Lightweight Low-Level Query-Centric User Interface Modeling

Filip Kis
Media Technology and Interaction Design
KTH - Royal Institute of Technology, CSC
Stockholm, Sweden
fkis@kth.se

Cristian Bogdan
Media Technology and Interaction Design
KTH - Royal Institute of Technology, CSC
Stockholm, Sweden
cristi@csc.kth.se

Abstract—Query languages are usually small, powerful and easy to learn, and that motivated us to investigate their role in modeling user interfaces. We explore annotation of template user interface with queries, especially embedded queries, to achieve partial generation of user interface with the ability to generate user interface with interaction styles such as form fill-in and direct manipulation. Our model-driven development approach targets especially highly interactive data intensive applications. While our query-oriented user interface modeling principles have already been applied in an agile development environment, the role of this paper is to present a generalization of query-oriented modeling towards more interaction styles. Our modeling approach is annotation-based, thereby lightweight, and although it operates at a lower level than most current UI modeling approaches, it shows promising potential.

Keywords—query-oriented modeling; model-driven development; user interface modeling

I. INTRODUCTION

User interface development accounts for a large share of the total development cost of any interactive product [1]. An important body of research has been attempting to reduce these costs through model-driven UI development [2] but these modeling approaches are not yet widely adopted in practice. A major cause of this lack of adoption is believed to be the lack of control that the modeler has over the final generated user interface [3]. For this reason, alternatives to fully automatic UI generation were proposed in recent years [4], [5] aiming to combine manual UI design with model-driven automatic inferencing of certain parts of the interactive system.

Producing graphical user interface layout manually allows the involvement of interaction designers for creating the interface, as well as the involvement of customers, users and other stakeholders to give feedback on the user interface at early stages of the product development, when the costs are still low and when changes are still inexpensive to make. The aim of our model-driven development research is to link the manually-produced user interface with the rest of the system in an efficient way that allows to further reduce development costs. This approach poses several challenges:

1) **Low modeler programming skill**: interaction designers and other UI-concerned people will have little programming training. Even if model-driven tools reduce programming, there is often the assumption that the modeler is a person trained in programming, and this is not often the case for interaction designers.

2) **Early UI prototyping and iterations**: it is often desired that the user interface is worked upon early in the project development cycle, and that it should be iteratively improved during the project. This results in a need for the interface to evolve independently from the modeling of its link to the system, or to co-evolve in an uncomplicated manner.

3) **Support for custom widgets**: interaction designers will often require graphical widgets (components) that are specific to the application and not provided by the framework. While such widgets can be developed by programmers in the project, linking them in a model-driven fashion to the rest of the system is a challenge that has not been much addressed since most of the current model-driven approaches use mostly standard interactive widgets. Custom widgets can also include custom layouts or special policies for placing widgets in a UI container.

4) **Support for direct manipulation**: many of the existing examples of model-driven UI development support interaction styles like form fill-in and menu selection. Direct manipulation is supported in most user interface development frameworks and it is therefore expected by users and interaction designers.

Our approach in addressing this challenges is based on two steps:

1) **We support early UI work by the design of template user interfaces** that allow the designers to specify the user interface layout early in the process, with no dependency on the underlying system levels. Template user interfaces are interfaces that represent usually once an interface element that can repeat zero or more times, and are populated with example data rather than data from a functioning system. The example data serves to better illustrate how the template layout is populated, thus being able to get more realistic feedback on the prototyped interface. Working at the
low level of widgets and containers to compose a template user interface is familiar to the interaction designer, and it also differentiates our UI modeling approach from the mainstream of the current approaches, which are high-level, working at either the task [6] or communication [7]–[9] levels. Also at this low level we support advanced, application specific data presentation and interaction by allowing custom widgets and containers to be programmed at this stage. When such widgets make e.g. diagram (geometrical) representations, graphic transformation rules can be added and tested on the example data.

2) We ensure the model-driven linking of the interface with the underlying system via lightweight annotations of the template user interface. The annotation language that we experimented with is query-based, encouraged by our decade-long experience with query languages for large intranet interfaces developed by novice programmers and interaction designers [10], [11]. UI containers are annotated with selection queries, UI widget properties are annotated with query projections. We support drag and drop by allowing widgets that support drag and/or drop interaction to be annotated with semantic annotations. The annotations are hierarchical, i.e. annotations of a UI widget or container are interpreted in the context of the annotations of the embedding UI containers. This hierarchy is natural for the user interface designers.

Due to the lightness of the annotations, and their natural distributions in the UI widget tree, the dependencies between the two steps are weak, allowing for the iteration of the whole process by going back to step 1 and making changes to the manually-developed template user interface, in an iterative prototyping fashion.

The remainder of the paper is structured as follows: Section II presents an example application that is illustrative for our approach, as it requires drag and drop interaction, custom widgets and custom containers, multiple representations of a data model, etc. Section III presents our modeling approach, illustrated on the example applications. The interpretation of the model to render the user interface is then demonstrated in section IV. We evaluate our approach in section V and compare to related approaches in section VI. Finally, we summarize our achievement and present future work in section VII and give a conclusion in section VIII.

II. CONTEXT AND RUNNING EXAMPLE

Our research was motivated by a decade-long experience in building web interface applications [11] where most of development was done in an agile fashion by domain experts who are not skilled in software engineering. This work with amateur programmers and amateur interaction designers was informed by studies of amateurs of various vocations [12], [13]. We provided them with an approach where they could make user interface (web pages with form and link interaction) based on a domain model described in a simple language, by inserting queries (interrogating the domain model) in the HTML code in a hierarchic fashion which we will also detail in this paper. With this tool they were able to build and maintain over a long term a large intranet with 1800 interactive screens, processing over 100 entity types.

While this success was encouraging, we wanted to explore interaction styles beyond form fillin. We thus started to consider more interactive applications, which provide for updating a graphical scene (rather than switching to another page in web manner) and to provide for further interaction styles such as direct manipulation, while maintaining the simple query annotations that allowed untrained domain experts to get involved in user interface production.

We will illustrate our UI modeling approach by way of an interactive example application. The application is a factory production planner (figure 1). The user can plan tasks on the factory production lines by dragging the tasks on the lines (top of figure 1). Each task has a customer name and duration. After the task has been planned (dropped on specific production line) the task gets a start and an end date. At most one task can be planned on the given production line at a given time. Unplanned tasks are shown on the parking

1 Throughout this paper we refer to the production planner (application domain) meaning of task and not the Task Analysis meaning used in some UI modeling approaches.
line (middle of figure 1). Task can be dragged from and to the parking line. Besides the graphical task diagram, the task data is also shown and can be modified in a table (lower part of figure 1). The application is built using the Java Swing GUI framework.

Figure 2 represents the domain model of the running example. While the model is relatively simple with just two entity types, it is complex enough to illustrate our modeling approach, and a highly interactive application can be modeled even with this simple domain model.

III. OUR UI MODELING APPROACH

Our UI modeling approach targets data intensive applications with highly interactive screens representing mostly CRUD operations. We are also targeting the development environments where the users and the designers are involved early in the development process.

Figure 3 represents our UI modeling process. The left side shows the steps where user interface layout is being prototyped by interaction designers and other stakeholders, without the need to link the interface to a domain model. We call this phase layout prototyping, and its result is a UI template populated with some example data. The typical UI widgets are placed on the screen with the total control of the designer. They are also filled with some example data to illustrate what the interface content can be. This gives a sense of look and feel of the application, without being interactive yet. GUI building tools, familiar to the designers, can be used in this step.

Interaction prototyping is the modeling that links the UI template to the domain model, resulting in an interactive application. Unlike layout prototyping, interaction prototyping requires the domain model. The modeler annotates the UI template with queries that link the domain model to the widgets that are supposed to show the information. The modeler also makes semantic annotations to provide advanced interaction like direct manipulation and data transfer. This step can, typically, still be achieved in a GUI building tool by annotating the involved widgets via properties supported by most GUI builders.

A. Layout Prototyping

1) UI Template: several approaches to automatic UI generation exist today, however in our experience they don’t provide the designer with enough control on how the final UI will look like. For this reason we have decided to allow the designer to build the user interface on their own, using the tools and process they are already familiar with. The designer can further populate the prototyped interface with example data, which shows how and where the data appears and how it affects the properties of various widgets used. The prototype interface together with its example data forms a template user interface.

In our production line example, the designer creates just one production line and just one task on the line (knowing that in real case there will be more or maybe none). The designer thus provides a template that shows how repeated elements of the interface will be laid out. Templates can be embedded: the production line template contains the task template.

The production line is a container widget, that uses no layout strategy, since the widgets it contains (tasks) need to be placed freely on the horizontal axis which it draws as its lower border. The task widgets are also simple widgets that place themselves depending on the represented task’s starting date, and set their width depending on the task duration. In graphical appearance, the task widgets are standard labels with a rectangular border. For both the production line container and the task widgets we could have used standard UI framework components with various border settings to set their graphical appearance, but our example also intends to prove the ability to integrate custom widgets with standard ones in a UI template, so we implemented two simple custom widget types.

The designer also places one extra line that is supposed to represent tasks that are not on any production line (they are parked). This is a standard container using a flow layout, containing Task widgets that do not position themselves (let the parking container do so), but just set their width according to the represented task’s duration.

The production line example container, as well as the parking container can be laid out together inside another
container. All the layout facilities offered by the GUI framework can be used to combine template elements, whether they will be annotated later or not.

Finally the designer places a table that represents the task data and allows the user to edit some of the columns (customer, days and start date). The table shows another illustration of user interface templating, with only one row being added as a template for the multiple rows that will be present in the running application. However it is possible for the designer to provide more than one template example, to arrive to a more realistic UI layout prototype. In that case, only the first template will be taken into account in further modeling steps.

The choice of tools used to build the template is not important as long as these tools offer the possibility to access the properties of widgets (as we will illustrate in section III-B) and (if needed) to add custom widgets. For Java this is the case with the major GUI builders (NetBeans, IntelliJ Idea and Eclipse).

2) Example Data: while composing the template UI, the designer can mark that some of the involved properties will be dynamic. For example the text property of each custom task widget will be retrieved from the domain model. The designer marks this fact by associating the text property with an example value "VIP". The designer thus marks that the value is dynamic by providing an example value for it. Where that dynamic value will come from (the customer name in the domain model) is not important at the Layout prototyping stage.

Such dynamic values can also be marked in the table. In this case the designer has chosen the same "VIP" example value for the first table column, though there is no connection between the two values yet (the connection will be established when the template will be annotated). Further example values are provided for the other columns. The designer has also chosen to provide two template rows in the table, to achieve consistency with the template tasks used in the production line and in the parking line respectively. Figure 4 shows the template UI with example data.

The example data allows a realistic exemplary illustration of the UI in the early stage of development. The interaction with that early prototype including the example data will be limited. Also, our approach at the moment is limited to one example data set per widget, which results in the possibility of showing only one variation of UI template at the time. This is not a problem for the simple templates, however graphical scenes are often dynamic, rendering different elements depending on different data values. We aim to enhance our modeling approach in the future to support such variations by allowing several example data sets and allowing the users to switch between them during the prototype demonstration.

3) Graphic Transformation Rules: in order to support interactive diagrams such as the task diagram, the designer can specify how certain values from the domain model will be mapped to positions and sizes in the pixel coordinates. The task widgets must be placed according to the task start date, so the example value that the designer should provide is a date. To convert that example value to an abscissa, a geometric transformation rule is associated with the template line container widget.

\[
\text{abscissa}(\text{Date } aDate) : (aDate - \text{viewportStartDate}) \cdot \text{pixelsPerDay}
\]

where viewportStartDate is the date represented by the leftmost diagram position, \(aDate\) is the date to be transformed, and \(\text{pixelsPerDay}\) represents the diagram

\(3\) Although we present transformation rules and annotations in a mnemonic language to show their symbolic fundament, interactive tools can be made for creating and maintaining them.
scale. `viewportStartDate` and `pixelsPerDay` can be set via user interaction (e.g. from a scrollbar or scale gauge or, as in the current example, can be set to default values.

Once the (1) transformation rule is defined, the `position.X` property of any task widget can be set to a example date value, and later on to a date expression from the domain model.

Geometric transformation rules are bidirectional. The inverse of the (1) expression can be used to convert in the opposite sense, between the `position.X` property of a newly placed task widget to the `startDate`, which is useful in case of a drop action, to determine the new start date of the dropped task.

A further element of the geometric transformation rule is the relation that specifies how a date interval (number of days) is transformed into a size in pixels:

\[
\text{width(int days)} : \text{days/pixelsPerDay} \quad (2)
\]

Once this is defined, the example data provided for the `size.width` property of a task widget will be interpreted as an interval in days and not as a size in pixels. The parking line widget will use the same geometric transformation rule (composed of (1), (2) and their reverse relations) but only (2) will take effect because the `position.X` property is not set for the parked tasks (they are not planned so they are simply placed one after the other by a flow layout).

Geometric transformation rules are one case of the more general graphic transformation rules. Other kinds of graphic transformation rules can be e.g. setting the foreground color of a widget depending on some value from the domain model. Thus we use the more general notion of graphic transformation rules to refer to any rule that changes the graphical appearance, (shape, size, position, color, etc) based on values from the domain model (or from example data).

### B. Interaction Prototyping

In this phase of UI modeling the modeler takes the template UI and the domain model and puts them together to provide the interaction. The first task is to link the data to the widgets so that the querying and updates of the information in the widgets reflects the real data. After this is achieved, the modeler can add some advanced interaction. In our example application we have implemented drag and drop functionality to demonstrate such a case.

1) **Query annotation:** since major part of our targeted applications are CRUD operation, queries play a central role. To build the queries over the domain model the modeler uses our Query Language (QL) which is an adapted version of Hibernate Query Language (HQL). The queries consist of two parts: selection and projection. The selection is used to select the instances of some entities (in QL represented with `FROM` statement) that satisfy certain criteria (represented with `WHERE` statement). The projection chooses which fields of the selected entities will be displayed.

In our running example the user annotates the container of the example production line with the following selection

\[
\text{FROM ProductionLine line ORDER BY line.name} \quad (3)
\]

This will iterate over all the production lines and (re)generate the content in the container for each line. Inside the container the user annotates the text property of the label with the `line.name` projection.

The next step is to represent all the tasks for specific production line. This is achieved by annotating the container of the tasks with the following query:

\[
\text{FROM Task t WHERE t.line = line} \quad (4)
\]

For the remaining two parts of the template (parked tasks and table representation of tasks) similar annotations are applied. The table, as explained in detail in the next section, supports editing of the data. This means that when the application is ran, any changes to an entity in the table should be represented in all the places where the same entity is present. For the controls that support editing (table cells, text inputs, etc.) this is done automatically as soon as they are annotated with some query projection.

At this stage, the modeler can run the application. Figure 5 shows the UI with annotations. Displaying the annotation is part of the tool support and enables the modeler to investigate the annotation superimposed on the actual UI. The annotations can also be shown while the actual data is present and the application is running. This can be beneficial in complex applications with many queries and embedded queries to enable the modeler to find errors.

2) **Semantic annotations:** while query annotations describe how the data will be retrieved and (in the case of the table) what data should be changed by the UI elements, semantics of other user actions must be specified. In general a semantic annotation will provide a link between a user action and an operation described in the domain model.
Semantic annotations related to drag and drop need to specify the domain model object associated to the dragged widget. Thus the task widgets are annotated as follows:

\[
drag: t
\]  
(5)

This means that the \( t \) object (defined in the query annotation (4)) will be set for data transfer in case of a drag action, and it also specifies that drag actions are possible on this template widget in the first place. At the drop end of the drag-and-drop action, the line container has the following semantic annotation:

\[
drop(Task\ task, \ x, \ y):
\]
\[
line.plan(task, \ x)
\]  
(6)

The \( \text{drop()} \) annotation expresses that: (i) objects of type \( \text{Task} \) can be dropped on this template widget (line container widget), (ii) the drop point coordinates are named \( x \) and \( y \), (iii) at the drop action, the \( \text{plan()} \) method of the line object will be called, specifying which task is planned, and at what date and (iv) the start date is obtained via the graphic-data transformation rule (1), starting from the \( x \) abscissa. The \( \text{plan()} \) method expects a date but an integer abscissa is provided, therefore the graphic-data transformation rule takes effect.

A task can also be dropped upon another planned task, not directly on the line, if the place/timeslot where the task needs to be planned is already occupied. Therefore for task widgets representing the \( t \) task the following semantic annotation is made:

\[
drop(Task\ task, \ x, \ y):
\]
\[
t.line.plan(task, \ t.startDate)
\]  
(7)

In this case the drop semantic is that the task will be planned on the same line and at the starting date of the task \( t \) represented by the widget. This semantic is different from the drop semantic in (6) in that the dropped task "replaces" the original task, no matter where on the original task’s widget the drop occurs (i.e. \( x \) is not used). This is a behavior that resulted from tests with users, where it was considered to be a more natural to replace. Note also that the whereabouts of the \( t \) task (drop destination) are not specified by any rule, instead, the \( \text{plan()} \) method from the domain model will take care of moving \( t \) to a later date.

Dragging is possible from any UI widget that represents a domain model object. For example if a \( \text{drag: t} \) semantic annotation is present on a table row, the task \( t \) represented in the row can be dragged on any widget that accepts tasks. The drag-and-drop behavior is automatically generated by the model interpreter so the correct drag cursor is shown to indicate whether a drop is possible or not on various widgets.

Our current implementation of the approach supports the demonstrated drag and drop interaction and its respective annotation. We plan to enhance the modeling capabilities, in order to support other typical interactions, for example scrolling of the task diagram to allow for longer planning time periods. We assume that custom widgets may need their own specific, non-standard, interactions (besides drag and drop). We aim to support the modeling of such interactions via annotations as shown for drag and drop above.

IV. INTERPRETATION OF MODELED UI

The UI model in our approach consists of an annotated template widget tree (Figure 6). The blue annotations represent queries, and will be used to determine how many times a template UI sub-tree will be repeated inside the container they annotate. The green annotations make use of query projections to set properties of the involved widgets. Finally the orange annotations are semantic annotations that indicate the meaning of user actions. In the following sections we will describe how the model is operationalized initially to form an expanded UI tree, where template sub-trees are duplicated according to the cardinality of results from query execution (blue in Figure 7). Once that is achieved, the expanded UI tree needs to be updated due to user interaction, or due to data change. If it is known that only select parts of the data have changed, selective query execution can be applied during update (orange in Figure 7).

A. Synthesis of expanded UI tree through query execution

The expanded UI tree is built from the template UI tree by running the queries that annotate containers. For each result of the query that annotates a container, the container subtree from the template UI tree is instantiated. Each such subtree copy is added to the container instance in the expanded tree. For example the subtree of of the \( \text{LinesContainer} \) at the top of Figure 6 is instantiated for each result of its annotating query (3). Such subtree instantiation also implies applying all the property (green) annotations on the properties of the newly created widgets. In the \( \text{LinesContainer} \) example, one \( \text{LineLabel} \) is created for each result of the query (3) (i.e. for each value of \( \text{line} \)), and the \( \text{text= line.name} \) annotation is applied for each such label, setting the \( \text{text} \) label property to the indicated expression.

Tree expansion is more complex when containers that have a query annotation are themselves tree descendants of query-annotated containers. In that case, their query is an \emph{embedded} query and it needs to be combined with the query of the parent container. For example the \( \text{LineComponent} \) in Figure 6 is annotated with query (4), which uses the \( \text{line} \) symbol, defined in the query that annotates its enclosing \( \text{LinesContainer} \) (3). Therefore annotations (3) and (4) need to be combined to find the composed query of the \( \text{LineComponent} \), as follows:

\[
\text{FROM Task } t, \text{ Line line WHERE } t.line = \text{ line}
\]  
(8)

The composed query (8) is executed and one \( \text{TaskComponent} \) is created for each result (i.e. each
value of \( t \). The results of query (8) are correlated with the results of query (3) to find which LineComponent is the ancestor of each TaskComponent. For such ancestors the \( t.line=\text{line} \) condition is satisfied by the \( t \) of the TaskComponent and the \( \text{line} \) of the LineComponent.

B. Update of expanded UI tree through selective query execution

The semantics of many kinds of user interactions can be generated automatically. For example the semantics of editing of data in the table can be determined based on the property annotations of the table columns. Other interaction semantics can be modeled with semantic annotations, like drag and drop. When events that correspond to such interaction are triggered at runtime and changes to the data happen, the annotated UI model is traversed again to search for the query annotations that correspond to changed entities. When such annotations are found, the respective queries are re-executed and the parts of the UI that they are annotating are updated.

In our example application we have several such cases. For example the table displaying tasks is editable on some columns. If the user modifies some values (for example, the name of the customer), the annotated UI model will be searched for any query that uses the customer name in one of its projections. From Figure 6 we can see that the \( \text{name} \) property is present in the annotation of the LineLabel widget. The query annotating the enclosing container of the LineLabel is re-executed and the widgets affected by its results and projections get re-rendered.

The same approach is applied on more complex interaction like drag and drop. In such cases, the widgets’ semantic annotations are evaluated to find the entities that need to be updated. Once the data changes are known, the widget tree is traversed to find the queries that need to be re-executed, and the widgets they annotate are updated as described above.

V. Evaluation

We have evaluated our approach using a criteria inspired from the related work on the UI interface modeling [4], [5].

1) Development cost and productivity: the development costs are relatively low in regard to the query and semantic annotations, as they are brief and easy to make. Development costs are at the traditional levels for the custom widgets, though in the case of our example application, the development costs of custom widgets are reduced by the graphic transformation rules. Both custom widgets and graphic transformation rules are reusable, as it will be shown below, thus their development costs may be well motivated.

2) Quality of generated application: due to the lightweight nature of the annotations, it is possible to produce reliable and highly-interactive applications in a quick manner. Custom widgets however suffer from the quality problems of manually-written code. Through the integration of the custom widgets in our modeling approach, we can provide unique features, and that suggests that some risks in regard to their code reliability are worth taking.

3) Tool support: we have not yet integrated our approach in any tool as our objective is to validate the approach first. However the approach is designed to be integrated in principle in any GUI builder because the template UI
can be produced in any such builder, and managing widget properties is a common feature of GUI builders. In general our strategy is to integrate our approach with any GUI framework that represents layout as a widget tree, and supports widget property annotation. This includes most desktop and web GUI frameworks. For some of the GUI builders that support a certain GUI framework, it may be possible to support our approach without providing a plugin in that GUI builder, but simply counting on the property annotations supported in that tool.

4) **Usability of the final UI:** since the layout is totally under the control of the designer, the usability will depend a lot on the designer skill. This also means that our approach provides all needed prerequisites for achieving high usability, since the interaction designer can use tools and languages they are familiar with (widgets, containers, properties). Also supporting advanced interaction styles such as direct manipulation, as well as suggestive diagram layouts, contribute to achieving high usability.

5) **Consistency:** the generated interface consistency is, due to the template UI, mostly responsibility of the designer. However our approach makes use of query analysis to determine the type of the data fed to the interface, and the types of widgets and properties using this data are checked against data types, leading to a consistency at data type level.

6) **Scalability:** Our approach scales well from the performance perspective regarding big datasets because of the optimization of the queries done automatically through the analysis of the widget tree annotations. This has already been demonstrated with a data intensive web application [11]. However, for the desktop application some further improvements can be made, specially with the partial updates of the complex widgets. For example, redrawing just specific cells in the table on data updates.

One specific performance scalability concern in our approach is complex interfaces, with large widget trees. Analyzing these trees on each user interaction may appear to have an impact on performance. However, that impact cannot be very high because the number of tree elements in a template UI should not exceed the order of magnitude of hundreds, otherwise rendering them would result in dialogues that are hardly manageable by the user. Also the analysis operates in linear time and does not require query execution. Instead, the number of queries executed is optimized to a minimum as a result of the analysis. In future work we plan to explore this case more extensively.

7) **Extensibility, maintainability, flexibility:** layout changes are easy to do since our approach provides manual template layout. Some problems may occur if layout changes change the hierarchy of embedded query annotations. Given a certain element of the data model, the widget representing that element can be changed easily (though we currently do not provide for changing all widgets representing a certain data type). Changes in the domain model may affect the query annotations and widget property annotations. However our query analysis techniques can detect such dependencies and in the future we can also provide refactoring of the annotations when the domain model changes.

Many UI modeling approaches provide for automatic addition of UI widgets when fields are added in the domain model, and in general treating groups of fields rather than individual fields. Our intended approach to address this issue is to recognize patterns in the template UI where multiple
fields of a certain entity are shown, and to propose the addition of a new widget set of the same pattern when a field is added in the domain model. For example if a new field is added to the Task entity, our modeling approach could propose to add that field in the table template.

8) Retargetability: is low in our approach since the template UI is built on a given GUI framework, therefore retargeting to another GUI framework is not directly possible. In the future we intend to provide for the extraction of an abstract UI model from the template, and to instantiate that abstract UI template in any target GUI framework. Annotations would survive such model transformations since annotations are mostly dependent on the domain model and not on the UI templates.

9) Ease of use by the modeler, learning curve: are assessed to be a strong point of our approach. Query annotation was shown to be learnable in a matter of hours [11], and hierarchical query embedding is regarded as natural. We strive to keep similar simplicity in semantic annotations, property annotations and graphic transformation rules, and in the future we can provide more tool support for them. Template user interface use the standard widget set that the designers are familiar with. We are still assessing the learnability of the overall combination of template UIs with query annotations but their individual learnability gives a good prospect for the learning curve of their combination.

VI. RELATED WORK

Although our work was motivated by a long experience in partial web UI modeling [11] we have found recent approaches with similar ideas. The approach presented by Schramm et al. [5] also aims at combining manual UI building with modeling. Their approach however introduces a new set of tools to achieve this and has a limited set of UI widgets that in a way limit the possibilities and restrict the complexity of the final solution.

Another recent publication [4] demonstrates industry experience in modeling UI by applying certain design patterns and supporting a specific type of widget composition. Their experience however shows that such an approach has a long learning process which is a problem for non enterprise level solutions.

The modeling and generation of UI has a long history with many approaches being based on modeling the user behavior through task models [2], [14], [15] or discourse models [16]. Most of these modeling approaches and their recent advances focus on various model transformations to support multiple contexts of use [6], but are often too strict in regard to the modeling methodology [17], [18].

Meijler et al. [19] give an overview of differences between generative and interpretative UI modeling. Generative modeling often requires a complex process of model regeneration and transformation on any change made by the modeler, while the interpretative gives more flexibility as changes are seen right away in the runtime. The drawback of the interpretative is the restriction to one targeted runtime. We have consciously accepted this drawback, for the flexibility of fast UI prototype iterations and total designer control of the modeled UI.

VII. SUMMARY AND FUTURE WORK

Inspired by a long-term experience of using hierarchical query annotations to produce a large web 1.0 intranet [11] we have presented a generalization of query annotations to more interactive user interfaces, applicable to any widget set with tree-based layout. We also show how interaction styles specific to such highly interactive user interfaces can be supported in modeling, through our examples of modeling interactive diagram representations using free layout, and drag-and-drop interaction. Our approach is based on specifying template user interfaces using tools and widget sets familiar to interaction designers, and allows interaction designers to start their work early in the development process, and to populate their designs with example data for better illustration and early testing.

We have prepared our approach to be integrated in any GUI builder that can produce interfaces in the target GUI framework. Such GUI builders will thus constitute the basis for tool support of query-annotated user interface modeling.

We plan to explore such GUI builder integration in future work. We also intend to support the extraction of the abstract user interface from the concrete template UI, to be able to target a modeled user interface to another GUI platform.

In future work we also plan to develop the concept of hierarchical query annotations to include other query languages and possibly combinations of query languages (HQL, SPARQL, noSQL database queries, etc). While our current query annotations depend on a domain model, we contemplate the possibility of initiating the creation of the domain model by analyzing the queries, which would result in user interface driven application development.

VIII. CONCLUSION

The lightweight query-annotation approach we introduce, although working at a low level, unlike the current UI modeling approaches, shows a promising potential. This potential is suggested by the ability to support interaction styles that are difficult to support with current high-level modeling approaches. The low-level approach also has the advantage that it operates with notions that are familiar to interaction designers (UI templates) or are easy to learn (query annotations).

ACKNOWLEDGMENT

We would like to thank the Young Faculty programme of the CSC school at KTH. The makumba.org community has inspired the needs for this work and our running example code use their open-source code.
REFERENCES


