Generating Recommendation Dialogues from Product Models

Sven Radde and **Matthias Beck** and **Burkhard Freitag**
Institute for Information Systems and Software Technology, University of Passau
D-94030 Passau, Germany
{sven.radde, matthias.beck, burkhard.freitag}@uni-passau.de

Abstract

For complex and frequently changing product domains, the maintenance of an electronic recommender system is a time- and money-consuming task, as the man-machine interface has to be adapted to the product model any time the latter is changed. Ideally, changes in the model would lead to an automatic adaptation of the recommendation dialogue without much overhead. In this paper we present an approach to generate an elaborate dialogue from a given product model, ensuring efficient reaction to changes of the model. Using statecharts to structure the dialogue, an intuitive, easily visualizable internal representation can be inferred that is able to handle all principal functions required of a recommender system, such as dynamic dialogue management, belief revision and the generation of recommendations.

Introduction

Today's market for new automobiles is characterized by a huge number of choices, extended by different variants per vehicle, numerous optional features and special equipment, often available only in packages combined with other extras. Customers need qualified consultation to match their preferences with these complex product models.

Apart from its complexity, the product domain changes frequently and often radically. Updates of the product domain may stem from current technical innovations which often require significant adjustments to a recommendation dialogue to accommodate previously unknown functionality. As an example, imagine the recent boom of GPS navigation assistants and the currently emerging approaches to location-based services where salespersons have to familiarize themselves with entirely new technologies, services and even business models. Furthermore, the recommendation process has to be adapted to the current marketing campaigns and to regular changes of the product catalogue such as updated versions of a given vehicle type.

Salespersons have to invest much time making themselves familiar with the new product catalogues to keep their recommendation strategies up-to-date. Supporting a salesperson with an electronic recommender system reduces the amount of time that has to be invested into training and lead to a more efficient recommendation process.

The contribution of this paper is a uniform method to infer the structure of the recommendation dialogue from the product model. This way, normal maintenance procedures applied to the model, e.g., to include new products into the next catalogue sent to the vendors, automatically lead to an updated dialogue, which enables timely and cost-efficient reactions to changes in the product domain.

Statecharts are used to model the dialogue that captures the preferences of the customer in an elaborate preference model. It includes functions to handle system- and user-initiated changes of topic, belief revision and to generate recommendations based on the preferences. For the latter, we transform the preference model into statements of an extension of standard SQL that allows a fine-grained ranking of the product catalogue with respect to the preferences of the customer.

In this paper we present an application of *statecharts* as a way to formalize recommendation dialogues and a *generation method* to infer the statecharts from the model of a product domain. Our approach uses an elaborate *preference model*, dynamic *dialogue management* and a *ranking method* for the product catalogue. The rest of this paper is organized as follows: In the next section we describe the underlying product metamodel before giving an overview of the dialogue modelling and our preference model. Afterwards, we explain in detail how we represent the recommendation dialogue using statecharts. We conclude with a discussion of related work and some outlooks.

Product Metamodel

Our approach is to generate the recommendation dialogue from the description of a product in a generic way. This ensures that changes in the product model are directly represented in the dialogue during the process of normal model maintenance. To this end, we will first formulate our notion of the product metamodel.

Our modelling takes into account that a product usually consists of different *components* that are largely independent of each other. This is especially true if totally independent products are to be "bundled" together. Each component of a product part will have a number of distinctive *features*. These form the basis of a decision for or against this particular component. The *value-range* $dom(f)$ of a feature is represented in the model as a finite set of values (in practice,

continuous features like "price" can be discretized easily). A simplified model of the product domain "car" will serve as our running example for the remainder of this paper:

Example 1 *A car consists of three components that are mutually independent from the point of view of the customer's decision making: the vehicle body, the engine, and the lacquering/painting. Apart from a designation/name, the body has the following features: A design form such as "*Sedan*", "*Roadster*", "*SUV*"' etc. and a price which is divided into classes, e.g., "*less than $ 10.000*", "*between $ 10.000 and $ 20.000*" etc. Engines may be distinguished by their horsepower, fuel type, fuel consumption and their price. The painting has a type ("*normal*" or "*metallic*"), of course a color and, again, a price.*

While product components can be treated in a largely independent way, restrictions on the way different components can be combined may exist (e.g., metallic paint may not be available for some product lines of vehicle bodies). The metamodel allows to specify so-called *restriction relations* $R_{f_1,f_2} \subseteq dom(f_1) \times dom(f_2)$ to model the fact that a valid product may have any combination of values $v_1 \in dom(f_1)$ and $v_2 \in dom(f_2)$ *except* if $(v_1, v_2) \in R_{f_1,f_2}$.

Structuring the Dialogue

The primary goal of the recommendation dialogue is to elicit the customer preferences in a way such that suitable recommendations can be made. We choose to obtain these preferences *explicitly*, by asking the customer questions, instead of applying implicit methods such as collaborative filtering. Explicit elicitation mimics the structure of a conventional talk with a salesperson and seems therefore suitable to our intention of supplementing a natural recommendation dialogue. It is our goal to derive the structure of the recommendation dialogue directly from the product model itself.

As noted previously, product parts may often be combined in a largely independent way to compose the product ("bundle") to be sold, while a user can have preferences regarding the possible values of the features of each component. Whenever this precondition holds, it is suitable to represent the product components as separate *topics* which are structured as a number of *questions* eliciting preferences for a particular feature from the user.

Example 2 *The recommendation dialogue for our car example is composed of three topics: "vehicle body", "engine" and "painting". The topic "painting" will consist of three questions, dealing with "type", "color" and "price", respectively.*

We conclude that the structure of the dialogue can be derived from the product model in a rather straightforward way. Before describing the formalization of this dialogue structure, we will detail our notion of a preference model.

Preference Model

As described, each feature f of the product model is handled by one question. The goal of a question is to elicit the preferences for each of the elements of $dom(f)$ that the customer may either like, dislike or does not care about.

We define a *preference* as a 3-tuple $pref = (f, v, prefval)$ with f being a feature of the product, $v \in dom(f)$ and $prefval \in \{pro, con, dontcare\}$.

Example 3 *When offered the possible different choices of color for his/her new car, e.g., red, green, or blue, the customer specifies the following three preferences: (color, red, pro), (color, green, con) and (color, blue, dontcare).*

The recommender system maintains a set P of all currently specified preferences. If a question regarding a particular feature f is answered another time, the older values are overwritten by the newly stated preferences

In general, the value-range is specified within the product model. However, given the current instance of the product catalogue and previously specified preferences, not every element of $dom(f)$ needs to be presented to the user. For example, the recommender system shall automatically prune the value-range $dom(f)$ of those values that are not present in the catalogue. Furthermore, if there exists a restriction relation $R_{f',f}$, the system should not ask for a value $v \in dom(f)$ if a preference of (f', v', pro) is already present in P and $(v', v) \in R_{f',f}$, because a positive preference for v could not be fulfilled anyway.

Representing Dialogues as Statecharts

We chose to treat the recommender system as a state machine and to use statecharts (Harel 1987; Wieringa 2003) as a means to visualize and formalize our modelling. Statecharts extend Mealy diagrams in some useful details (state reaction, state hierarchy and parallelism) that offer a very powerful and convenient means of expressing the different actions within our recommendation process. In addition, transforming statecharts into the syntax of UML 2.0 State Diagrams (cf. (Wieringa 2003; Drusinsky 2006; Booch, Rumbaugh, & Jacobson 2005)) is straightforward. Once the statechart has been generated based on the product model, runtime semantics and a concrete implementation can easily be inferred or even automatically generated with Model Driven Architecture techniques (see, e.g., (Frankel 2003) for information about *MDA*).

Representing a Single Topic

As we have described previously, the dialogue is structured into topics and questions by the underlying product model. We now present a method to construct a statechart given this structuring. A topic is a superstate with suitable name and consists of the following substates (see figure 1 for the construction of the "painting" topic from example 2):

First, we have an initial state, a final state and one state for each question. The system remains in one of the question states (meaning that the according question is currently displayed by the GUI) until the user responds to the asked question. This will fire the *answer* event which in turn will cause the preferences the user has given in his or her answer to be stored. Furthermore, it will cause a transition to the *choose_next_question* state which may either select another question of the same topic for being asked next or cause a transition to the final state of this superstate, indicating a change of topic.

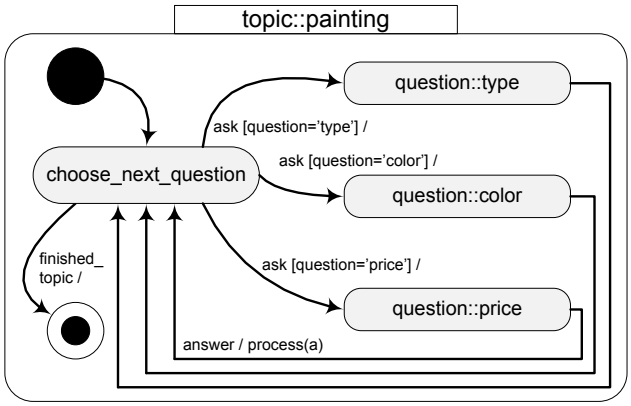

Figure 1: Statechart for topic "painting"

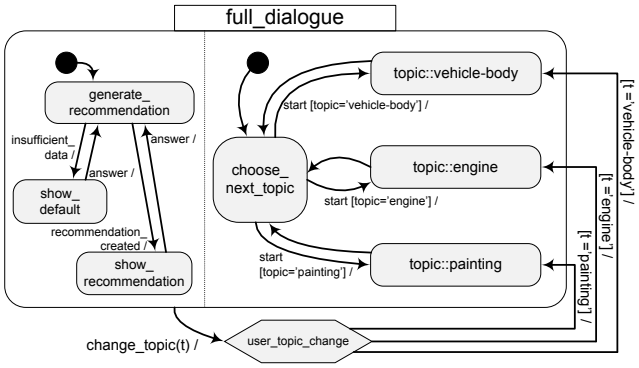

Figure 2: Statechart for the full dialogue

In the *choose_next_question* state, probabilistic inference based on Bayesian networks (see, e.g., (Russel & Norvig 1995)) is applied to reason about the necessity of a change of topic and the most suitable question to be asked next. This allows aggregation of, e.g., estimates of patience of the customer and quality of the recommendation into a probability of the necessity of a topic change. Furthermore, questions may change priority based on previous answers, which can be modelled in a Bayesian network rather naturally. Particularly the generation of probabilistic networks given our ever-changing product model is a big challenge. However, a full elaboration on the subject of the Bayesian inference techniques is beyond the scope of this paper.

Each of the topics of our dialogue structure is modelled separately, as described above. Before describing how all topics are integrated, we will describe the generation of recommendations.

Generating Recommendations

The necessary steps to generate a recommendation are essentially independent of the underlying product model and are therefore modelled in a generic way. This part of the system will run in parallel to the dialogue process and is illustrated in the left part of figure 2. Due to the assumed independence of the product components, recommendations can be constructed independently for each component as well. Note that only the recommendation belonging to the currently discussed product component is displayed.

We define a recommendation as a *sorted subset* of all available items of the product catalogue for a given product component. To generate recommendations, we transform our preference set into statements of the weight-annotated extension of the SQL as described in (Beck, Radde, & Freitag 2007). Roughly speaking, *pro*-preferences are assigned positive weights, *dontcare*-preferences are assigned weights of zero while *con*-preferences receive negative weights. This leads to an *ordered* result set with the most suitable products first.

In the state *generate_recommendation*, the currently available preferences are applied to the product catalogue to obtain the current recommendation. The recommendation engine may issue either the *recommendation_created* event which indicates a valid recommendation and causes a transition to the state *show_recommendation*. This instructs the recommendation system to display the recommended products in its user interface. Alternatively, the event *insufficient_data* means that there were not enough preferences to create a recommendation (i.e., no preferences at all, immediately after startup). The system state is switched to *show_default* where the user-interface displays some sort of "default" recommendation (e.g., current special offers).

Due to space limitations, we can only sketch the course of the dialogue if no products of the catalogue fulfill the stated preferences of the customer. The recommender system derives a set of "relaxation candidates" from the preferences and presents a list of the corresponding features to the user so that he or she can choose which preferences shall be revised. Then, a state transition leading directly to the appropriate question state is executed. From this point, the dialogue continues as if the question was answered normally.

Integrating all topics

We will now describe the integration of all topics as modelled along the lines of the example in figure 1. Each statechart modelling a particular topic is embedded into a larger statechart that contains all topics derived from the current product model (see figure 2) and the *choose_next_topic* state that is responsible for determining the next appropriate topic (compare this construction to the approach used within topics in figure 1). A change of topic is then initiated by a transition into one of the final states within a topic. This "returns" control to the *choose_next_topic* state, which will then invoke another of the topic states, taking into account which of the topics are not finished yet.

Note how figure 2 combines the generated dialogue structure with the aforementioned construction that deals with recommendations. The syntactical element of the statechart syntax used here is called an AND-state, introducing *parallelism* in the execution of statecharts. While the state machine is in the "compound" state *full_dialogue*, it is simultaneously in one of the substates of each component state. Each part of the compound state reacts to events and executes state transitions independently of the other part. Notice

that both the dialogue statechart (right part) and the recommendation statechart (left part) react to the *answer* event that is fired when the user answers a question ("event broadcasting"), resulting in the parallel changing to a new question and the generation of a recommendation.

To provide a user-friendly dialogue, initiation of topic changes must not be limited to the system. The user must be allowed to change the topic himself, which is consistent with our approach to mimic a dialogue with a salesperson. Two "levels" have to be considered: 1) changing the question within the current topic and 2) changing the topic itself.

To this end, we have introduced another two events: *change_question(q)* and *change_topic(t)*. Both events cause transitions to *decision states* that "dispatch" to the appropriate question or topic, respectively. The bottom of figure 2 illustrates the handling of *change_topic(t)*. Question switches within the same topic are similar and were omitted due to space limitations. The events are generated by the user interface, i.e., if the user clicks on appropriate links in the navigation menu of a web-based application.

Discussion and Conclusion

Statecharts have been proposed as a means of modelling general conversational dialogues (Kölzer 1999). However, her intention is to provide dialogue designers with an intuitive tool for modelling the dialogue, whereas we focus on generating the dialogue automatically and use statecharts as a means of visualization and basis of further processing (i.e., code generation).

In (Schmitt & Bergmann 2001), dialogues are formally modelled as state machines. In their model, a "situation" (state) contains information about the preferences of the user, the dialogue history and the current recommendation, resulting in a large space of possible situations. Depending on user input, different transition functions between these states are executed (called "interactions"). A dialogue developer has to define the "strategy" of the dialogue either by modelling a directed graph that combines the possible interactions with the possible situations (resembling, in a way, a statechart) or by providing a set of ECA rules. This approach is static and most probably not applicable to the rapidly changing domains that our approach targets due to the presumable complexity of the model. Alternatively, they propose dynamic techniques based on CBR to determine the next relevant question to be asked, e.g., by measuring information gain. It does not appear that an explicit product model is the basis for these methods as, contrastingly, is the case with our work. We are currently investigating how their approach can be integrated with our algorithms for question choosing.

In (O'Neill *et al.* 2005) an object-oriented architecture to handle spoken-language dialogues is proposed. The different topics of the domain are treated by "EnquiryExperts" with "SupportAgents" to handle secondary tasks. Strategy is defined by ECA rules, while learned knowledge is encapsulated in one "DialogFrame" per expert. The static rule base would be hard to maintain in the presence of changing product domains, because this approach does not suggest a way to infer rules, e.g., from the domain knowledge already modelled in the DialogFrames.

We have presented an approach to generate recommendation dialogues based on a given product model with particular attention on supporting frequent changes of the product model. The dialogue allows the elicitation of preferences by asking direct questions, as well as user- and system-initiated changes of topic and belief revision based on relaxation of preferences. Recommendations are generated using an SQL-extension that allows a fine-grained ranking of the product catalogue.

Execution of the dialogue statecharts has been implemented in a first prototype that is able to accommodate changes of the product model without the need for modifying the program itself. We are currently evaluating techniques for a more "intelligent" selection of questions based on probabilistic inference with Bayesian networks. This will also require to introduce a sophisticated user model to allow, e.g., reasoning about patience and preferences of topic.

Furthermore, we seek to exploit the fact that statecharts have a defined semantics as a part of the UML, which would allow their execution by a general "statechart-interpreter", in contrast to the J2EE web application that forms our current prototype. Also, we will investigate the design of a suitable UML profile to employ Model Driven Architecture techniques to directly obtain program code from the diagrams.

References

Beck, M.; Radde, S.; and Freitag, B. 2007. Ranking von Produktempfehlungen mit präferenz-annotiertem SQL. In *Datenbanksysteme in Business, Technologie und Web*, volume 103 of *LNI*, 82–95. Springer Verlag. In German.

Booch, G.; Rumbaugh, J.; and Jacobson, I. 2005. *The Unified Modelling Language User Guide*. Addison-Wesley.

Drusinsky, D. 2006. *Modeling and Verification Using UML Statecharts*. Elsevier.

Frankel, D. S. 2003. *Model Driven Architecture*. Wiley & Sons.

Harel, D. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8(3):231–274.

Kölzer, A. 1999. Universal Dialogue Specification for Conversational Systems. In *Special Issue on Intelligent Dialogue Systems*, number 9 in News Journal on Intelligent User Interfaces. ETAI.

O'Neill, I.; Hanna, P.; Liu, X.; Greer, D.; and McTear, M. 2005. Implementing advanced spoken dialogue management in Java. In *Special issue on principles and practice of programming in java*, volume 54 of *Science of Computer Programming*, 99–124. Elsevier.

Russel, S. J., and Norvig, P. 1995. *Artificial Intelligence: A Moderm Approach*. Prentice Hall International, Inc.

Schmitt, S., and Bergmann, R. 2001. A Formal Approach to Dialogs with Online Customers. In *Proceedings of the 14th Bled Electronics Commerce Conference*, 309–328.

Wieringa, R. J. 2003. *Design Methods for Reactive Systems*. Morgan Kaufmann.