Beyond Data Models for Automated User Interface Generation

Angel R. Puerta, Henrik Eriksson, John H. Gennari, and Mark A. Musen

Medical Computer Science Group
Knowledge Systems Laboratory
Departments of Medicine and Computer Science
Stanford University
Stanford, CA, 94305-5479
{puerta,eriksson,gennari,musen}@camis.stanford.edu

Researchers in the area of automated design of user interfaces have shown that the layout of an interface can, in many cases, be generated from the application's data model using an intelligent program that applies design rules. The specification of interface behavior, however, has not been automated in the same manner, and is mostly a programmatic task. Mecano is a model-based user-interface development environment that extends the notion of automating interface design from data models. Mecano uses a domain model—a high-level knowledge representation that augments significantly the expressiveness of a data model—to generate automatically both the static layout and the dynamic behavior of an interface. Mecano has been applied successfully to completely generate the layout and the dynamic behavior of relatively large and complex, domain-specific, form- and graph-based interfaces for medical applications and several other domains.

Keywords

Model-based interface development
Automated interface design
Interface models
Domain Models
Data Models

1. Introduction

One of the areas that is receiving increased interest by researchers is that of *model-based user interface development.* This emerging technology is centered around the premise that a declarative interface model can be used as a basis for building interface development environments. The model-based approach facilitates the automation of the design and implementation of user interfaces.

In addition, researchers have shown that an application's data model can be used effectively to generate the static layout of an application's interface (deBaar, 1992; Janssen, 1992). However, data models have not been applied to the generation of interface behavior specifications.

In this paper, we present Mecano, a model-based interface development environment that extends the concept of generating interface specifications from data models. Mecano employs a *domain model* to generate not only the layout of an interface, but also its dynamic behavior. Domain models are high-level representations of the objects and relationships in a domain. Because they explicitly declare domain characteristics that are not normally part of data models, a domain model offers the possibility of automating a larger part of the interface design process than what is feasible with regular data models.

The rest of the paper is organized as follows. We first detail the concept of model-based interface development and its relationship with automated user-interface design. Then, we define and exemplify domain models and contrast them to data models. Furthermore, we explore the architecture of Mecano and describe the interface-behavior generation process. Finally, we relate this work to other research efforts and present a number of conclusions.

2. Model-Based User-Interface Development

The design and implementation of user interfaces is an iterative process that cycles from design to development until a satisfactory product is achieved. There are a number of tools that support the different phases of user interface construction. Current tools either focus on the design phase, or on the development phase, and have a number of shortcomings that make interfaces difficult to build.

The principal areas where current user interface tools falter are:

- **Lack of integrated design and development support.** Design tools such as Hypercard do not support development, whereas development tools such as many UIMSs do not support high-level design.
- **Lack of support for dynamic behavior specifications.** Conventional programming languages are the only way to specify most of the dynamic aspects of a user interface.
- **Low level of automation.** Interface components such as windows and menus must be designed and specified one by one. There is no support for making global changes that affect all components or groups of components.
- **Poor lifecycle support.** Design changes are difficult to propagate, and maintenance is time-consuming because every change must be applied manually to each interface component.

The shortcomings of available user-interface tools are due to a common cause: Developers work at a low level of abstraction—with items such as windows, widgets, and programming constructs—but have no access to explicit representations of the design knowledge needed to create an interface. A solution to such shortcomings is to provide developers with declarative *interface models*, that allow manipulation of all facets of interface design at a high level of abstraction. The key idea is to use an interface model as the central component of an integrated design and development environment that supports all phases of user interface construction.

Figure 1 shows a basic generic architecture for model-based interface-development systems. The key component is a declarative interface model that represents the various characteristics of a user interface design, from presentation and behavior to user- and working-environment preferences. Because the interface model covers all facets of interface design and development, it can be used as a central repository of knowledge that software tools can access to perform functions related to interface construction. Thus, the set of design-time tools manipulates the model directly to build

an interface design. Examples of such tools are model editors, design critics, and design-alternative generators. Similarly, run-time tools use the model to add support to the human–computer interaction process. Examples of run-time tools are help generators and performance-monitoring tools. The run-time tools are intrinsically related to the application-state monitor (which in some systems may be part of the run-time system). This component keeps track of the current, previous, and possible future states of the interaction, and communicates with the run-time tools to relay needed state information to those tools. The run-time system—in some systems called the interface generator—accesses the model to implement the design embodied in a particular *interface model instance* as a running user interface.

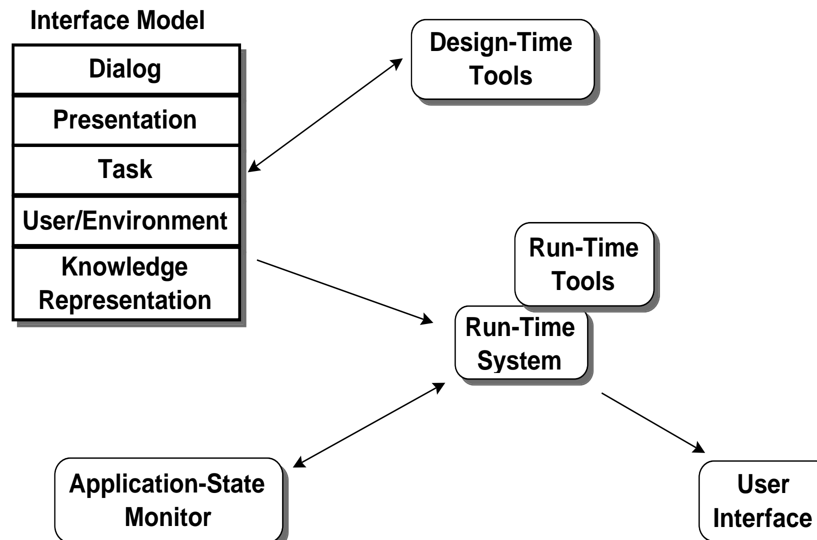

Figure 1. The basic architecture of a model-based user interface development system. The interface model is a central repository of interface design knowledge. A set of design tools manipulate the model to achieve a particular design. The run-time system implements the design in the interface model as a running interface. The run-time tools support the use of the interface—with functions such as help—by monitoring the state of the interaction through the application-state monitor.

Model-based systems create an integrated interface-development environment where developers move from a generic description of interfaces (the interface model) to a specific description of a single interface (the interface-model instance) using design-time tools. The systems then take the model instances and implement them as interfaces, adding the interaction support of the run-time tools. By centralizing all the design knowledge, and by abstracting such knowledge at a high level, model-based systems offer the opportunity to streamline the iterative process of interface construction, to allow implementation of global definitions and design changes, to support specification of dynamic behavior without the need for conventional programming, and to automate major parts of the interface-development cycle.

The few model-based systems that have been developed tend to fall into one of two categories: (1) systems that *assist* in the design process, and (2) systems that *automate* the interface-design process. Systems in the first category normally include design-time tools with advanced model-visualization and editing capabilities. The underlying philosophy is to facilitate the task of model manipulation. Systems in the second category contain complex tools that can instantiate large portions of the interface model for a given interface. Their corresponding philosophy is to minimize the amount of effort needed for model manipulation during design of an interface. In general, design-assistance systems offer maximum design flexibility but increase developer effort,

whereas design-automation systems minimize developer effort but offer less design flexibility, thus requiring additional custom-tailoring to complete satisfactory interfaces. Mecano is a design-automation model-based development system with special facilities to assist developers in customizing generated interfaces. In the next section, we examine the approach to interface generation in Mecano.

3. Automatic User Interface Generation

One important type of design-time tool in model-based interface-development systems is that of automatic interface generators. Such tools partially specify an interface design from a higher-level specification, such as a data model, or a dialog representation. Of special interest to the goals of Mecano as an environment that can integrate the development of applications and interfaces is the use of an application's data model to generate the static layout of an interface (deBaar 1992; Janssen, 1992).

Figure 2 shows a generic framework for automated interface-generation environments that employ data models. An intelligent program examines the data model and applies a set of design rules to produce a static layout design for an interface. Because the data model is shared between the interface design and the target application design, both designs can be coupled, and changes to the application design can be propagated easily to the interface design. The dynamic behavior of the interface, however, must be specified separately. This process can take many forms, from using a graphical editor to construct dialog Petri nets (Janssen, 1992), to assigning sets of pre- and postconditions to each interface object (Gieskens 1992). Although working with high-level dialog specifications is helpful to interface developers, it does not automate the design of dynamic behavior. For large interfaces, editing the dialog specifications is still a time-consuming task involving the definition of hundreds of actions and conditions, some of which may conflict with each other.

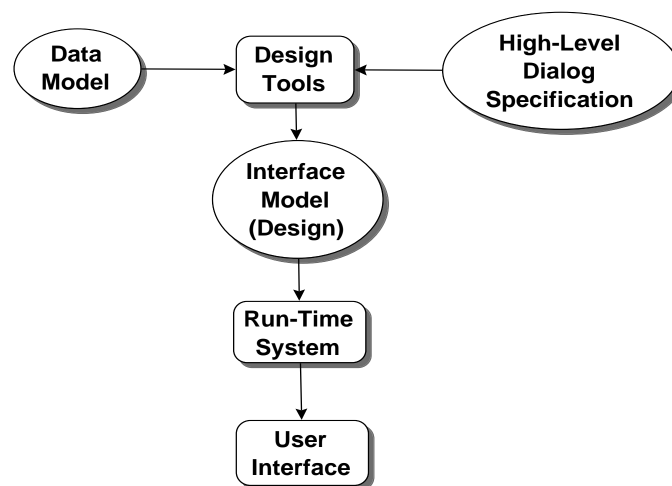

Figure 2. Generic framework for automated interface-generation environments that employ data models. The interface design is produced by tools that examine a data model and a dialog specification. The design may be represented implicitly or explicitly (as an interface model). The run-time system implements the design.

The main reason that current data-model approaches cannot automate the design of dynamic behavior is that data models themselves are very limited in what they express. They are applied

only to serve as a vocabulary to access the data structures of the application. The intelligent design tools that examine the data model can only make design decisions based on the information in the data model.

Most of the dynamic behavior of an interface is domain-specific, but data models are not used to capture effectively the characteristics of a given domain. In Mecano, we intend to use domain models instead of data models to generate interfaces. A domain model is a representation that captures all the definitions and relationships of a given application domain and that subsumes the data model for the application. By substituting the data model in Figure 2 with a domain model, Mecano does not require dialog-specification editing and is able to generate complete dynamic-behavior specifications even for large interfaces.

4. Domain Models

A domain model is a representation of both the objects in a domain and their relationships. As such, a domain model may include a data model of the domain. In the same spirit that interface models provide developers with access to a higher-level representation of design knowledge, domain models also allow access to a level of representation higher than that of data models. Whereas data models establish a vocabulary to access the data structures of an application, domain models establish a vocabulary to access the objects in an application domain. As we will detail in this paper, domain relationships are a key to determining the dynamic behavior of user interfaces. With Mecano, we exploit the relationships defined in domain models to generate dialog and layout specifications for user interfaces. The result is a *theory* of how to map domain concepts to interface designs through a series of *mappings* connecting a domain object, a domain characteristic, or a domain relationship to an interface design element—such as a window navigation tree, an interaction style, or a dialog constraint.

Figures 3 and 4 show partial views of a model for the medical domain of therapy planning according to standard treatment *protocols*. We have defined this model using a frame-based representation language that defines class hierarchies (Gennari, 1993). This frame-representation language is used to define not only domain models, but also the interface modes that Mecano employs. Keeping both types of models in the same language improves shareability of the models with other groups, and facilitates the generation of interface-model instances from domain models.

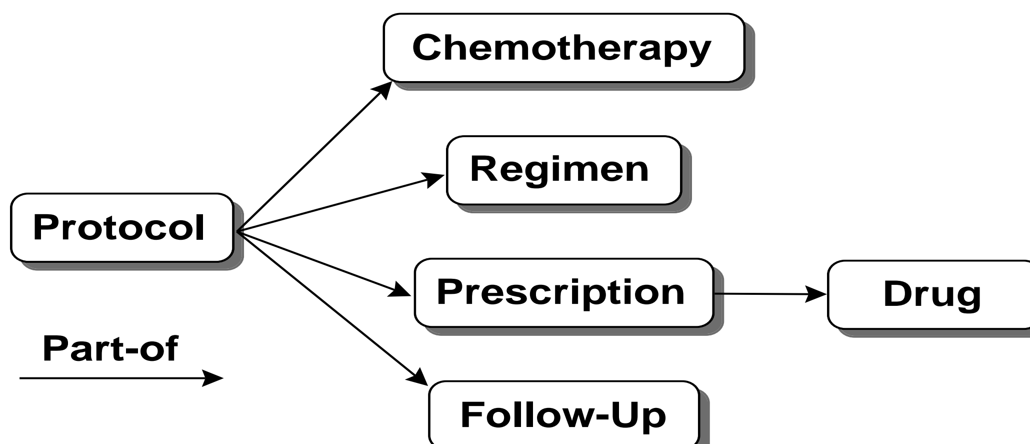

Figure 3. Partial view of a medical domain model for therapy (protocol) administration. The *part-of* hierarchy can be mapped to the window navigation tree of an interface—an example of the type of mappings domain-to-interface that are exploited by Mecano to generate interface designs.

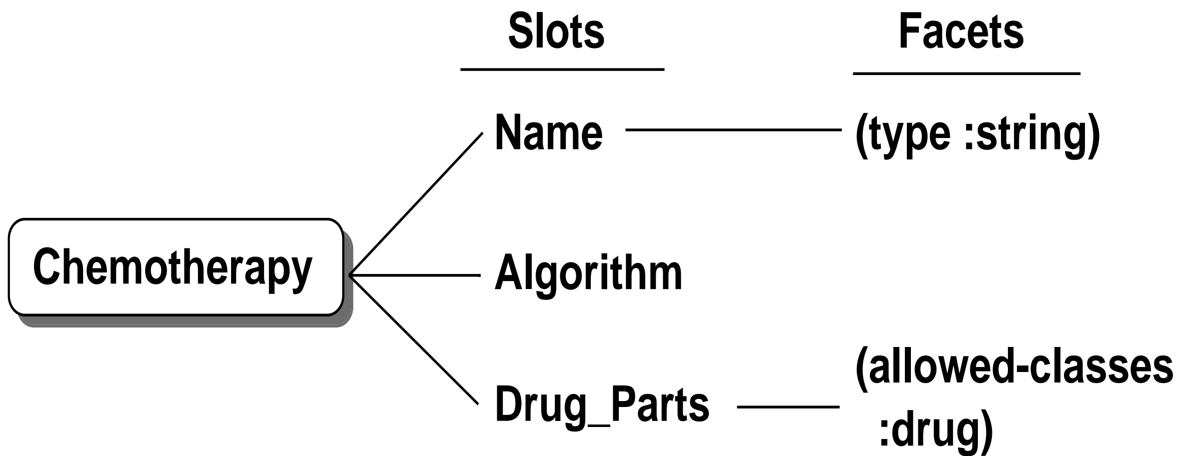

Figure 4. Partial view of the slots and facets (properties) for the *chemotherapy* class of Figure 3. The slot type can be mapped to an interaction style (e.g., type *string* to *text-field* object).

5. The Mecano Architecture

Figure 5 shows the major components of the Mecano architecture. This architecture follows the basic model-based system architecture shown in Figure 1 with some minor variations. The design-time tools include a model editor (for both domain and interface models), an intelligent designer to generate interface model instances (i.e., interface designs) from domain models, and an interface builder to custom-tailor the designs produced by the intelligent designer tool. There is no application-state monitor since this component is intended to be subsumed by the run-time system.

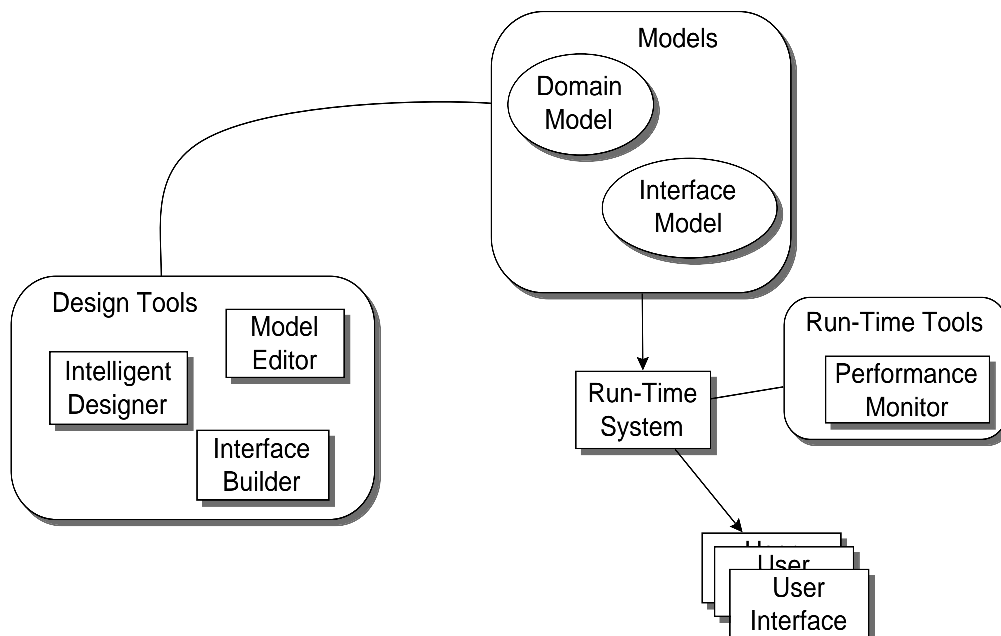

Figure 5. The main components of the Mecano architecture. The system delivers a development environment where all phases of interface construction, from design to maintenance, are supported. The architecture follows the basic model-based system architecture shown in Figure 1.

Mecano integrates design, development, and maintenance capabilities in a single environment. It manipulates sharable objects (domain and interface models) that can be used, in the spirit of ARPA's Knowledge Sharing Effort (Neches, 1991), by other groups to generate interfaces in their host environments. Mecano also provides a degree of platform independence by producing textual interface model instances (i.e., interface specifications) that are implemented by a run-time system. In this manner, textual specifications can be generated in Mecano, and such specifications can be implemented by an appropriate run-time system in a different platform.

The process of generating automatically interfaces within the Mecano context is depicted in Figure 6. The central concept is that of interface model *instantiation* where the intelligent-designer tool processes a given domain model and creates an application- and domain-specific *instance* of the generic interface model. The instance is created by applying a series of mappings between domain and interface characteristics. An interface model instance is a fully represented interface design that is application and domain specific. Once such an instance is available, it can be implemented as an interface by feeding it to a run-time system in the form of an interface specification written in a declarative language.

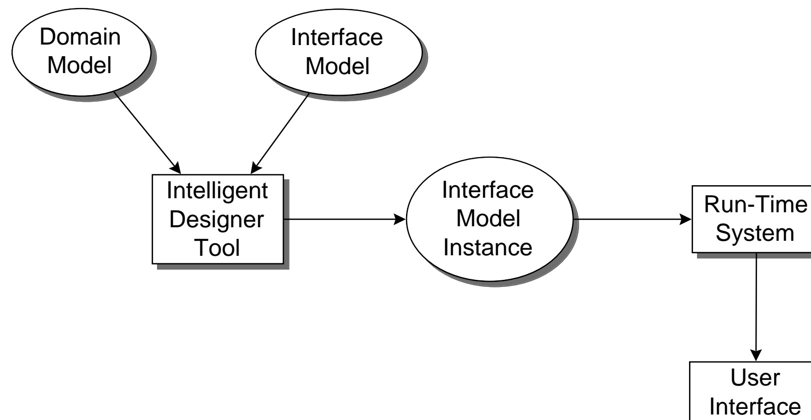

Figure 6. Generating interfaces in Mecano. The intelligent-designer tool *instantiates* the interface model for a given application and domain. It processes the domain model and creates the desired instance through a mapping of domain to interface characteristics. The run-time system implements the interface model instance by accessing an interface-specification language version of the instance.

6. Coupling Interface and Application Design

The goal of *separating* an application from its interface at the development level has been established as a sound software-engineering principle. The separation allows for a degree of encapsulation of the design of both elements and minimizes the impact of application software changes on interface software. The interface and its target application, however, still share a common data—or knowledge—representation. The application operates on the data and the interface allows end users to visualize those data. As a consequence, any changes that an application designer may make to a data model necessitate corresponding changes to the interface design. To propagate such changes more effectively, it is essential that both application design and interface design be *coupled*.

The application data model is the obvious first candidate for the basis for coupling both designs; that approach has been taken by some groups (deBaar, 1992; Janssen, 1992). Furthermore, researchers are now exploring domain models, rather than data models, as the foundation for

domain-specific software applications (Puerta, 1993b). These researchers, however, have concentrated on application design but not on user-interface design.

The Mecano approach to user interface generation has the added advantage of providing a direct way to achieve two highly desirable goals: coupling application design with interface design *and* automating the generation of *complete* interface designs—including static layout and dynamic behavior. Our laboratory is developing an environment to build domain-specific software architectures, called PROTÉGÉ-II (Puerta, 1993b) that uses domain models as the basis for application development. Mecano and PROTÉGÉ-II will constitute an environment in which the coupling of application design and interface design is realized.

7. Generating Interfaces with Mecano

The interface development cycle of Mecano is shown in Figure 7. After a domain and task analysis, a domain model is defined with the model editor shown in Figure 8. It is not necessary to build domain models from scratch for every application. For example, a domain model for medical therapy planning can be reused, with minor variations, in other applications. This is a significant advantage of Mecano over systems that design from data models because data models are difficult to reuse across applications.

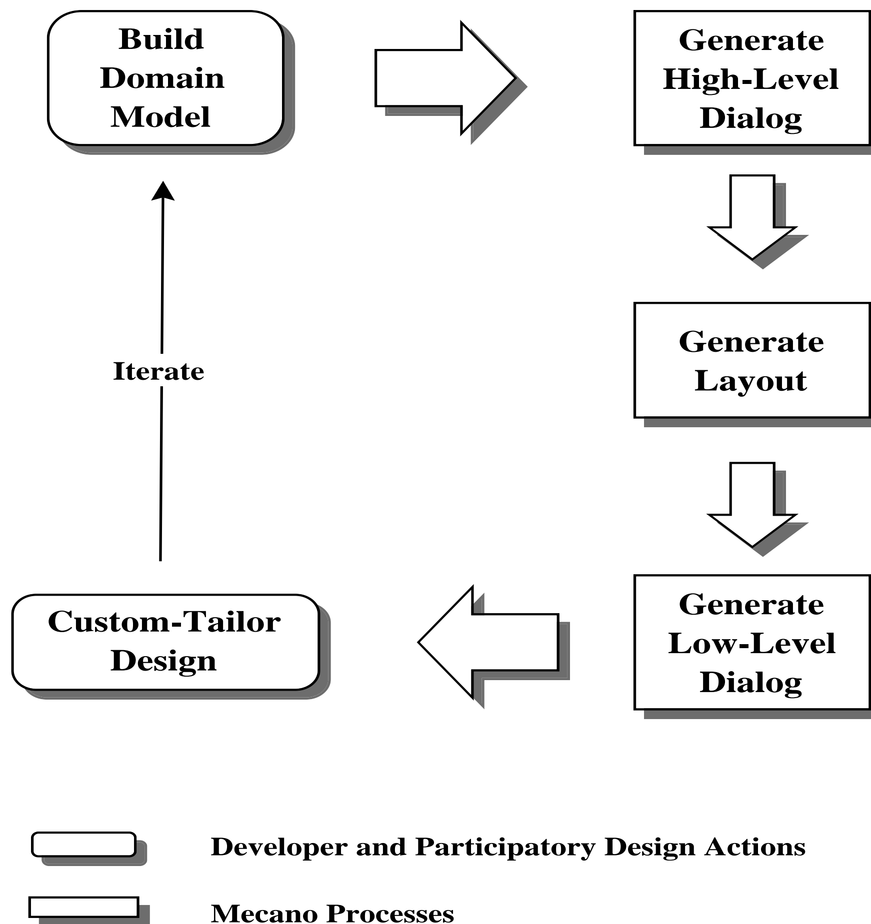

Figure 7. Mecano defines an iterative interface development process with a high degree of automation.

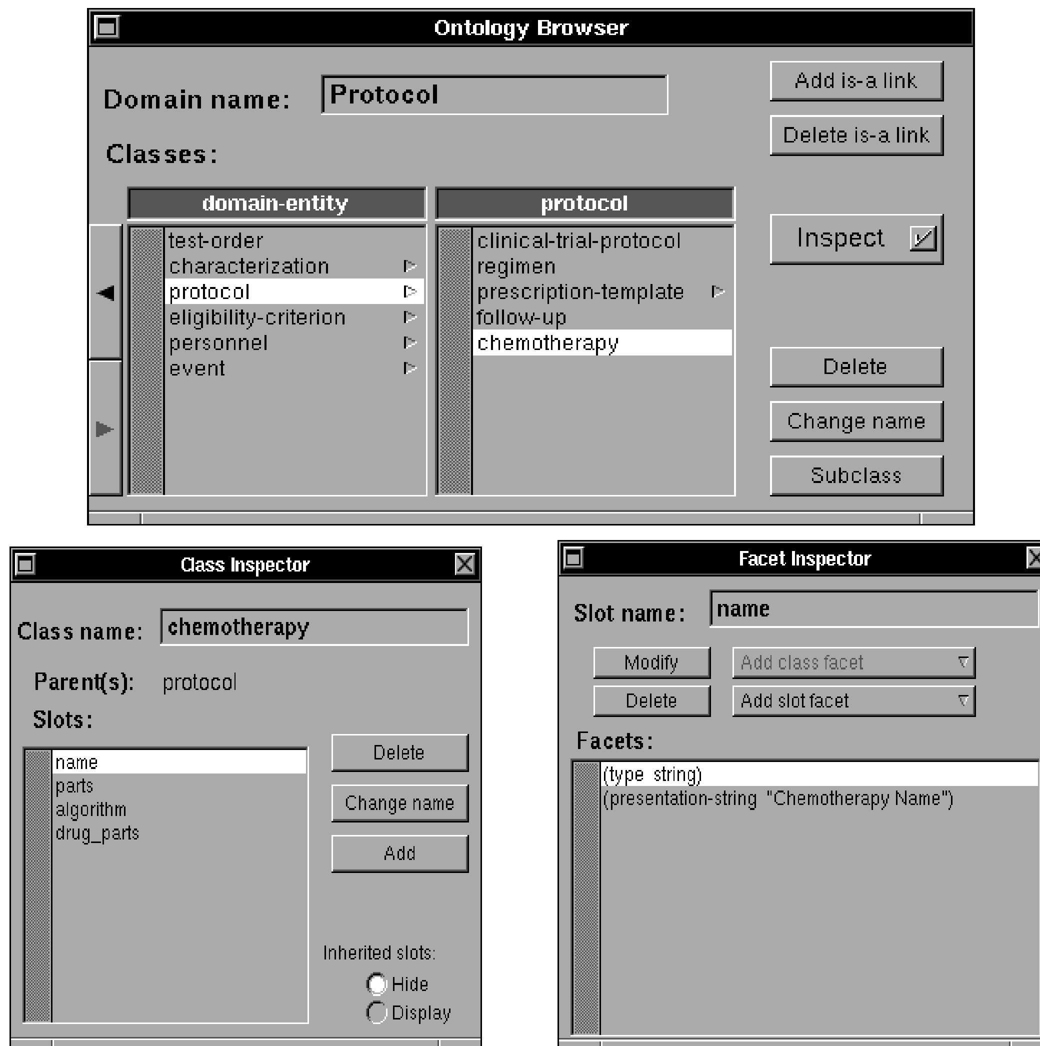

Figure 8. Developers edit domain models with a browser tool that allows definition, review, and inspection of models.

Once edited, the domain model is used to generate dialog specifications. These specifications have two levels in Mecano (Puerta, 1994; Puerta, 1993a):

- High-level dialog defines all interface windows, assigns interface objects to windows, and specifies the navigation schema among windows in the interface.

- Low-level dialog defines specific dialog elements (widgets) to each interface object created at the high level and specifies how the standard behavior of the dialog element is modified for the given domain.

7.1. High-Level Dialog Generation

The elements of the high-level dialog specification are generated by examining the class hierarchy of the domain model (see Figure 3) and the slots of each class (see Figure 4). Figure 9 shows an

interface generated from the partial domain model shown in Figures 3, and 4. The complete medical domain model for therapy administration generates an interface with over 60 windows and hundreds of widgets. Note that the dialog for window navigation is established during high-level dialog design but that it can be refined, or augmented, at low-level dialog design time. The procedure to generate a high-level dialog design is as follows:

- Each class in the hierarchy is assigned a window.

- Window navigation is established by searching the class hierarchy for links indicated by the *allowed-classes* facet in the domain model. For example, the *Drug* window shown in Figure 9 is accessed from the *Chemo* window because the Drug class is an allowed class for the slot *Drug_Part.*

- Each window is assigned one *interface object* per slot in the class. After generation, the developer has the option of customizing the interface by splitting windows multiple objects into two or more windows. Interface objects are assigned actual widgets during low-level dialog design.

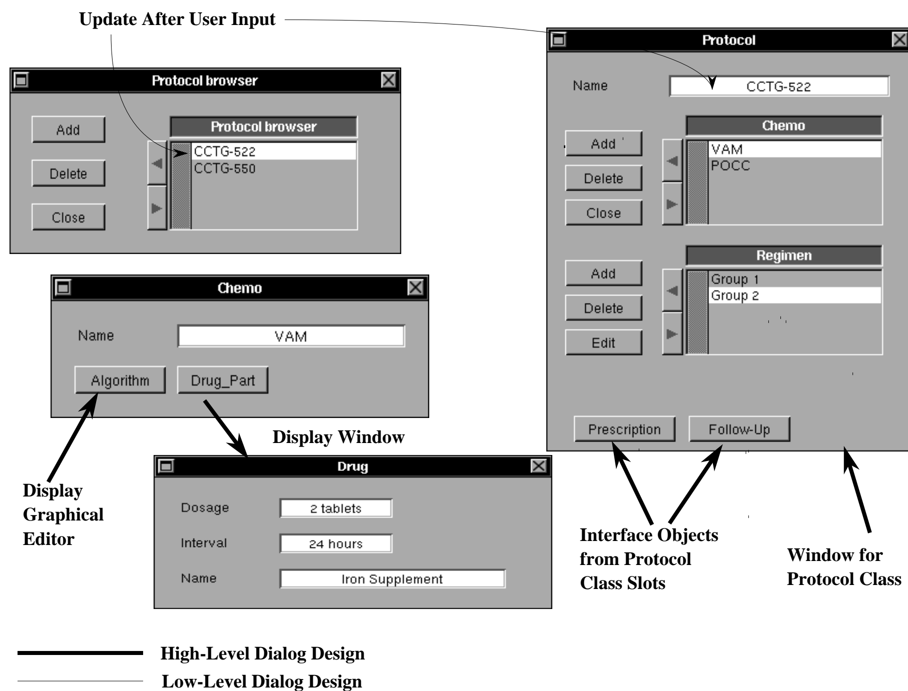

Figure 9. A form-based interface generated from the domain models partially shown in Figures 3 and 4. The interface generated from the full domain model for medical therapy consists of over 60 windows and hundreds of dialog elements.

7.2. Layout Generation

Layout generation is similar to that performed by other systems such as UIDE (deBaar, 1992; Gieskens, 1992) and GENIUS (Janssen, 1993) that use data models to generate the layout:

- Each interface object defined at high-level design time is assigned a dialog element (widget) by examining the facets of the corresponding slot in the domain model. For example an object of *type string* is assigned a text field, an object of *type Boolean* is assigned a check-box widget, and an object of *type string* and *cardinality multiple* (i.e., the object can be multiple-valued) is assigned a list browser.

- Each dialog element is placed on its corresponding window by a layout algorithm that observes interface design guidelines.

7.3. Low-Level Dialog Generation

Elements of the low-level dialog specification are generated by examining the facets (properties) defined for each slot in the domain model (see Figure 4). These facets include *part-of* relationships among classes.

- Each dialog element may be assigned *actions* beyond the standard behavior of the dialog element by examining the facets of the corresponding slot in the domain model. Examples of dialog-element actions include disabling editing in other dialog elements, and updating values in other dialog elements after a user input action (see Figure 9).

Note that the specification of dialog-element actions is one of the important operations that are not automated in systems that rely on data models for interface generation.

7.4. Layout and Design Revision

After Mecano produces generates an executable design, the developer conducts participatory design sessions with end users to custom tailor the design and make appropriate changes. Note that the required changes may necessitate editing of the corresponding domain model, and consequently, regenerating a new interface design.

Mecano provides facilities for reapplying any customizations done to a design before it was regenerated (Eriksson, 1994). Thus, Mecano allows developers to experiment with early prototypes and to quickly revise stable designs. Our experience in participatory layout revision with end-users is that working sessions, even for large interfaces, can be completed is a few hours at the most.

7.5. Domain-Specific Graphical Editors

In addition to the form-based interfaces shown in Figure 9, Mecano can generate layout and behavior specifications for domain-specific graphical editors. For example, consider the following slot information for the class *Protocol*:

```
(slot algorithm
    (type :procedure)
    (allowed-classes :xrt :chemotherapy :drug))
```

The intelligent designer tool in Mecano examines this slot and creates an abstract interface object for the slot, then due to the type procedure in the slot, it maps the interface object to a graphical editor as its dialog element. The tool also defines three graphical objects to be used during editing, one for *x-ray therapies* (xrt), one for *chemotherapies (chemo),* and one for *drugs* using information derived from the *allowed-classes* facet of the algorithm slot. Figure 10 shows a graphical editor generated from the above slot definition.

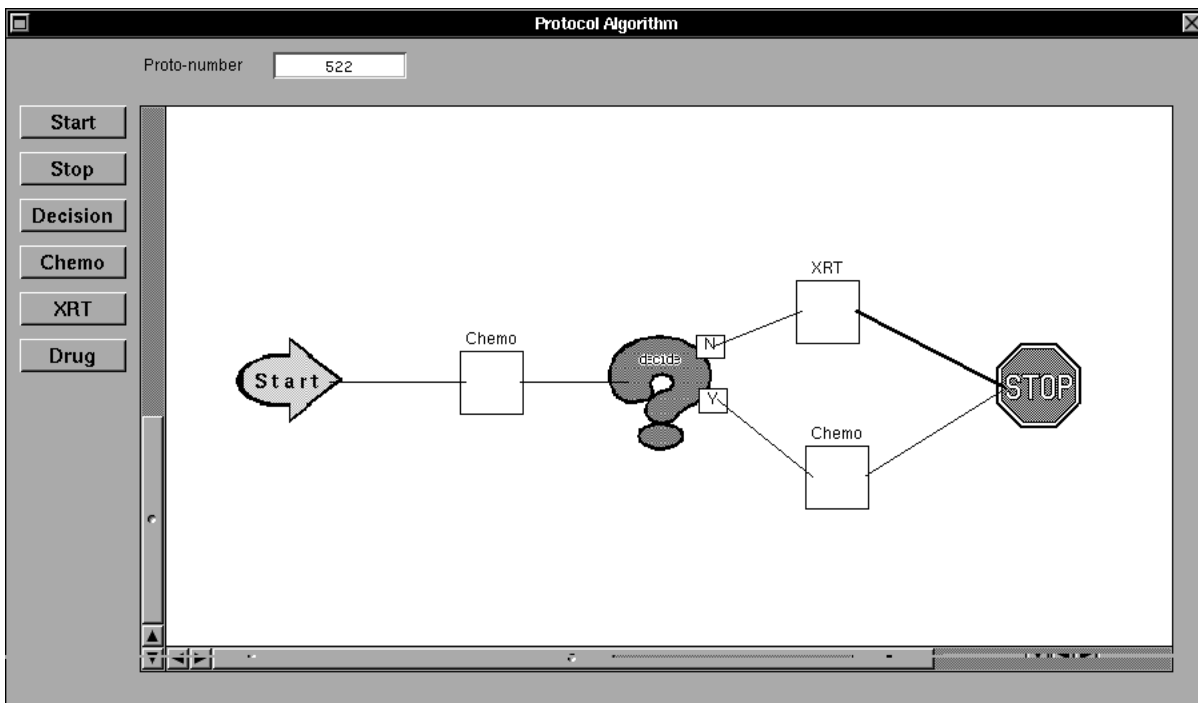

Figure 10. A graphical editor for the specification of medical procedures. The available drawing objects and the constraints on their interconnectivity at run-time are determined by Mecano during the dialog-generation phases.

8. Related Work

There are a number of model-based development environments reported in literature that are closely related to the Mecano effort.

The TRIDENT (Vanderdonckt, 1993) system bases the interface generation process on a user-task analysis that yields a user-task model and an application model that drive the generation process. TRIDENT automatically designs high-level dialog and presentation specifications that follow declarative interface design guidelines stored as part of the system's knowledge base. TRIDENT is geared toward assisting developers, in contrast to the automation emphasis on Mecano. Thus, users of TRIDENT must use extensively the available model editing tools to define in greta detail the task and application characteristics required for dialog generation. TRIDENT does not generate low-level dialog specifications.

Another development environment centered around a user-task model is ADEPT (Johnson, 1994). This system allows development of interfaces through an evolutionary process of editing a task model, generating a prototype, and refining the task model. As with TRIDENT, it requires extensive editing and the specification of interface actions directly through the task model editing tool—a step automated for the type of interfaces generated with Mecano.

The GENIUS environment (Janssen, 1993) uses an entity–relationship data model, along with a graphical editor for dialog specifications, to generate interfaces. The data model, which can be edited graphically, provides the basis for the definition of the interface components and their layout. The graphical editor allows the review of *dialog nets,* a variation of Petri nets, that define

the actions of the interface objects and the conditions that preclude or follow those actions. GENIUS is not designed to generate behavior specifications from high-level models (e.g., a task model).

The UIDE environment includes a tool for static layout generation from an extended data model (deBaar, 1992). The specification of dynamic behavior, however, must be achieved by defining sets of pre- and postconditions (Gieskens, 1992) for each one of the interface objects.

HUMANOID (Szekely, 1993) defines an elaborate interface model that includes components for the application, the presentation, and the dialog. Developers construct application models and HUMANOID picks among a number of *templates* of interfaces to display the interface. The developer can then refine the behavior of the interface by editing the dialog model. HUMANOID assists, but does not automate, the generation of dynamic behavior specifications, and requires considerable additional developer effort to generate interfaces that do not conform to its templates, as is the case with most complex interfaces.

9. Analysis and Conclusions

We have presented Mecano, a model-based development environment that extends the notion of using data models to drive interface-specification generation. The main advantages and contributions of Mecano are:

- Use of domain models to drive interface-specification generation. Domain models make explicit domain information and relationships that are not included in data models. Domain models are reusable across applications.

- Generation of both the static layout and the dynamic behavior of domain-specific, form- and graph-based interfaces, including relative large and complex ones, for multiple domains (e.g., medical treatment, elevator configuration).

- A highly automated design environment that supports the full development cycle of an interface while coupling interface and application design.

- Textual interface-model *instances* that can be made executable by multiple run-time systems, thus providing a degree of portability to the generated interfaces.

- A basic theory of how to amp domain characteristics to interface design specifications.

The automatic nature of the Mecano development process constricts the design space of the generated interfaces. We have emphasized participatory design revisions as part of the Mecano environment to balance in part the lack of human input in the generation phase. In addition, the generation of interfaces from domain models is most effective for interfaces with relatively fixed dialog structures—as is the case with the form- and graph-based interfaces supported by Mecano. Interfaces with highly complex and flexible dialog structures do probably need the existence of a detailed task model—such as those in ADEPT (Johnson, 1994)—to drive the generation of the interface. We are researching ways to combine the task modeling capabilities available in Mecano through its generic interface model with the domain-model driven approach to user interface generation.

Overall, Mecano provides a combination of comprehensive design and development support, level of automation, and portability of generated interfaces that should form the basis for continued research into complete and effective interface development environments.

Acknowledgments

This work has been supported in part by grants LM05157 and LM05305 from the National Library of Medicine, and by gifts from Digital Equipment Corporation. Dr. Musen is recipient of NSF Young Investigator Award IRI-9257578.

References

de Baar, D.J.M.J., Foley, J.D. and Mullet, K.E. (1992). Coupling Application Design and User Interface Design. In *Proceedings of Human Factors in Computing Systems, CHI'92*. Monterey, California, May 1992, pp. 259–266.

Eriksson, H., Puerta, A.R. and Musen, M.A. (1994). *Generation of Knowledge-Acquisition Tools from Domain Ontologies.* In Proceedings of the Eighth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop. Banff, Canada, February 1994. pp. 7.1–7.20.

Gennari, J.H. (1993). *A Brief Guide to Maître and MODEL: An Ontology Editor and a Frame-Based Knowledge Representation Language.* Stanford University, Knowledge Systems Laboratory, Report KSL-93-46, Stanford, USA. June 1993.

Gieskens, D.F. and Foley, J.D. (1992). Controlling User Interface Objects through Pre- and Postconditions. In *Proceedings of Human Factors in Computing Systems, CHI'92*. Monterey, USA, May 1992, pp. 189–194.

Janssen, C., Weisbecker A. and Ziegler J. (1993). Generating User Interfaces from Data Models and dialog Net Specifications. In *Proceedings of Human Factors in Computing Systems, INTERCHI'93*. Amsterdam, The Netherlands, April 1993, pp. 418–423.

Johnson, P., Wilson, W., and Johnson, H. (1994). Scenarios, Task Analysis, and the ADEPT Design Environment. In J. Carroll (ed) *Scenario-Based Design.* Addison-Wesley. (In Press).

Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T., and Swartout, W. (1991). Enabling Technology for Knowledge Sharing. *AI Magazine,* **12** (3), pp. 36–56

Puerta A.R. (1993a). The Study of Models of Intelligent Interfaces. In Proceedings of the 1993 International Workshop on Intelligent User Interfaces. Orlando, USA, January 1993, pp. 71–80.

Puerta, A.R., Tu, S.W., and Musen, M.A. (1993b). Modeling Tasks with Mechanisms. *International Journal of Intelligent Systems,* **8**(1), pp. 129–152.

Puerta, A.R., Eriksson, H., Gennari, J.H., and Musen, M.A. (1994). Model-Based Automated Generation of User Interfaces. In *Proceedings of the Twelfth National Conference on Artificial Intelligence, AAAI94.* Seattle, USA, August 1994.

Szekely, P., Luo, P. and Neches, R. (1993). Beyond Interface Builders: Model-Based Interface Tools. In *Proceedings of Human Factors in Computing Systems, INTERCHI'93*. Amsterdam, The Netherlands, April 1993, pp. 383–390.

Vanderdonckt, J.M., and Bodart, F. (1993). In *Proceedings of Human Factors in Computing Systems, INTERCHI'93*. Amsterdam, The Netherlands, April 1993, pp. 424–429.