Extending Applications with Visualization

Soren Lauesen, Mohammad Kuhail, Kostas Pandazos, Shangjin Xu and Mads B. Andersen

Abstract—It is hard to add visualization and interaction to existing applications. The data may be in the database, but users cannot see it in a convenient way. Often different departments need to see the data in their own way. You may of course ask the supplier of the system to develop such extensions, but this is expensive and time consuming. The ideal would be that local non-programmer staff could do it, but current tools are either hard to integrate with daily production applications, require too much programming or lack visualization.

This paper presents *uVis*, a tool that allows local staff to build an additional user interface. They combine simple components in the traditional drag-drop-set-property way, but contrary to current tools, any property may be a formula that combines data from other visual components and from relational databases. The formulas look rather simple, yet they are able to combine simple components into traditional as well as new visualizations and provide interaction. Uvis has its own integrated development environment (IDE) that helps local staff write formulas, see database tables, locate errors, etc.

At present uVis performs reasonably. As an example, it takes 0.8 seconds to retrieve 10,000 data records and show them as 10,000 visual components. We are currently testing the usability of the tool. Preliminary results indicate that it is roughly as easy to learn as spreadsheets.

Index Terms—Data visualization, database, interaction, user interface, end-user development, toolkits.

1 The Problem and the Solution

Many large IT applications lack integrated data visualization. They show data only in textual form as simple database fields and tables of data records. Further they are hard to customize so that individual user departments see data in different ways. A good example is electronic health records, where different medical specialties (intensive care, hearth surgery, psychiatry, etc.) use the same database but need to see data in their own way. Other examples are ERP systems, financial systems, project management and resource allocation.

In order to create an extension of an application for daily production work, it must comprise a combination of textual presentation, advanced graphical visualization, interaction and data entry. In other words, it must provide a complete additional user interface.

One way to deal with the problem is to ask the system provider to deliver local extensions with visualization. However, this is very expensive, particularly if the extension is to be part of normal system maintenance. In addition it is hard to get the user interface right and easy to use; it requires several iterations, which adds further to the cost.

The ideal would be that local staff with some IT expertise could make this kind of extensions in cooperation with local end-users. We will use the term *local designer* to mean this local IT expertise. They don't have to be professional programmers, but they need some IT expertise such as understanding database tables and understanding formulas that compute values.

This paper presents a tool (uVis) that supports local designers. As many other tools, uVis is based on dragging components to the screen to be designed, and setting their properties (the drag-drop-set-property principle). However, any property can be a formula that computes a value based on data in other components as well as data in relational databases. The formulas look like spreadsheet formulas but are able to combine simple components into existing or new visualizations. The formulas also provide interaction and data entry without further programming.

Figure 1 shows a visualization screen developed locally with uVis. During bronchoscopy, the surgeon marks spots where he takes a biopsy. When he later gets the lab results, he changes the color of the marks accordingly. Notice that the bronchia are upside-down, because this is the way the surgeon sees the patient during bronchoscopy. The screen is a simple visualization, but it requires interaction and database updates. Present tools require programming to do this.

Figure 2 is a more complex visualization made locally with uVis (inspired by Lifelines [11]). It gives an overview of a patient's medical record from birth to some time into the future. This specific patient has several chronic diseases and has been treated by a general practitioner for years. On the 16th of January, the patient was hospitalized with a serious infection. The hospital stopped most of the patient's medication and ordered something else.

All the data is extracted from the existing database. The data is shown as color (e.g. the color of the note icons), as shape and as position and size (e.g. the medication boxes). The time scale at the top consists of several zoom intervals, and the end-user can zoom by dragging points in time from one interval to another. The end-user can click on the various objects to see details.

2 Related Work

Existing tools vary in many ways. Some need programming expertise, some don't. Some integrate well with existing data sources, some need programming to do so. Some allow you to make new visualizations, some only provide predefined ones.

Myers et al. [7] gave an excellent overview of user interface tools in 2000 and explained why drag-and-drop tools were more successful than program-based tools. What is the situation today? A recent study [10] showed that local designers (called "savvy users") still need better tools and more attention from the information visualization community. Below we will give an overview of tools today.

- *Soren Lauesen, E-Mail: slauesen@itu.dk.*
- *Mohammad A. Kuhail, E-Mail: moak@itu.dk*
- *Kostas Pandazos, E-mail: kopa@itu.dk*
- *Shangjin Xu, E-mail: xush@itu.dk*
- *Mads B. Andersen: mban@itu.dk*
- *All authors are with the IT-University of Copenhagen.*

- *Manuscript received . . .*

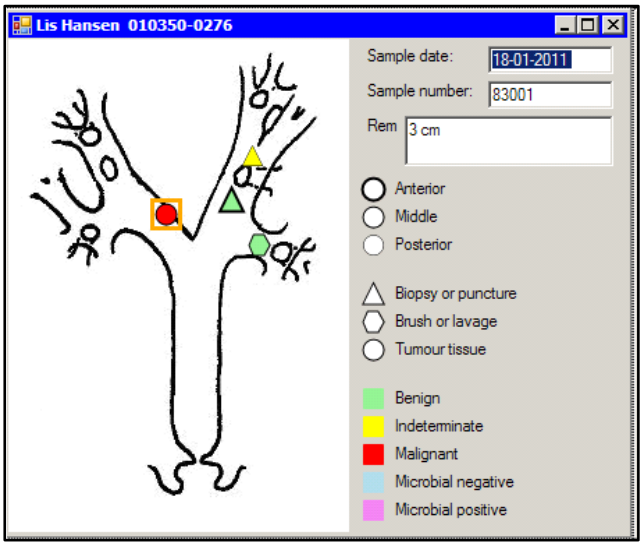

Figure 1. Surgeon marks biopsy location during bronchoscopy. Later he changes the color to reflect the lab results.

2.1 Standard Graphics

Standard graphical presentations such as pie charts and bar charts are provided in Excel [6], Google Spreadsheets [4], [2] and others. They don't require programming skills and widely used.

However, designers have no way of creating new visualizations beyond what is predefined. The tools are stand-alone systems, which mean that users have to manually transfer existing data to the tool. Further you have little interaction with the data and cannot feed data back to the existing application without programming.

2.2 Data Analytical Tools

These tools integrate well with existing data and help users explore the data. They don't require programming skills.

Tableau [13] is a commercial data analytical tool based on Polaris [15]. Users can visually compare ordinal and quantitative fields of relational tables. Tableau also employs interaction techniques such as zooming, and filtering. Spotfire [14] and Omniscope [9] are in the same category as Tableau.

However, with these tools designers have no way of creating new visualizations beyond what is predefined. You cannot integrate these systems with existing applications and you cannot feed data back to the existing applications without programming.

Improvise [17] is more powerful than the other tools in this category. Designers can combine visualizations and add simple controls. Expressions and data used for the dialog can be stored in a repository "behind" the visualization. It is to some extent a drag-drop-set property tool, but the properties are not general formulas. The queries, for instance, seem to be defined in the repository. Improvise cannot update source data and is not intended for close integration with daily production systems.

2.3 Graphics APIs

Several programming languages provide general purpose graphics libraries (APIs), such as GDI+ and Java 2D. They provide basic components such as line, polygon, and ellipse. By means of a program you can make them create any visualization and bind to any data.

However, to accomplish this, you must be a professional programmer.

2.4 Visualization Toolkits

These toolkits allow you to construct traditional and new visualizations by means of a special programming language (sometimes called a domain-specific language).

Protovis [1] uses a declarative programming language based on Javascript. At run-time, the language generates visible components, called marks. Each mark has a piece of data that it transforms to visual properties by means of functions (similar to formulas). An event can be attached to a mark as another function. All data are embedded in the Protovis program as lists of constants, possibly nested. Each mark must have such a list and Protovis generates a mark instance for each element in the list. To visualize existing data, other tools are needed to transform the data to lists of constants. The successor of Protovis is D3 where visualizations are created with CSS3, HTML5 and SVG.

Prefuse [5] is the predecessor of Protovis. Prefuse can access relational databases and query the tables with SQL statements. The result is stored as a table structure and visualized.

InfoVis [3] is in the same category as Protovis, but uses another kind of visible components.

These toolkits don't use a drag-and-drop approach but require a program that generates the screens. They are not easy to integrate with existing relational data and you cannot feed data back to the existing application without programming.

2.5 Drag-Drop-Set-Property Environments

Current industry tools such as Microsoft Visual Studio, Eclipse and NetBeans allow designers to construct user screens with drag-and-drop of text boxes, buttons and other components. For each component the designer sets size, position, color and other properties to a constant. This approach can quickly generate a mockup that looks right, and the designer can set some dummy values in each text box.

In simple cases the designer can also bind a database table to a component, for instance a combo box. However, when a join is required or the screen has to interact, "programming behind" is needed and local designers have to let professional programmers do this. Making an advanced visualization with these tools requires a lot of programming, and according to our own industrial experience even professional programmers find it hard.

2.6 Summary

When we look at tools for a local designer, the tools above have gaps in one or more areas. They may require too much programming, they may not allow construction of new visualizations, they may be hard to integrate with existing applications or they may not use the drag-and-drop approach that has proven successful with local designers [7].

Uvis tries to fill all of these gaps with one simple principle, but it has been a challenge.

3 Design Rationale for uVis

Uvis uses the drag-drop-set-property principle, but the properties are not constants, but formulas that compute a value. The challenge is to make the formulas so powerful that simple components can combine to traditional as well as new visualizations. The formulas must also be able to access databases and provide interaction. At the same time, they must be easy to understand, somehow like spreadsheet formulas.

Why have we chosen to access databases directly? The short answer is that it gives the designer sufficient freedom to experiment with the many ways to present the raw data. Today many systems are service oriented (SOA) and don't get their data through SQL, but through XML-services that retrieve data in a predefined way, corresponding to a specific database query. Our industrial experience is that even slight changes to the user interface require a different database query and this requires that the programmers develop a new

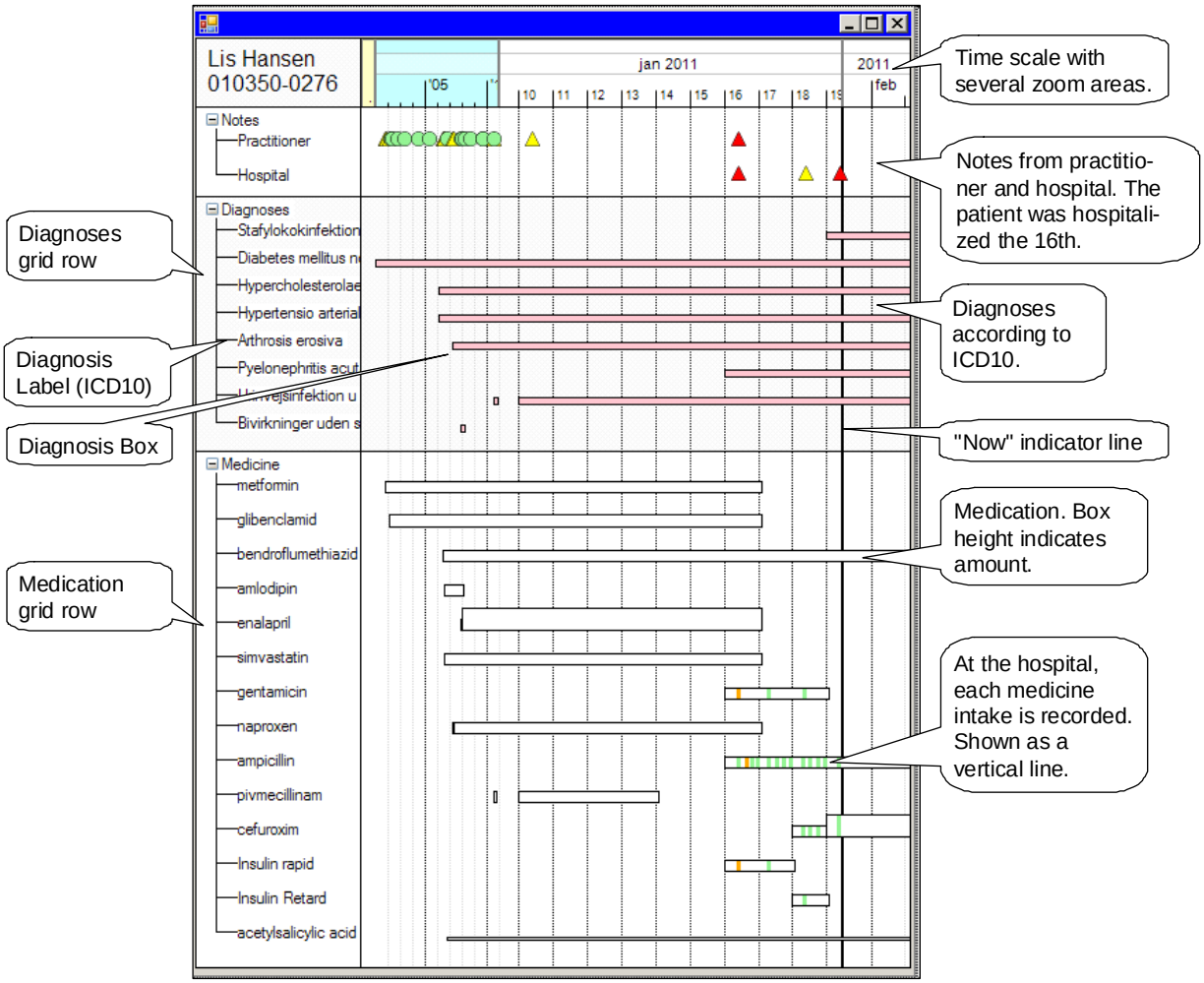

Figure 2. Overview of a patient's medical record from birth to some time into the future.

service. There are various ways to get around the problem, but we will not discuss them here.

VALUE FORMULAS

By means of formulas, the designer can tell uVis to compute a value based on other components and database contents. The value can for instance become the position or color of a component. It can also be a list of data rows to be used for generating many components (see more in section 3.2).

EVENT-HANDLER FORMULAS

In order to handle interaction such as clicking or typing, it is not sufficient to compute values. Some action must be performed too, for instance setting a value, opening a form or committing a database transaction. This is handled by properties that are event handlers. A component may for instance have a Click property. Whenever the user clicks the component, uVis carries out the Click property's formula. It may for instance open a form.

Depending on the application, it may be necessary to perform actions beyond the built-in uVis ones, 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. Usually it will be a programmer. The designer can then call the method through an event handler. This is "programming behind", but it is used far less than in the traditional approaches.

3.1 Addressing visual components and database stuff

The formula language should be of "spreadsheet complexity", but there is an inherent problem here. A spreadsheet formula can only address cells in the spreadsheet and it uses constant row and cell numbers to do so. In contrast, uVis must be able to address several kinds of entities given by the surrounding software:

1. Forms (a special kind of component). The designer defines the form name.
2. Components on forms (also called controls or widgets). The designer defines the component name.
3. Properties of components. The operating system (MS Windows) has defined most of the property names.
4. Database tables. The database has defined the name.
5. Database fields. The database has defined the name.
6. Relationships between database tables. The designer can define the name.

In a visualization such as Figure 2, a formula in one component must refer to properties in other components. As an example, the height of the diagnoses grid row must refer to the bottom of the last diagnosis label in order to give the grid row the correct height. Since the bottom of the last diagnosis label is also defined by a formula, uVis must be able to deal with a network of formulas in the same way as spreadsheet formulas. And in the same way as spreadsheets, it has to deal with potential circular references [12]. However, in our case it is

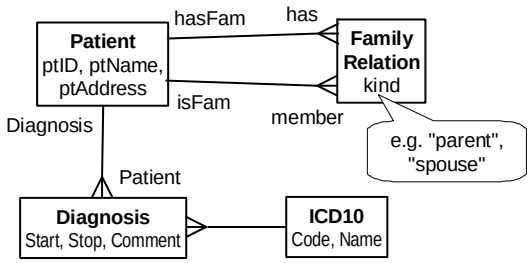

Figure 3. Entity-relationship diagram of some of the tables.

much more complex because entities are created dynamically and addressing may use several levels of names.

How to address all of these entities in a uniform way? The tradition in programming is the dot-notation: A.B.C means walk from A to B to C. This works well when walking from a form to a component on the form, and then to a property in the component.

However, addressing database entities follows the SQL paradigm. Figure 3 shows an entity-relation diagram of some of the tables behind the life line. The crow's foot from Patient to Diagnosis shows that each data row in the Patient table is related to several data rows in the Diagnosis table. To address *ptName* in the Patient table and *Start* in the related Diagnosis table for a specific patient, programmers would have to write something like this SQL query:

```
SELECT ptName, Start FROM Patient JOIN Diagnosis
ON Patient.ptID = Diagnosis.ptID WHERE Patient.ptID = 52;
```

Some real programming is involved too because the "52" is the key for the patient we want. It was inserted into the query string by a program based on end-user input.

This is a completely different paradigm and it is hard to see how it can be combined with the programming paradigm in a simple way. (This is part of the problem called *object-relational impedance mismatch* [8]).

Uvis deals with it by allowing a formula to walk from one table to another along a crow's foot. Whether we walk to one or many rows depends on the direction of the crow's foot.

Walking to many rows generates a list of rows and uVis then creates a visual component for each row. As a result each component is connected to a row in a table or a query.

Using the walk-principle, the SELECT statement above will look like this in a uVis formula:

```
Patient -< Diagnosis Where Patient.ptID = txtID.Text
```

The -< symbolizes a one-to-many crow's foot. It tells uVis to walk from a patient row along the crow's foot the related Diagnosis rows. The Where clause is similar to the SQL version, but the criterion 52 is now a reference to a visual component, the textbox txtID. Uvis automatically gets the value the user has typed in this box (Text), inserts it in the SQL statement and sends it to the database. The Select-part has disappeared. Uvis generates it automatically based on what the formulas refer to.

The uVis queries support the usual Sql clauses such as Where, Order By, Group By and Top.

In the example there is only one crow's foot between the two tables. However, in general there may be many. Figure 3 shows an example. A row in FamilyRelation specifies that two patients are related to each other. One of the crow's feet points to one of these patients and the other points to the other patient. Clearly we somehow need to name the crow's feet. Uvis provides two names for each crow's foot: one when we come from the many-end and one when we come from the one-end. When there is only one crow's foot between two tables, the crow's-foot names can be the table names. Figure 3 shows this for Patient -< Diagnosis.

In section 3.2 we will show an example of walking along named crow's feet.

Next, let us see how a formula can walk between visible components and database tables. Figure 4 shows an example where a DiagnosisBox aligns its Bottom to the bottom of the related DiagnosisLabel. A diagnosis label is connected to an ICD10 row. ICD-10 is the international classification of diseases. Each row contains a diagnosis code and a diagnosis name (e.g. M154, Arthrosis erosiva).

The Bottom formula starts in the diagnosis box itself (Me), walks to the connected Diagnosis row in the database, from there along the crow's foot to the related ICD10 row in the database, and from there

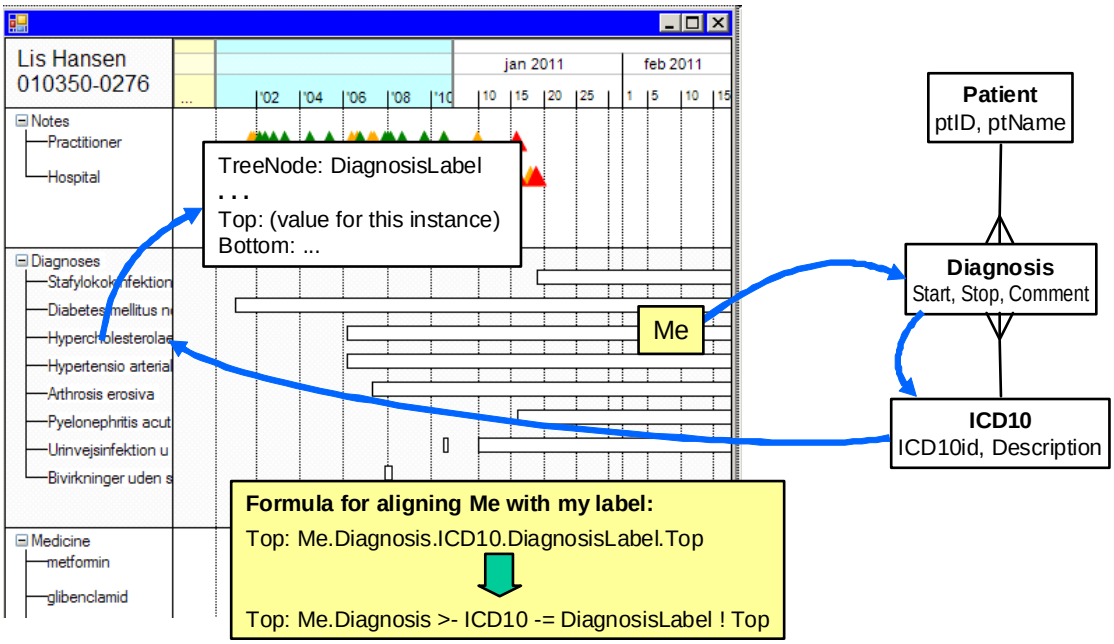

Figure 4. A formula that walks from a visual component to a data row to another data row, to a component and to the component's property.

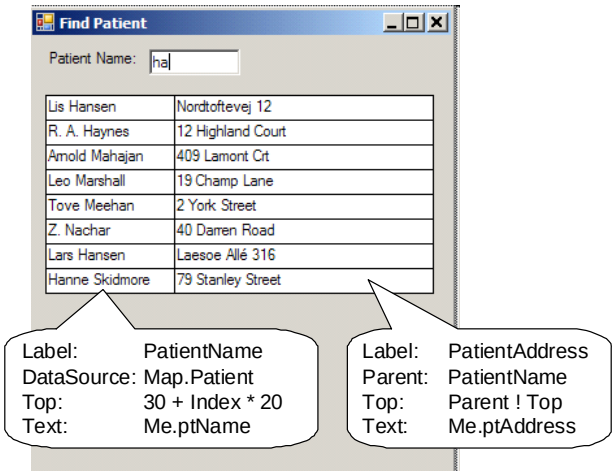

Figure 5. Label PatientName repeats itself for each patient row. Label PatientAddress repeats itself for each label PatientName.

to the connected diagnosis label. Then it gets the Bottom position of this label. With the dot-notation it would look like this:

```
Box:     DiagnosisBox
Bottom:  Me.Diagnosis.ICD10.DiagnosisLabel.Bottom
```

Early experiments showed that this was hard to read. What was visual component stuff and what was database stuff? It was much easier to read if we used different "dot" operators to show what is what. If the designer types the formula as above, the uVis compiler will show it as this:

```
Bottom: Me.Diagnosis >- ICD10 -= DiagnosisLabel ! Bottom
```

The dot (.) tells uVis to walk from the component to the connected Diagnosis row. The >- tells uVis to walk from the Diagnosis row along the crow's foot to an ICD10 row. The -= tells uVis to walk from a row to the connected visual component, DiagnosisLabel. Finally the ! (bang) tells uVis to look for a property in the component. With some experience, the designer can immediately see whether this is what he intended.

The various dot-operators also allow us to resolve name conflicts. Some of the entities have names we cannot change. We may for instance encounter a database field called Top and this conflicts with the name of the built-in Top property. The designer can tell uVis what he means by using dot (.) or bang (!). In some cases we need a prefix to tell uVis what we talk about. As an example, we may want to refer to the database table Top. To resolve the ambiguity, the designer has to use a Map prefix and write Map.Top.

Some components offer properties with complex functionality. The timeScale in Figure 4 is an example. It has a property called Hpos (horizontal position) that converts a point in time to a horizontal pixel position. The DiagnosisBox calls Hpos to convert the start time from the Diagnosis row to its own Left position. In the same way it converts the stop time from the row to its Right position:

```
Box:     DiagnosisBox
Left:    timeScale ! Hpos(Me.Start)
Right:   timeScale ! Hpos(Me.Stop)
```

3.2 Repeated components

A visualization will usually have repeated components. Visualization tools such as Protovis generate them based on lists of data (arrays). Uvis does the same, but the difference is that in uVis the generation of the list - including the joins - is a simple "walk" with a formula, while in Protovis it is a separate, complex program.

We have observed that designers with low IT skills find it hard to understand the relation between a list of data and the repeated components that calculate their own individual property values. We strive for "spreadsheet complexity" so how does a spreadsheet handle the repetition? It doesn't. The user has to copy cells manually as needed. The formula in the cell can use relative addressing and in that way generate different values for different cells. The manual copying is not suitable for the kind of applications we aim at, so uVis generate lists in spite of the mental barrier they create.

So how does uVis generate repeated components? All components have a property called DataSource. It has a formula that calculates a list of data rows. Uvis will then generate a component instance for each row and connect the component instance to the row. We will illustrate the principle with examples from Figure 5.

REPEAT ACCORDING TO DATABASE CONTENTS

The FindPatient screen has a list of patients. Each row in the list consists of two textboxes, the patient name and the patient address. We will first make the patient name repeat itself. In the basic version its DataSource and other key properties look like this:

```
Label:      PatientName
DataSource: Map.Patient
Top:        30 + Index * 20
Text:       Me.ptName
```

The data source is the Patient table, so the list of data rows comprises all patients in the database. Uvis will generate a PatientName label for each of them. The first component will have Index=0 and its Top position in pixels will thus be 30. The next component has Index=1 and Top position 50. Each component will show the patient name (ptName) from the connected row.

Formulas may refer to list components by index. The formula PatientName[2] will refer to the third component in the list. A component can refer to its sibling with Me[Index-1], etc.

The real version needs a search criterion so that it doesn't show 20,000 patients. The search criterion is a textbox, *Crit*, and we want wild-card search on the patient name. Further we want to show at most 20 patients. This is accomplished with this DataSource property:

```
DataSource:   Map.Patient Top 20
              Where ptName Like "%" & Crit ! Text & "%"
```

If the end user has typed for instance "ha" in the Crit box, uVis will generate this Where clause:

```
Where ptName Like "%ha%"
```

As a result, the database will retrieve patients with a name that contains "ha". Due to the Top 20 clause, it will at most retrieve 20 patients.

We will now give an example that is not for uVis beginners, but shows the expressiveness of the walk principle. Assume that we want a list of all patients related to a selected patient according to Figure 3. We would start in the patient row, walk along *hasFam* to all related *FamilyRelation* rows, and for each of these along *member* to the related patient. In uVis notation it looks like this:

```
Patient -< hasFam >- member Where Patient.ptID = txtID.Text
```

The resulting list of rows has fields from the start patient, from FamilyRelation (hasFam) and from the target patient (member). We can address the fields by means of the relation name. As an example we could make a list of the related patients with this label component:

```
Label:        Relatives
DataSource:   Patient -< hasFam >- member
              Where Patient.ptID = txtID.Text
Text:         Patient.ptName & " has " & member.ptName &
              " as " & hasFam.kind
Top:          Index * 15
```

(We use the Visual Basic notation & for string concatenation.) The result would look like this:

John Smith has Alice Smith as child
John Smith has Sofia Auriel as parent

REPEAT ACCORDING TO ANOTHER COMPONENT

Another way to repeat a component is to specify that it has a repeating component as its *parent*. Uvis will then generate a child component for each parent component. In order to generate the patient addresses, the designer creates a patient address label and sets its *parent* property in this way:

```
Label:    PatientAddress
Parent:   PatientName
Top:      Parent ! Top
Text:     ptAddress     ' A Parent prefix is not needed
```

There will now be one PatientAddress for each PatientName. Further, the Top formula refers to the parent's top and in that way aligns the patient address to the patient name.

CREATE BUNDLES OF COMPONENTS

Sometimes we don't need a single component for each parent component, but a whole bundle of components. This is the case with the diagnosis boxes in Figure 4. For each diagnosis label, there is a bundle of diagnosis boxes because the patient may get the same disease several times. We can use these formulas to generate the bundles:

```
Box:      DiagnosisBox
Parent:   DiagnosisLabel
DataSource: Parent -< Diagnosis
```

Each DiagnosisLabel is connected to an ICD10 row. Uvis will walk to all diagnosis rows that relate to this ICD10 row and generate a list of them. It then creates a DiagnosisBox for each row in the list. The result is a bundle of components for each DiagnosisLabel.

The parent concept works in many levels. As an example, the colored intake bars at the bottom of Figure 2 are made by walking from each medication box to the related intakes.

3.3 Interaction

The end-user searches and selects a patient from the FindPatient form (Figure 5), then clicks the patient name to open the patient form (the lifeline). This involves two events: One when the user types into the *Crit* textbox and one when he clicks the patient name. The *Crit* textbox needs this event handler:

```
Label:    Crit
TextChanged: Refresh( )
```

TextChanged is an event provided by MS Windows. It is triggered whenever the textbox changes its value, e.g. when the user types a character, but not when he presses an arrow key. We have specified that when the event occurs, uVis must refresh all open forms, i.e. recompute all formulas, requery the database if the SQL statement has changed, and update the screen accordingly. The result is a live search where the list of patients changes as the user types.

If the database is heavily loaded, the response may be slow. The designer may then decide to refresh the screen only when the end-user has finished typing. Another built-in event, FocusLost, triggers this. So the designer would have to write this instead:

```
FocusLost: Refresh( )
```

Opening the patient form is handled by the Click event in the patient name:

```
Label:    PatientName
Click:    OpenForm("PatientForm", Me.patientID)
```

OpenForm has one or more parameters. The first is the name of the form, the rest are parameters transferred to the open form. In this case there is one parameter, the ID of the patient clicked. The patient form has a data source that gets a patient with the ID passed as a parameter. It looks like this:

```
Form:     LifeLine
DataSource: PatientName Where ptID = Param[0]
```

An event handler can also set properties that don't have a formula. As an example the designer could define that F6 expands and collapses the medicine tree. The medicine tree has a Collapsed property that defines whether the tree is collapsed or not. All we have to do is to toggle Collapsed and then call refresh. In Visual Basic style, the event handler looks like this:

```
KeyDown: If e.KeyCode = Keys.F6 Then _
             MedTree ! Collapsed = Not MedTree ! Collapsed,
             Refresh( )
```

3.4 Formula Language

We have based the formula language on Visual Basic because it is widely known among the designers we aim at. This choice means that the designer can type names without caring about upper/lower case, and uVis gives feedback by correcting them to the proper case. It also means that "=" means assignment or comparison depending on the context, and other trivial details.

Most of the functions available in Visual Basic are also available in uVis, for instance Sin( ), ToDay( ), Format( ) and Choose. As an example, the colors and shapes of the Note icons are generated with these formulas:

```
BackColor: Choose(Me.noteWarningLevel,  Color.LightGreen,
             Color.Yellow, Color.Red)
```

We have introduced some additional functions and operators to simplify life for the local designer. One is called *OnError*. It makes it easy to handle missing or inconsistent data in the database. As an example, diagnoses have Stop time = null until the disease is gone. As a result the formula for calculating the Right position of a diagnosis box would fail. Using *OnError* the designer writes this formula:

```
Right: timeScale ! Hpos(Me.Stop OnError Date()+30)
```

When Stop is null, OnError catches the error and makes Right correspond to a point in time 30 days ahead. This is why most of the diagnoses in Figure 4 end at the right border of the screen.

3.5 Uvis Architecture

Uvis provides two kinds of visual components: the ones provided by Windows Forms and some implemented with the more basic GDI interface. There are two reasons for the GDI components: (1) Windows Forms doesn't provide components needed for advanced visualization, such as pie slices and line curves. (2) Windows Forms is slow when the form contains many components, and when we pass 1000 components response time becomes intolerable. However, to the designer the two kinds of components look the same and have the same properties, Parent, DataSource, Top, Height, etc.

There have been many surprises working with Windows Forms on the level we do. As an example, the designer puts a component on a panel with the formula *Left: 100*. What happens when we scroll the panel for instance 50 pixels to the left? The designer would expect that Left is still 100. Not so in Forms. Left becomes 50. The uVis kernel has to compensate for this. We also noted that designers often

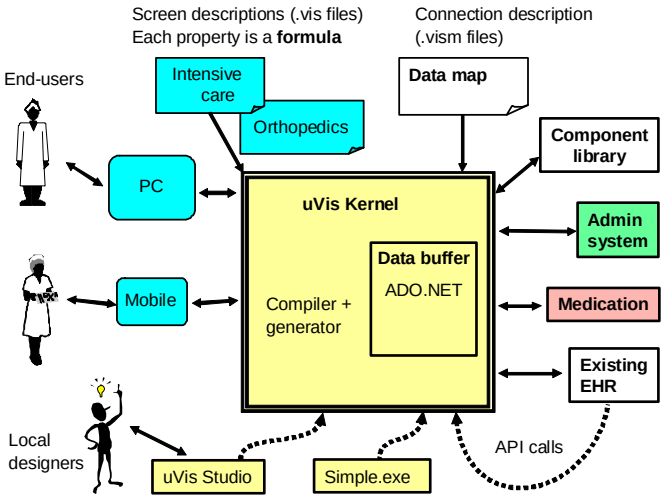

Figure 6. Uvis architecture.

tried to specify formulas for Right or Bottom, but Forms doesn't support this. We made the kernel support it and transform the values to what Forms allow.

We have added some visual components our self. Except for the time scale, they are all rather simple.

Figure 6 gives an overview of uVis and its surroundings. A uVis application consists of a folder with a file for each form (extension .vis) and a file that specifies the connection to databases and the relations between tables (extension .vism). These files are in a traditional text format and may be edited with notepad. The vis-files contain formulas similar to the ones we have shown above. The folder may also contain icons, pictures and other resources used by the forms.

The uVis kernel is a set of API's that can be called from any .NET program, in that way integrating uVis with the program. The API's can open the vis and vism files and run the application. Other API's provide creation of components, setting of formulas, access to error messages, etc.

We provide two stand-alone programs that call the uVis kernel. One is a development environment, uVis Studio. It uses the APIs for setting formulas, etc. The other is a simple program that allows the end-user to select an application folder, open the initial form and run the application.

The code for the various visual components is kept in a library folder. Adding a new kind of component is basically a matter of programming it and adding it to the folder.

3.6 Uvis Studio

In principle you could develop a uVis application with notepad, but it is hard to remember the names of components, properties, database tables, etc. Further, you don't see the changed screens immediately, but have to save the file and open it with the Simple program. A development environment helps here.

Figure 7 shows the uVis development environment (uVis Studio) in action. The forms you design are floating on the desktop in the same way as they will do in the final application. The Studio is a separate form with toolbox panel, property grid, etc. as in other development environments. When you type or edit a property formula, Studio provides Intellisense, meaning that it tells you which property names, field names, etc. you can use at this point. As soon as you have edited a formula, Studio refreshes the forms you design so that you see the consequences of your change.

The Studio can be in several interaction modes. In end-user mode, you can use the forms as the end user will do, click on buttons to open additional forms, scroll, enter data, etc. In design mode you

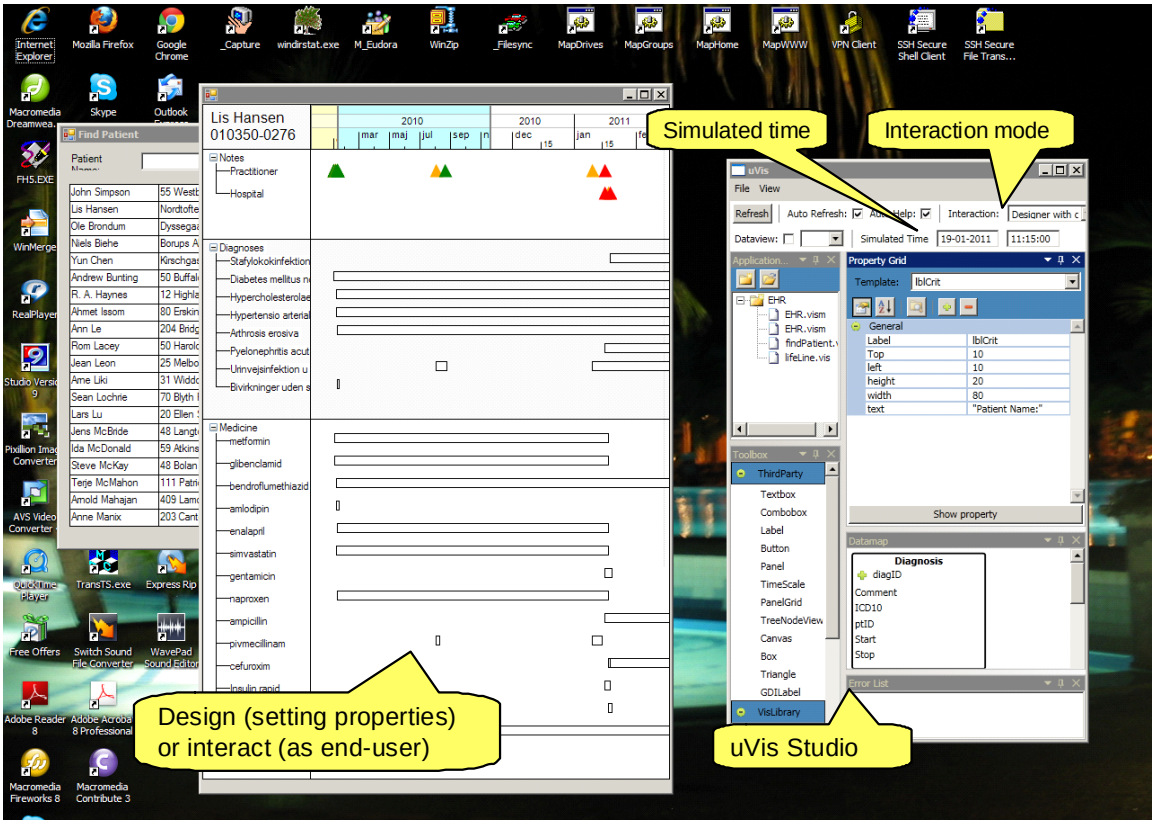

Figure 7. The forms being designed (at the left) and uVis Studio (at the right).

click on components to see and edit their properties through Studio, drag and drop new components to the form you design, etc. There is also a mixed mode where most interaction works in end-user mode, but when you use Control+click, you see and edit properties.

When you work directly with data from a database, it is crucial to get an overview of the existing data. Uvis Studio provides a panel with boxes and crow's feet for all the tables, similar to Figure 3. You can detach the panel and enlarge it to get a better overview. You can also click on a table box and see the real data as a data grid.

We have paid some attention to the test situation versus the deployed version. Whether it is one situation or the other depends on the data-map file you provide. In a typical test situation, you connect to a test database (with anonymized data in case of a health record system). You may also want to use simulated time rather than real time so that a form such as Figure 4 looks the same today as when you do some testing next week.

When you deploy the application for real use, you provide a data map file that connects to real data and uses real time.

4 Experience and Quality Aspects

We have developed many small applications in an experimental way with uVis (see some examples on the last page). Most of them became surprisingly simple as we learned to utilize the power of the formula language. In particular, the spreadsheet-like style was a great help. As an example, a simple formula could express that the height of a tree node was the difference between the top of the first child node and the bottom of the last child node, and these in turn depended on the top of the tree node. Sometimes the formulas were so complex that we thought there would be a cyclical reference, yet uVis easily computed them.

Another simplification was interaction. With the traditional .NET approach, components (controls) had to respond to many kinds of events, for instance color changed and border changed, in addition to the real user events such as click and key down. This makes it very hard to get an overview of who calls whom and when. With uVis, we only had to deal with the real user events and then ask uVis to refresh everything. That this works in practice depends of course on refresh being fast.

SPEED

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 when 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.

The table below shows the time to query a local database for *n* rows, each of 1700 bytes, create a box for each row and show it on the screen. The table shows figures for using simple GDI boxes versus MS Forms' GUI controls.

As the table shows, with simple GDI boxes, uVis can query and show 10,000 boxes in 0.8 s. With GUI controls the time for 4,000 boxes is 8.1 s. In both cases there is a base time of around 50 ms to start a query.

Number of GDI boxes	Total time	Database part	Time per box	Database per box
1,000	135 ms	67 ms	135 µs	67 µs
10,000	800 ms	275 ms	80 µs	27 µs
Number of GUI boxes				
1,000	650 ms	67 ms		
2,000	2100 ms	91 ms		
4,000	8100 ms	126 ms		

However, we have made one important optimization. We minimize the number of queries. A query base-time of 50 ms easily makes the system slow. As an example, if each line of medication in Figure 2 (one bundle of components) used its own query, we would use 14 queries to show the medications, amounting to 0.7 s just for this. The total time to show the lifeline in Figure 2 would be several seconds. Instead we make one query to get all the medications, one to get all the diagnoses, etc. As a result we can show the lifeline in around 0.5 s. Refreshing is much faster because we don't have to requery the database.

USABILITY

Usability is of course an issue when we aim at non-programmer designers. We are currently running usability tests to see how difficult the tool is to learn. At the same time we develop tutorials and other support information. Preliminary results indicate that it is roughly as easy to learn as spreadsheets. Programmers learn to use it in half an hour, staff with some MS Access background has to experiment for a couple of hours, some novices also learn it in a few hours and some novices will probably never figure out.

SECURITY

There are several security aspects if we want to deploy a uVis application as an extension of an existing application for daily work. One aspect is whether the formulas can crash the system so that other users are harmed. This is easy; the formulas have no access to other parts of the system. However, if the visualization draws heavily on the database, other users may experience longer response times. We see only one way to prevent it. Measure the processor capacity used when the uVis application runs on test data rather than production data. The easy change between data maps for testing and data maps for production supports this.

Another aspect is access rights - whether the end user is allowed to see or update the data in question. There are several ways to do it. One is to rely on access rights implemented in the database, another is to allow access only through views, and a third one is to use protected and/or encrypted data maps and vis-files. This has to be worked out for the specific application.

5 Conclusion

We have shown that it is possible to construct a tool (uVis) that allows local non-programmers to add visualization and interaction to traditional applications that use a relational database. Uvis provides a solution to the object-relational impedance mismatch, because formulas can "walk" across visual components and relational tables in a uniform way.

In addition the tool combines the best of existing tools:

1. an integrated development environment (IDE) with drag-drop-set-property of visual components,
2. properties that are formulas and can combine data from other visual components and from relational databases,
3. formulas that can combine simple components into traditional as well as new visualizations,
4. interaction and database updates,
5. no need to "program behind".

Further, the tool has adequate performance and usability similar to spreadsheets.

REFERENCES

[1] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. IEEE Trans. Vis. and Comp. Graphics, 15(6):1121–1128, 2009

[2] M. Bostock, V. Ogievetsky, and J. Heer, "D³ Data-Driven Documents," *Visualization and Computer Graphics*, IEEE Transactions on , vol.17, no.12, pp.2301-2309, Dec. 2011

[3] J.-D. Fekete. "The InfoVis Toolkit". *In Proc. IEEE InfoVis*, pages 167–174, 2004.

[4] Google Visualization API. http://code.google.com/apis/visualization/documentation/gallery.html, February 2012.

[5] J. Heer, S. K. Card, and J. A. Landay. "Prefuse: a toolkit for interactive information visualization". *In Proc. ACM CHI*, pages 421–430, 2005

[6] Microsoft Excel. http://office.microsoft.com/en-us/excel/, February 2012.

[7] B. Myers, S. E. Hudson and R. Pausch: Past, present and future of user interface software tools. ACM Transaction on Computer-Human Interaction, Vol. 7, No. 1, March 2000, pp. 3-28.

[8] Object-relational impedance mismatch, Wikipedia. March 2012. http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch

[9] Omniscope | Visokio. http://www.visokio.com/omniscope, February 2011.

[10] K. Pantazos and S. Lauesen, "Constructing Visualizations with InfoVis Tools - An Evaluation From A User Perspective", *In Proceedings of the International Conference on Information Visualization Theory and Applications*, 2012.

[11] C. Plaisant, D. Heller, J. Li, B. Shneiderman, R. Mushlin, and J. Karat. Visualizing medical records with lifelines. *In CHI 98 conference summary on Human factors in computing systems, CHI '98*, pages 28–29, New York, NY, USA, 1998. ACM.Processing. http://processing.org, February 2011.

[12] P. Sestoft. "A Spreadsheet Core". IT University Technical Report Series TR-2006-91, ISSN 1600–6100, September 2006.

[13] Tableau. http://www.tableausoftware.com/, February 2011.

[14] Spotfire. http://spotfire.tibco.com/, February 2011.

[15] C. Stolte, D. Tang, and P. Hanrahan.. "Polaris: a system for query, analysis, and visualization of multidimensional databases". *Commun. ACM 51*, 11 (November 2008), 75-84, 2008.

[16] F. B. Viegas, M. Wattenberg,, F. van Ham, J. Kriss and M. McKeon, "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

[17] C. E. Weaver. "Building Highly-Coordinated Visualizations in Improvise," Information Visualization, 2004. *INFOVIS 2004*. IEEE Symposium on , vol., no., pp.159-166, 0-0 0