark FROM
Bronchial WHERE [Bronchial.ptID] = 0103500276
```

The developer could have written SELECT * instead of listing all the fields he needs, but this would retrieve all fields in the table. In a real-life database with many fields, this may be very slow.

He would also have to insert the actual patient ID (0103500276) into the SQL statement in a programmatic way.

Uvis generates the SQL statement automatically based on the Rows formula, collects the necessary fields from all the formulas in the screen, and inserts them as the SELECT part. Although the Rows formula still has the flavor of an SQL statement (the Where clause), it is much more compact and simpler to write than the real SQL statement.

When uVis creates components with a Rows formula, they become a *bundle* of component instances. Each component in the bundle has an *Index* property that is 0 for the first component, 1 for the next, etc.

When uVis has created the components, it evaluates the property formulas and sets the property values. In our example, uVis Studio automatically defined the formulas for Top and Left so that the icons appeared as a staircase:

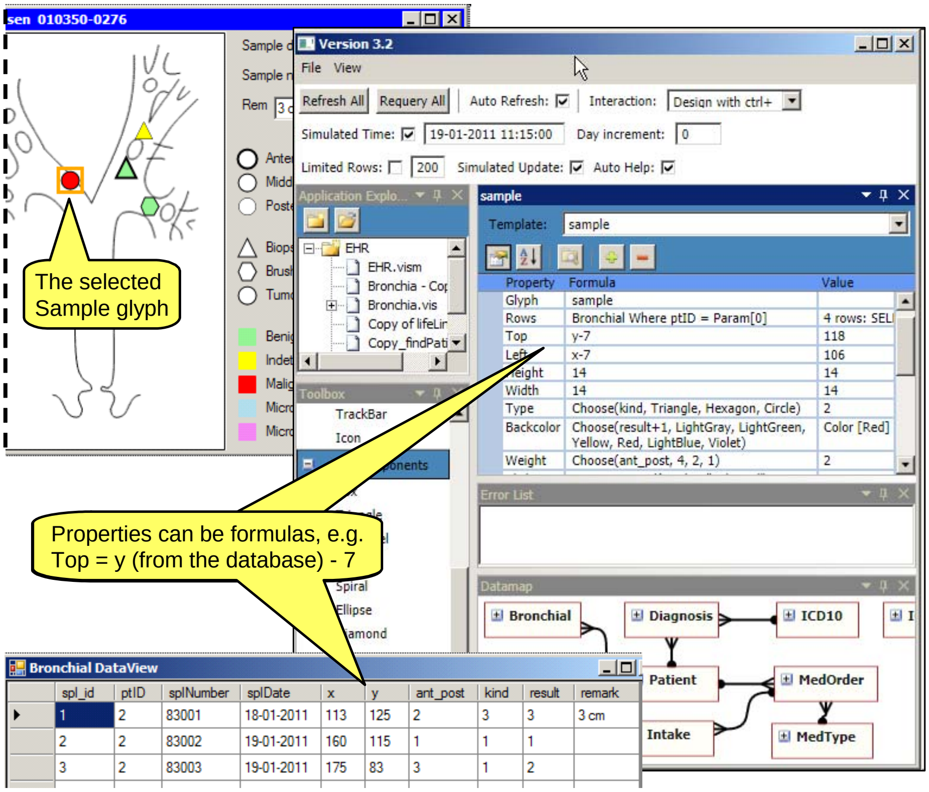
```
Top: 53 + Index * 7
Left: 104 + Index *7
```

In this way the first component got Top = 53 (Index = 0), the next Top = 60, etc.

Refer to external data. The developer's next task is to set the other formulas for the glyph. Figure 3 shows the result. He has selected one of the glyph instances. He sees its property formulas in the property grid and also the actual property value. The formulas are the same for all the component instances, but the values vary between instances in the bundle. He has also opened the bronchial table. The selected glyph corresponds to the first row of the table.

The Left position of the glyph is computed by the formula *x-7*, where x is a field in the database. For this specific glyph, the formula gave 106 as the result and uVis put the glyph 106 pixels from the left edge of the user screen.

The shape of the glyph is computed by its Type property. The formula retrieves the *kind* field from the database and chooses the corresponding shape: triangle, hexagon



Auto Refresh: ✓ Interaction: Design with ctrl+

Simulated Time: ✓ 19-01-2011 11:15:00 Day increment: 0

Limited Rows: ☐ 200 Simulated Update: ✓ Auto Help: ✓

sample

Template: sample

Property	Formula	Value
Glyph	sample	
Rows	Bronchial Where ptID = Param[0]	4 rows: SEL
Top	y-7	118
Left	x-7	106
Height	14	14
Width	14	14
Type	Choose(kind, Triangle, Hexagon, Circle)	2
Backcolor	Choose(result+1, LightGray, LightGreen, Yellow, Red, LightBlue, Violet)	Color [Red]
Weight	Choose(ant_post, 4, 2, 1)	2

Bronchial DataView

	spl_id	ptID	splNumber	splDate	x	y	ant_post	kind	result	remark
▶	1	2	83001	18-01-2011	113	125	2	3	3	3 cm
	2	2	83002	19-01-2011	160	115	1	1	1	
	3	2	83003	19-01-2011	175	83	3	1	2	

Figure 3. Using database fields to set position, shape and color

or circle. When the developer has typed the formula or changed it, the system updates the user screen immediately.

During typing, uVis gives auto-complete suggestions (Intellisense).

Save the screens. When the developer closes the bronchial screen or closes uVis Studio, uVis saves the screen as a vis-file that can be read with simple tools, e.g. notepad. Figure 4 shows part of the vis-file for the bronchial screen.

3.2 End-user Data and Interaction

We will show an example of how the end-user can interact with the screen. This also illustrates how a formula can address properties in other components.

The developer has decided that when the end-user clicks a *Sample* icon, it should be marked with an orange frame. Further, the details of the sample should be shown in the text boxes at the top right of the bronchial screen (Figure 3).

The key part of this is a box component that serves as a Marker. The developer has given it these property formulas:

```
Box:            Marker
Selected:       Init -1   ' The selected sample. <0 when nothing is selected.
Visible:        Selected >= 0
Top:            Sample[ Selected ].Top-3 Default 0
Left:           Sample[ Selected ].Left-3 Default 0
Height:         20
Width:          20
Weight:         3
BorderColor:    Orange
BackColor:      Transparent
```

The developer has added his own property, *Selected*. It is not a built-in property such as Top and BorderColor, but what we call a *designer* property. *Init -1* means that *Selected* initially is -1, but the value can change as a result of end-user actions. When the end-user selects a *Sample* glyph, *Selected* should become the Index of the glyph.

Visible is a built-in property. The formula says that the Marker should be visible when something is selected (*selected >= 0*). Initially it will be invisible.

Top says: Walk to the bundle of samples. Take the glyph with the index given by *Selected*. Take its Top property value and subtract 3 pixels to make the orange frame surround the glyph. If this doesn't work, for instance because nothing has been selected, use the default value and make Top = 0. This is an example of addressing a property in the same component (*Selected*) and in another component (*Sample[i].Top*). Notice that uVis can address specific items in a bundle as if it was an array.

The remaining properties should now be obvious. The result is that an orange square shows around the glyph when it is selected.

When a Sample is selected, the screen should also update the text boxes at the top right. The developer handles it with formulas like this one for the sample-date text box:

```
TextBox: splDate
…
Text: Sample[ Marker.Selected ].splDate Default ""
```

The Text property specifies what to show in the text box. The formula says: Walk to the bundle of samples. Also walk to the Marker box and get its *Selected* value. Use it as the Index in the bundle to get the selected glyph. Finally walk to the data row connected to the glyph and get its sample date (*splDate*). If nothing is selected, use an empty text as the default.

The walk principle. Data references in uVis formulas use the walk principle: The system walks from object to object to get the result. The Text formula above is an example of this. The formula walks to a bundle of components, then to another component to get the index and use it to select a component in the bundle, and finally to a row connected to the component to get the desired field.

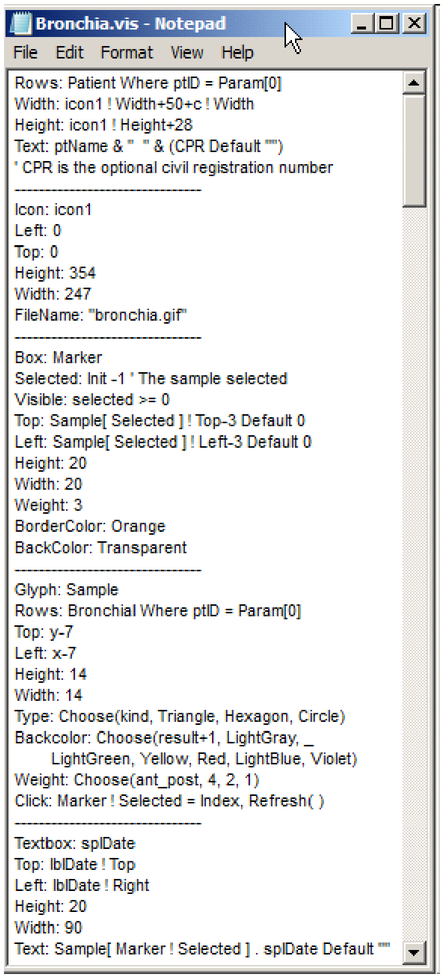
Dot-operators and name ambiguity. Understanding a formula such as the Text formula above requires good knowledge of what are visual components and what are database elements. To help the developer, uVis can change the dots in the formula to show what is what. A dot (.) means database elements and a bang (!) means visual component elements. Using these dot-operators, uVis would show the Text formula like this:

```
Sample[ Marker ! Selected ] . splDate
Default ""
```

With a bit of training, the developer can see at a glance that *Selected* is a component property and "splDate" is a database field. The dot-operators also help resolving name ambiguity. Assume that the database had a field called *Top*. The developer cannot change this name, nor can he change the property name *Top*. But he can use bang or dot to tell the compiler whether he means a component property or a database field.

Event handler properties. To make the selection construction run, we need a way to set *Selected*. This is done through the *Sample* component. It should respond when the end-user clicks it. The developer has defined an event handler property for it:

```
Glyph:   Sample
…
Click:   Marker ! Selected = Index, Refresh( )
```



```
Rows: Patient Where ptID = Param[0]
Width: icon1 ! Width+50+c ! Width
Height: icon1 ! Height+28
Text: ptName & "  " & (CPR Default "")
' CPR is the optional civil registration number
------------------------------
Icon: icon1
Left: 0
Top: 0
Height: 354
Width: 247
FileName: "bronchia.gif"
------------------------------
Box: Marker
Selected: Init -1 ' The sample selected
Visible: selected >= 0
Top: Sample[ Selected ] ! Top-3 Default 0
Left: Sample[ Selected ] ! Left-3 Default 0
Height: 20
Width: 20
Weight: 3
BorderColor: Orange
BackColor: Transparent
------------------------------
Glyph: Sample
Rows: Bronchial Where ptID = Param[0]
Top: y-7
Left: x-7
Height: 14
Width: 14
Type: Choose(kind, Triangle, Hexagon, Circle)
Backcolor: Choose(result+1, LightGray, _
      LightGreen, Yellow, Red, LightBlue, Violet)
Weight: Choose(ant_post, 4, 2, 1)
Click: Marker ! Selected = Index, Refresh( )
------------------------------
Textbox: splDate
Top: lblDate ! Top
Left: lblDate ! Right
Height: 20
Width: 90
Text: Sample[ Marker ! Selected ] . splDate Default ""
```

Figure 4. Saved vis-file

When the end-user clicks a *Sample* glyph, uVis performs the actions (*statements*) in the click formula. As a result, *Selected* will become the index of the clicked glyph. The statement *Refresh( )* asks uVis to re-compute all formulas and redraw components where a property value has changed. As a result the Marker will appear or move, and the text boxes will show data about the selected sample.

In contrast to ordinary formulas such as *Top*, an event handler formula cannot be evaluated at any time. An event handler is evaluated only in response to an end-user action. In industrial tools, components send a lot of events to each other, for instance *BorderChanged* and *ValueChanged*. Uvis has no such events because all updates are made with *Refresh( )*, which re-computes and redraws as needed. This is the spread-sheet principle, but it assumes good performance (see section 5).

Event handler statements include setting a value in a property or a database field, committing a database transaction, opening a form, refreshing the screen, and reque-rying the database.

Depending on the application, it may be necessary to perform actions beyond the built-in uVis statements, for instance to send an email or transmit data to/from an external system. This requires that someone makes a piece of real program, tests it and exposes it as a method that can be called from an event handler formula. This is "programming behind", but it is used far less than in the traditional approaches.

3.3 Joins - Walking Among Tables

Above we showed examples of walking between visual components. It also makes sense to walk between database tables.

The data map in Figure 3 shows tables connected with crow's feet. Each crow's foot corresponds to a one-to-many relationship between the tables. It makes sense to walk along these feet. As an example, the Patient table has a one-to-many crow's foot to the MedOrder table. It symbolizes that we can walk from a patient to many medi-cine orders. Walking the other way from a MedOrder to Patient, we come to only one patient.

Let us look at a tough example, the medicine orders at the bottom of the Lifeline (Figure 1). We have connected the Lifeline screen to a single patient row with this formula:

```
Rows: Patient Where ptID = Param[0]
```

We want to generate a box for each of the patient's medicine orders. We drop a box in the lower part of the Lifeline screen and give it these properties:

```
Box:     MedOrderBox
Rows:    Parent -< MedOrder
```

The Rows formula means: Start in the patient row connected to the user screen (Par-ent means the user screen in this case). The -< symbolizes a one-to-many crow's foot. Now walk along the crow's foot to the MedOrder table. The result is a set of rows, one for each of the patient's medicine orders. Each row contains fields about the medicine order (the type of medicine, the start and stop time for the medication, the amount per dose, the dose, and the number of times per day).

The result is a staircase of medicine orders. We align them to the time scale at the top of the Lifeline with these formulas:

```
Left:    timeScale ! HPos(startTime)
Right:   timeScale ! HPos(stopTime)
```

The Left formula means: For this medicine order, walk to the time scale component at the top of the Lifeline. Call its horizontal position property (HPos) and ask it to translate the startTime of the medication to a pixel position. Use this position as the Left property. The Right formula works the same way. The result is that each MedOrderBox is stretched correctly in the time dimension.

What about *Width?* Uvis accepts any two of Left, Width and Height because we observed that developers often tried to do so.

We want to show the amount of medicine as the height of the boxes. To do this, we multiply the amount per unit, the dose and the times per day. These fields are readily available in the medicine order row. However, is this a large or a small dose? To indicate this, we need access to the MedType table that contains the DDD (normal daily dose). Fortunately the E/R diagram has a crow's foot from MedOrder to MedType. We can just continue walking from MedOrder to MedType.

To do this, we change the Rows formula and can then include DDD in the Height formula:

```
Box:     MedOrderBox
Rows:    Parent -< MedOrder >- MedType
Height:  (amount * dosage * timesPerDay) / DDD * 8
```

The Rows formula now gets the MedOrder rows as before and then walks on for each MedOrder row to the related MedType. It includes the relevant MedType fields in each of the rows, e.g. DDD.

The last step of the Rows formula is a one-to-many relation, symbolized with >-. If the developer makes a mistake and types -< instead, or dot or bang, the compiler will replace it with >- to show what kind of relation it actually is.

In practice, there are often missing data in the tables. As an example, the real medicine data we work with often lack a reference to MedType. The visualization can easily deal with this by means of the *Default* operator explained earlier.

Here too, the notation is very compact. If the developer had to write the SQL statement for this Rows formula, it would look like this:

```
SELECT MedOrder.ptID, MedOrder.amount, MedOrder.dosage, MedOrder.timesPerDay,
MedOrder.startTime, MedOrder.stopTime, MedOrder.medID, MedOrder.orderID
MedType.DDD
FROM MedOrder LEFT JOIN MedType ON MedOrder.medID = MedType.medID
WHERE [MedOrder.ptID] =0103500276
```

4 Development Scenario

In this section we show how a local developer would use uVis to construct the screens in Figure 1 and deploy them in the department.

Connecting to the database

First the local developer needs a *data-map* file that gives uVis access to the existing database or to equivalent system services. Most likely he will use an anonymized test version of the database. The central IT department would give him the necessary permissions and help him set up the data-map file.

Next he opens uVis Studio and tells it to select the data-map file. His computer screen will now look like Figure 2, but the bronchial screen is an empty default screen.

Constructing the screens
The local developer will now drag and drop components on the screen and set their properties as explained above. He can develop several screens at the same time.

Debugging and Testing the Screens
The local developer has made the basic test of the screens while he constructed them. Whenever he typed or changed a formula, the system immediately retrieved data from the database and showed it exactly as the end-user would see it.

Uvis detects errors and highlights them in the formulas and in the error list panel. Uvis can also show the raw data behind the visual components to help the designer locate a problem.

However, the local developer should make additional tests of the screens. He should make usability tests of the screens to ensure that they are understandable to other end-users. He should test that the screens show abnormal data correctly; preferably he should have a small test database with abnormal data for this. And he should test that the queries don't overload the database. Here he needs a large test database for measuring performance.

We have paid much attention to the test situations. For instance, it is easy to switch between databases. You make a copy of the data-map file for each database and modify the database connection string. When you run a test, you simply open the proper map file. You can run uVis in various test modes, for instance simulate that the date and time is something that matches the data you test with. You can avoid that uVis makes real database updates, but only simulates that data has been updated. To avoid that a database query by mistake takes a very long time and blocks testing, you can limit database access so that the system never retrieves more than for instance 200 rows from a table.

Deploy the new user interface
After testing, the local developer has to deploy the system so that end-users can use it for production. He needs the necessary rights to access the production database. Further, the end-users need rights to open the map file and access the production database.

He puts the map-file and the vis-files in a folder on the department's file server. The end-users can now open the map file in the same way as they would open a Word file. As a result, uVis will connect to the database and open the start screen that the local developer made. From there, users can navigate to other uVis screens.

Close integration with existing applications
In the scenario above, end-users opened a map-file to see the existing data through uVis screens. Another approach is that an existing application asks uVis to open the map-file and show the uVis screens. This requires a small change to the existing application so that it calls a uVis API when the user for instance clicks a certain button on an existing screen.

Some professional visualization developers have responded to uVis saying that it might save them 90% of their visualization efforts. To accomplish this, they might use uVis API's directly in their programs.

5 Evaluation and Limitations

We have developed many small applications in an experimental way with uVis and also developed customizable versions of bar charts, pie slices and other traditional visualizations (Pantazos [16]). Most of them became surprisingly simple as we learned to utilize the power of the formula language.

Performance

When the end-user interacts, the event handler formula will usually do something and then ask uVis to refresh the screen in the same way as a spreadsheet recalculates all cells. There are various ways to optimize refreshing, for instance only recompute properties that depend on the item changed. At present we don't try to optimize. We get adequate performance with a simple algorithm:

Recompute all formulas, requery the database if an SQL statement has changed, set all component properties to the new computed value (whether it has changed or not), and update the screen accordingly.

Queries to the database are usually the performance bottleneck. When a component generates several instances of itself by means of a Rows formula, uVis retrieves all the rows with a single query. This means that the bundle of bronchial biopsies is retrieved with only one query. Bundles can be nested so that we have bundles of bundles. The rule applies here too: Uvis retrieves all the second-level bundles with a single query.

The table below shows the performance for the Lifeline with the screen shown in Figure 1 (average of 10 measurements on an ordinary 2GHz PC with 2 GB memory and a local MS Access database). The total time to open the screen is 0.7 seconds including 0.4 s to make 8 queries to the database.

Most of this time would be spent also if the screen was produced by a hand-coded program. The only overhead uVis adds is the time to scan and compile the vis-file and the time to compute the properties. This amounts to 0.09s, 13% of the total time.

The time to refresh the entire screen is 0.08 s. It is spent on computing the properties and showing (rendering) the components.

Usability for developers

Ease of learning is an issue when we aim at non-programmers. However, usability is not only a property of the formula language. Computerized cognitive aids and human assistance is also very important to help developers get started.

Time to open Lifeline, Figure 1		**Refresh time**
	ms	ms
Scan the .vis-file (5500 chars)	23	
Compile 180 formulas	20	
Create 146 components	90	
Compute properties for 146 components	43	46
SQL queries (8 queries, 140 rows total)	401	
Show 146 components	94	32
Total time to open	671	78

For the last year we have run usability tests of the tool including the planned assistance. Based on the results, we gradually improved the tool. As an example, we observed that test subjects were puzzled when they specified *Rows* and nothing seemed to happen (because all the instances appeared on top of each other). As a result, we changed the development environment so that the instances initially appear as a staircase. We have improved many other things too, e.g. terminology (*Rows* or something else?), auto-completion where the tool suggests what to type, direct inspection of the database contents (table view) and direct inspection of the property values (Kuhail [10]).

We have made usability tests with a total of 24 typical local developers (non-programmers) and 6 developers that had experience with other visualization tools. We tested with one user at a time. The tests took between 2 and 3 hours.

All subjects except one could make simple visualizations with uVis, but sometimes had to ask for help because there was a bug in the system. As the visualization complexity increased, fewer could make it within the 2-3 hours. The subjects understood the concepts and could reason about the formulas. However, the join operators (-< and >-) were not as intuitive as we had hoped. This was related to the subjects having a weak understanding of databases in general.

Some subjects said that they would like to experiment more with the tool on their own. The subjects who had programmed visualizations professionally said that a mature uVis would have saved 90% of their time.

Details of the evaluation are published in Kuhail [9].

Limitations

Uvis is currently an operational prototype that proves the concept: It is possible to make such a tool, it can become sufficiently easy to use for the target developers, and it performs well on the computer. However, many details are missing before local developers can use it on their own, for instance better documentation and tooltips.

One important limitation is that uVis at present only runs on the Windows Forms platform. When we started the project early 2009, the Web wasn't suited for such a tool, but this has changed with HTML 5 and OData.

Another important limitation is that it is harder to get access to raw table data than we anticipated. Many commercial systems have buried their data behind web-services and multi-layer architectures, and are unable to give access to data in such a way that end-user developers can join tables and filter them according to end-user needs. Here too, OData may help.

6 Conclusion

Uvis allows end-user developers to implement non-standard data visualization and interaction by means of spreadsheet-like formulas for the component properties. During development, the tool shows the final screens all the time and allows the developer to interact with the screens in the same way as the end-user. Without changing context, the developer can drag and drop components and edit their property formulas. There is immediate visual feedback for each change. A formula can access database tables, visual components and end-user input.

Usability tests indicate that the target developers can learn the basics of the tool in 2-3 hours.

References

1. Bostock, M., Heer, J.: Protovis: A graphical toolkit for visualization. IEEE Trans. Vis. and Comp. Graphics, 15(6):1121–1128 (2009)
2. Bostock, M., Ogievetsky, V., Heer, J.: D³ Data-Driven Documents, Visualization and Computer Graphics, IEEE Transactions on, vol.17, no.12, pp.2301-2309 (2011)
3. Burnett, M., Atwood, J., Djang, R. W.: Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. In: Journal of Functional Programming, Volume 11, Issue 02, 155-206 (2001)
4. Brunett, M., Rothermel, G., Cook, C.: An Integrated Software Engineering Approach for End-User Programmers. In: Springer Netherlands, pages 87-113 (2006).
5. Cypher, A., Smith, D.: KidSim: End User Programming of Simulations. In: CHI'95: Human Factors in Computing Systems, Denver, CO, May 7-11, 27-34 (1995)
6. Fekete, J.-D.: The InfoVis Toolkit. Proc. IEEE InfoVis, pages 167–174 (2004)
7. Google Visualization API, http://code.google.com/apis/visualization/documentation/gallery.html
8. Heer,, J., Card, S. K., Landay, J. A.: Prefuse: a toolkit for interactive information visualization. Proc. ACM CHI, pages 421–430 (2005)
9. Kuhail, M. A.: Custom formula-based visualizations for Savvy Designers. Ph.D. thesis, IT-University of Copenhagen (2012).
10. Kuhail, M. A., Lauesen, S., Pantazos, K.: The Inspector: A Cognitive Artefact for Visual Mapping. In: IVAPP 2013 proceedings (2013)
11. Lewis, C: NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery. In: Glinert, E. (ed.) Visual Programming Environments: Paradigms and Systems, IEEE CS Press, Los Alamitos, California, 526-546 (1990)
12. Myers, B., Hudson, S. E., Pausch, R.: Past, present and future of user interface software tools. ACM Transaction on Computer-Human Interaction, Vol. 7, No. 1, pp. 3-28 (2000).
13. Norman, D.: The psychology of everyday things, Basic Books, New York (1988)
14. Omniscope | Visokio, http://www.visokio.com/omniscope.
15. Pantazos, K., Lauesen, S.: Constructing Visualizations with InfoVis Tools - An Evaluation from a user Perspective. In: GRAPP/IVAPP 2012: 731-736 (2012)
16. Pantazos, K., Kuhail, M., Lauesen, S., and Xu, S.: Constructing Custom Visualizations with a Development Environment. In: Proc. of Visualization and Data Analysis (2013)
17. Plaisant, C., Heller, D., Li, J., Shneiderman, B., Mushlin, R., Karat, J.: Visualizing medical records with lifelines. CHI 98 conference on Human factors in computing systems, CHI '98, pages 28–29, New York, NY, USA (1998)
18. Spotfire, http://spotfire.tibco.com/
19. Stolte, C., Tang, D., Hanrahan, P.: Polaris: a system for query, analysis, and visualization of multidimensional databases. Commun. ACM 51, 11, 75-84 (November 2008)
20. Tableau, http://www.tableausoftware.com/
21. Viegas, F. B., Wattenberg, M., van Ham, F., Kriss, J., McKeon, M.: ManyEyes: a Site for Visualization at Internet Scale. Visualization and Computer Graphics, IEEE Transactions on, vol.13, no.6, pp.1121-1128 (Nov.-Dec. 2007)
22. Weaver, C. E.: Building Highly-Coordinated Visualizations in Improvise. INFOVIS 2004. IEEE Symposium on Information Visualization (2004)