Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

Generating content presentation according to purpose

Sevan Kavaldjian, Jürgen Falb and Hermann Kaindl
Institute of Computer Technology
Vienna University of Technology
Vienna, Austria
{kavaldjian, falb, kaindl}@ict.tuwien.ac.at

***Abstract*—Programming graphical user interfaces is hard and expensive, while automatic generation is still quite challenging. One of the issues involved in automatic generation is the presentation of content from the domain of discourse according to its purpose in the current context of the human-machine dialogue. For example, it makes a difference whether the same piece of information is to be presented in the context of asking a user something, or simply for the purpose of informing.**

We address this issue through generating content presentation specifically according to the type of communicative act, that indicates the purpose. In the course of transforming a high-level discourse model to a structural user interface model, we apply specific model-transformation rules to content types, depending on the type of communicative act they are referred from. This results in automatically generated user interfaces with content presentations according to purpose.

***Index Terms*—user interface generation, discourse modeling, model transformation, content presentation**

I. Introduction

Manual creation of GUIs (graphical user interfaces) is hard and expensive, so we strive for automated generation. Instead of generating them from simple abstractions, we let an interaction designer (and even end users) model discourses in the sense of dialogues (supported by a tool). From such a high-level declarative discourse model, we have been able to automatically generate the overall structure and the "look" of a GUI, more precisely a WIMP (window, icon, menu, pointer) interface [1] as well as its behavior [2]. This automatic generation employs model transformations.

Still, we had to deal with the presentation of content of the domain of discourse. In particular, the concrete presentation needs to depend on the purpose of the envisaged interaction, since the GUI's usability would clearly not be satisfactory otherwise, and usability is one of the essential problems of automatically generated user interfaces. In this paper, we present our approach to automatic content generation according to purpose, which also employs model transformations.

The remainder of this paper is organized in the following manner. First, we sketch our discourse models. Then we elaborate on the model transformations from such a discourse model to a structural UI model including content presentation. Based on such a derived model, we explain automatic screen generation for the content presentation. Finally, we discuss our approach and compare it with related work.

II. High-level discourse models

The starting point for our automatic GUI generation is a discourse model. Such a discourse model is largely a declarative model and represents a class of possible dialogues. It has the following key ingredients:

- *communicative acts* as derived from speech acts [3] carrying *propositional content*,
- *adjacency pairs* adopted from Conversation Analysis [4], and
- *RST relations* inherited from Rhetorical Structure Theory (RST) [5].

Communicative acts represent basic units of language communication. Thus, any communication can be seen as enacting of communicative acts, acts such as making statements, giving commands, asking questions and so on. Communicative acts indicate the intention of the interaction, e.g., asking a *question* or issuing a *request*. Figure 1 shows such examples in two small excerpts of a larger discourse model for a simple online shop. Figure 1a shows two communicative acts (represented by rounded boxes), a closed question and an answer, for adding a product to the customer's shopping cart. For this purpose, the formal expression within the closed question enables the customer to *select one* instance *from all* instances of the class *Product*, and provides the intention of the question, *adding* the selected instance to the *ShoppingCart* represented by the variable *sc*. Figure 1b shows a different part of the discourse model, for informing the customer on all products available in the store.

Communicative acts typically refer to propositional content. In this example, it is about selecting a product by the customer and providing information on all products. In Figure 1 the propositional content is specified by the text below the type of each communicative act. In fact, it is the *same* propositional content for both communicative acts (*all Product*[1]) that gets uttered.

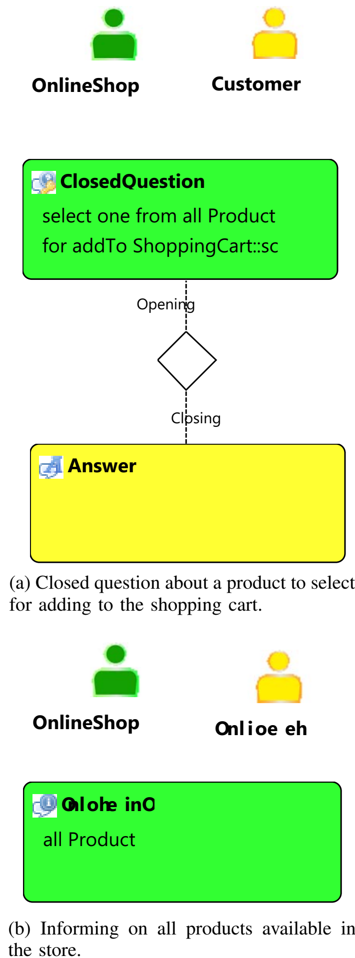
Propositional content is specified in our approach in a model of the domain of discourse, which specifies what the dialogues can "talk" about. Figure 2 shows a very small excerpt of such a model in a UML class diagram.[2] It specifies a single class named *Product* with four attributes.

Adjacency pairs are sequences of talk "turns" that are specific to human (oral) communication, e.g., a *question* should have a related *answer*. Figure 1a shows an example of such an adjacency pair.

[1] *Product* refers to the class with this name in the model of the domain of discourse.

[2] At the time of this writing, the specification of UML is available at http://www.omg.org.

978-1-4244-2794-9/09/$25.00 ©2009 IEEE



(a) Closed question about a product to select for adding to the shopping cart.

(b) Informing on all products available in the store.

Fig. 1. Excerpts of an online shop discourse model.

RST relations specify relationships among text portions and associated constraints and effects. The relationships in a text are organized in a tree structure, where the rhetorical relations are associated with non-leaf nodes, and text portions with leaf nodes. In our work we make use of RST for linking adjacency pairs and further structures made up of RST relations. Our example does not show an RST relation, however. Both excerpts in Figure 1 originate from different parts of a larger discourse model and are, therefore, more indirectly connected with each other. Still, one can imagine to link the *Closed Question-Answer* adjacency pair in Figure 1a via a *Background* relation with an *Informing* on the category the customer can choose from to support her in selecting a product.

III. Model Transformation to Structural UI Model

Our model-driven approach transforms discourse models into structural UI models that are close to the final user interface but still GUI toolkit-independent. We first present

Product
name : EString
price : EDouble
description : EString
picture : EString

Fig. 2. Small excerpt of a domain of discourse model.

the transformation process as well as the transformation rules and heuristics required for content presentation. Subsequently, we describe our transformation engine and its underlying technology.

A. Transformation process and rules

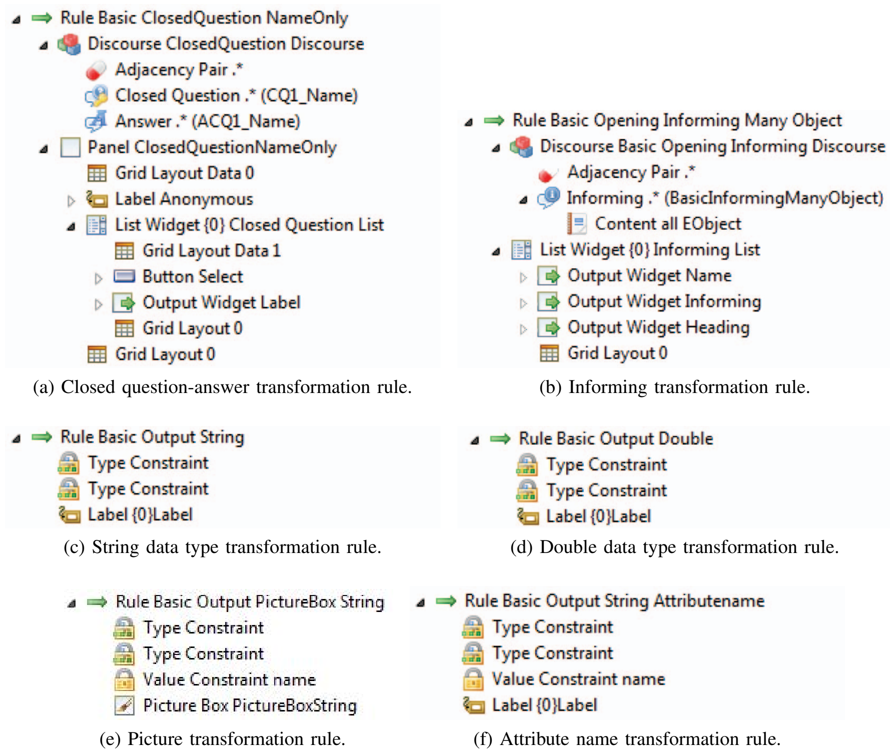
Our model-driven transformation approach uses a process consisting of two interleaved transformation steps:

1) The first step applies rules to discourse model elements that generate an overall UI structure by use of pattern matching. These rules generate abstract widgets like labels for headings and placeholders for data of the propositional content. They also associate parts of the propositional content with the generated placeholders.
2) The second step executes content transformation rules within the context of the rules of the first step. This embedding allows the selection of abstract widgets for the resulting structural UI depending on the content type, the content's referring communicative act type and the current context the communicative act is embedded in as defined by the enclosing rule.

We explain this process in more detail by means of our running example. For transforming the discourse model excerpts in Figure 1a and 1b, we need structural transformation rules for transforming the *Question–Answer* adjacency pair and the *Informing* communicative act (that makes up a degraded adjacency pair by itself). Second, we need content transformation rules for transforming content types, like strings, pictures and numbers depending on the communicative act they are embedded in. The transformation rules also contain heuristics to improve the generated structural UI model. For example, rules can transform content attributes differently based on their attribute name, e.g. a *name* attribute can be used as a heading for the rendered content. The two structural rules below are applied in the first step to our discourse model excerpts:

Closed question-answer transformation rule: The rule in Figure 3a transforms each *Closed Question-Answer* adjacency pair to a *Panel ClosedQuestionNameOnly* with a *Label Anonymous* and a *List Widget Closed Question List*, with each list entry consisting of an *Output Widget Label* placeholder for the content object's identifier (e.g., *name*) and a *Button Select* to select this list item. The output widget placeholder has a property that holds an OCL[3] expression, which selects parts of the content object the output widget is a placeholder for. For

[3]OCL — Object Constraint Language, see http://www.omg.org/cgi-bin/doc?ptc/2003-10-14 for its specification.



(a) Closed question-answer transformation rule.

(b) Informing transformation rule.

(c) String data type transformation rule.

(d) Double data type transformation rule.

(e) Picture transformation rule.

(f) Attribute name transformation rule.

Fig. 3. Transformation rules for closed question-answer (a) and informing (b), rules for transforming output widgets depending on data types (string (c) and double (d)), and for transformations based on heuristics (pictures (e) and attribute names (f)).

example, the *Output Widget Label* selects the *name* attribute of the content object, e.g., the product name, which is later on used for generating the label widgets. The corresponding OCL expression is “eAllAttributes→select(name=‘name’)”.[4] This and the other properties of the *Output Widget Label* are not shown in Figure 3a. The *Label Anonymous* represents a heading for the overall list and its text is derived from the name of the matched closed question communicative act. The generated *Grid Layout* elements allow the specification of the desired number of columns for the resulting list and the placement of the label and the button next to each other as shown in the final UI in Figure 5.

Informing transformation rule: The rule in Figure 3b matches an adjacency pair linked to an Informing (upper part of the figure). The Informing communicative act has to contain more than one content object (indicated by an *all* or *many* quantifier) for generating a list in the structural UI model. The rule transforms the matched input to a *List Widget* which contains the structure of one list item. The list widget contains *Output Widget placeholders* for the transformed content. The *Output Widget Heading* selects again the *name* attribute of the content object as representative for a list element’s heading. The OCL expression for filtering out the *name* attribute is identical to the one in the Closed question–answer transformation rule. Each of the other two output widget placeholders is a placeholder for all the other attributes of the content object. In the second stage, both are rendered differently, the *Output Widget Name* is used to render the attributes’ names and the *Output Widget Informing* is used to render the attributes’ values. This distinct rendering can be seen in the final user interface in Figure 6 below with the attribute names on the left side and the attribute values on the right side.

Within the context of the rules described above, the following four rules are used in the online shop example excerpts in the second step. They transform content object parts—mainly attributes—based on their content type and the type of communicative act that the content object is referred from.

String content type transformation rule: The rule in Figure 3c matches the *String* content type. The name of the rule reflects the matched content type. It generates a label for each string with the content value as the label’s text. The context of this rule is specified by two *Type Constraints*. The first one specifies that this rule can be applied only to *Output Widget* placeholders generated in the first step, and the second one specifies that this rule can only be executed in the context of *Informing* and *Closed Question* communicative acts. In our example, this rule generates a label widget in the resulting structural UI model for each product attribute of type string that is associated with an output widget placeholder.

[4]“eAllAttributes” is part of EClass in the underlying ECore meta-meta-model, which is an Essential Meta-Object Facility (EMOF) implementation in the Eclipse Modeling Framework (EMF).

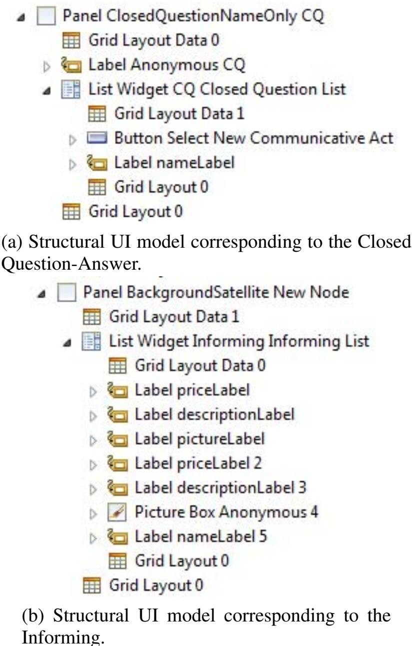
Double content type transformation rule: The rule in Figure 3d is identical to the *String content type transformation rule* apart from matching attributes of type *double* instead of type string. Thus, it creates a label for each numerical attribute. The name of the rule again reflects the matched content type.

Picture transformation rule: The rule in Figure 3e matches also content of type string like the *String content type transformation rule* but contains an additional constraint. This rule is only applied in the context of output widget placeholders and Informing communicative acts, which is specified by the two *Type Constraints* in Figure 3e. In addition, this rule contains a *Value Constraint* that checks if the name of the content attribute contains either the text "picture" or "image". If this constraint holds, the "picture" attribute's value is interpreted as a filename or URL and a *PictureBox* is generated in the resulting structural UI model for this content attribute. Since this rule is more specific than the string content type transformation rule, this rule is applied first when both match.

Attribute name transformation rule: The rule in Figure 3f matches any content attribute and is executed in the context of output widget placeholders and any kind of communicative acts. In contrast to the *String content type transformation rule*, this rule generates a *label* using the attribute's name instead of its value as label text. The attribute's name is retrieved in the rule by assigning the OCL expression "self.name" to the *Label* widget contained in the rule.[5]

When these rules are applied to our running example discourse excerpts, we get the generated structural UI model illustrated in Figure 4a for the discourse model excerpt in Figure 1a, and the structural UI model in Figure 4b for the discourse model excerpt in Figure 1b, respectively. The resulting structure of the *Closed Question-Answer* adjacency pair in Figure 4a corresponds to the structure shown in Figure 3a with the output widget placeholder replaced by the application of the *String content type transformation rule*. The resulting structure of the *Informing* in Figure 4b corresponds to the structure of the Informing rule in Figure 3b with the output widget placeholders replaced by applying all four content transformation rules to the attributes of an online shop product. The set of content transformation rules that can be applied is defined by the context formed by the superior structural transformation rule and the matched communicative act type (purpose of conveyed content). The context has to match all *type constraints* of the content transformation rules.

The rationale for generating the content presentation differently in this way is as follows (the screen shots in Figures 5 and 6 below may illustrate it better than the structural UI model in Figure 4). We render the content item in the context of the *Closed Question* less completely than in the context of the *Informing*, to provide a better overview for the customer during her product selection process. More precisely, in the context of the closed question only the name of the product is displayed, together with a select button. In contrast, the same content



(a) Structural UI model corresponding to the Closed Question-Answer.

(b) Structural UI model corresponding to the Informing.

Fig. 4. Structural UI models corresponding to the online shop discourse model excerpts in Figure 1.

item is presented in the context of Informing by showing all its available information, even including a picture and the names of the attribute fields. So, the different context matters, since the purposes are different: asking vs. informing.

B. Transformation engine

Our model transformation engine is a rule-based engine that is based on the Ecore[6] model. Our metamodels for discourse models and structural UI models are instances of the Ecore model and thus provide common model navigation in the source and target model and linking between elements of both models. We use this feature to add traceability links from widgets in the structural UI model to the original elements in the discourse model. Thus, the original information is also available during the final screen generation process.

The transformation engine iterates over all elements of a discourse model and first checks which rules trigger for the currently processed element. For a rule to trigger, two conditions must hold:

- the discourse pattern in the rule must match the corresponding part in the discourse model, and
- specified rule constraints must match the device properties we want to render for (e.g., screen real estate).

In case of content transformation rules which match content types, constraints can be specified on the rendering context, e.g., they can be used to constrain the set of types of communicative acts the content must appear in. This permits restricting a rule to a specific set of communicative acts. The

[5]The context for evaluating the OCL expression is the matched EAttribute object of the Ecore model.

[6]see http://www.eclipse.org/modeling/emf/ for the Eclipse Modeling Framework (EMF) specification.

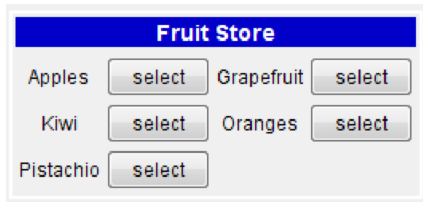


Fig. 5. Screenshot of the final UI representing the *ClosedQuestion*.

four content transformation rules all match content of Informing communicative acts. However, only the *string and double content type transformation rules* can also be applied to content of closed questions.

When multiple rules trigger, a conflict resolution strategy is used to select one rule for firing. The conflict resolution strategy selects the most specific rule based on the size of the pattern to match and the number of constraints. In addition, a rule priority can be used to select one of many rules that have the same specialization degree. For our running example, rule priorities are not necessary because the rules can be uniquely selected for firing without them.

When a rule fires, its widget tree structure is added to the resulting structural UI model and the widgets' content is selected from the discourse model by evaluating OCL expressions on the currently processed discourse model element. Using this rule-based transformation engine with the rules described in the previous section leads to a device-specific but GUI toolkit-independent structural UI model as shown, for example, in Figure 4.

IV. Screen generation

A structural UI model resulting from our model transformation process, like the one in Figure 4a or 4b, contains already the complete structure and layout information of the GUI but is still GUI toolkit-independent. *Screen generation* is our final step that transforms the structural model into GUI toolkit-specific windows and dialogs and generates code for them. Currently, we support the Java Swing[7] and Eclipse SWT[8] GUI toolkits. This screen generation step solves four tasks:

- It maps the abstract widgets of the structural model to toolkit-specific widgets,
- it maps the generic structural UI layout to a toolkit-specific layout,
- it formats the toolkit-specific widgets according to cascading stylesheets (CSS), and
- it generates the event handling and the binding to the user interface behavior (represented as a generated finite-state machine that is derived from the discourse model obeying the procedural semantics of the RST relations as described in [2]).

Figure 5 displays the screen resulting from the structural UI model in Figure 4a by applying the screen generation for

[7]http://java.sun.com/products/jfc/
[8]http://www.eclipse.org/swt

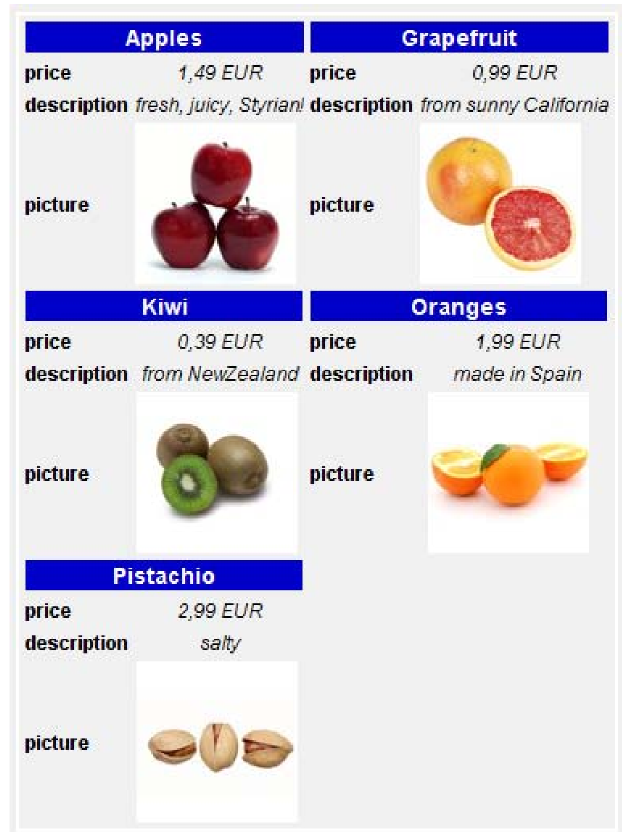


Fig. 6. Screenshot of the final UI representing the *Informing* on all products.

Java Swing. Figure 6 shows the screen resulting in analogous manner from the structural UI model in Figure 4b. In this example, we achieved a one-to-one mapping between structural model widgets and Java Swing widgets, only the PictureBox widget in Figure 4b is mapped to a Java ImageIcon embedded into a JLabel. In some cases, the mapping process is more complex, e.g., our structural UI metamodel contains an image map widget which requires an image, a label widget and the implementation of multiple active areas within Java Swing.

For transforming the layout information of the structural UI model, we implemented an algorithm that calculates layout data required for the toolkit-specific layout managers. E.g., for the Java Swing *GridBagLayout* we calculate the weight of each grid cell, which is important for resizing windows, depending on the widget types and the layouting rule applied to the RST relation.

Further, the screen generation process formats the generated widgets according to style information stored in a cascading stylesheet (CSS). The appropriate style is selected according to the selection algorithm defined in the CSS specification based on the widget name, the style identifier and the widget type defined in the structural UI model. So, this provides a GUI toolkit-independent mechanism for specifying widget styles that is transformed by the screen generation into toolkit-

specific method calls to set the toolkit widget's attributes.

In addition, the screen generation results in toolkit-specific event handlers that

- collect information from the input widgets,
- modify content objects provided by the application logic according to the collected information, or
- generate new content objects based on the collected information,
- and sends them to the application logic in response to a question or as a new request.

A more detailed description of the screen generation process can be found in [6].

V. Discussion

Still, an essential problem of automatically generated UIs is their usability. So, let us briefly discuss if and how our approach may help in this regard.

Generally, taking the user's tasks to be supported into account is supposed to help achieving good usability. In fact, many of the current approaches to automated UI generation start from task models. Our approach is built on discourse models, but we found that they implicitly also represent tasks. Primarily, our discourse models represent classes of dialogues with an end user. And in this way, it should help with regard to usability as well. After all, usage scenarios are well known in this regard, and our discourse models may also be viewed as classes of scenarios with additional information on how the steps are connected.

We are not aware of any approach to automated UI generation that would take the specific context into account for presenting content. In this particular paper, we address this issue and show that and how the specific purpose can be taken into account for presenting content. E.g., we propose more specific and appropriate widget selection for the specific purpose, and our model transformations implement it. So, this may be another step into the direction of improving usability of automatically generated UIs.

VI. Related Work

Our approach to GUI generation relates to TERESA by Mori *et al.* [7]. Both start from high-level models, but our discourse models have a focus on dialogues and seem to be even on a higher level than the task models shown in [7].

The UI Pilot approach by Puerta *et al.* [8] is semi-automatic by requiring the designer to specify tasks and a so-called wire-frame for the user interface. Afterwards, the tool can suggest widgets for each user interface element. This approach provides more flexibility to the user interface designer than our approach, which allows fully automatic content presentation.

Elkoutbi *et al.* [9] present an approach that generates a user interface prototype from scenarios. Scenarios are enriched with UI information and are automatically transformed into UML statecharts, which are then used for UI prototype generation. In contrast to this approach, we model classes of dialogues supporting a set of scenarios. We transform our discourse models to UML statecharts as well, but we do not have to enrich them for this purpose.

Pederiva *et al.* [10] describe a beautification process that helps a designer to improve a generated user interface via a constrained user interface editor. This editor allows applying beautification operations to specific UI elements, resulting in model-to-model transformation. Since our approach involves content presentation according to purpose, beautification should be less important for this part of UI generation.

VII. Conclusion

We address the problem of presentation of content from the domain of discourse according to its purpose in the current state of the human-machine dialogue. This purpose relates to the intention indicated by the type of the communicative act that refers to propositional content to be presented. Our approach takes this type into account in the course of automatic content presentation. In this way, it leads to generated user interfaces with content presentations according to purpose.

Acknowledgements

This research has been carried out in the CommRob project (http://www.commrob.eu) and is partially funded by the EU (contract number IST-045441 under the 6th framework programme).

References

[1] S. Kavaldjian, C. Bogdan, J. Falb, and H. Kaindl, "Transforming discourse models to structural user interface models," in *Models in Software Engineering, LNCS 5002*. Berlin / Heidelberg: Springer, 2008, vol. 5002/2008, pp. 77–88. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69073-3_9

[2] R. Popp, J. Falb, E. Arnautovic, H. Kaindl, S. Kavaldjian, D. Ertl, H. Horacek, and C. Bogdan, "Automatic generation of the behavior of a user interface from a high-level discourse model," in *Proceedings of the 42nd Annual Hawaii International Conference on System Sciences (HICSS-42)*. Piscataway, NJ, USA: IEEE Computer Society Press, 2009.

[3] J. R. Searle, *Speech Acts: An Essay in the Philosophy of Language*. Cambridge, England: Cambridge University Press, 1969.

[4] P. Luff, D. Frohlich, and N. Gilbert, *Computers and Conversation*. London, UK: Academic Press, January 1990.

[5] W. C. Mann and S. Thompson, "Rhetorical Structure Theory: Toward a functional theory of text organization," *Text*, vol. 8, no. 3, pp. 243–281, 1988.

[6] S. Kavaldjian, D. Raneburger, J. Falb, H. Kaindl, and D. Ertl, "Semi-automatic user interface generation considering pointing granularity," in *Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics (SMC2009)*, San Antonio, TX, USA, Oct. 2009.

[7] G. Mori, F. Paterno, and C. Santoro, "Design and development of multidevice user interfaces through multiple logical descriptions," *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 507–520, 8 2004.

[8] A. Puerta, M. Micheletti, and A. Mak, "The UI Pilot: A model-based tool to guide early interface design," in *Proceedings of the 10th International Conference on Intelligent User Interfaces (IUI'05)*. New York, NY, USA: ACM Press, 2005, pp. 215–222.

[9] M. Elkoutbi, I. Khriss, and R. K. Keller, "Automated prototyping of user interfaces based on UML scenarios," *Automated Software Engineering*, vol. 13, no. 1, pp. 5–40, 2006.

[10] I. Pederiva, J. Vanderdonckt, S. España, I. Panach, and O. Pastor, "The beautification process in model-driven engineering of user interfaces," in *Proceedings of the 11th IFIP TC 13 International Conference on Human-Computer Interaction — INTERACT 2007, Part I, LNCS 4662*. Rio de Janeiro, Brazil: Springer Berlin / Heidelberg, Sept. 2007, pp. 411–425. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74796-3_39