Towards a General Computational Framework for Model-Based Interface Development Systems

Angel Puerta and Jacob Eisenstein
Stanford University
251 Campus Drive – MSOB x215
Stanford, CA 94305-5479 USA
puerta@smi.stanford.edu, jacob1@leland.stanford.edu,
http://www.smi.stanford.edu/projects/mecano

ABSTRACT
Model-based interface development systems have not been able to progress beyond producing narrowly focused interface designs of restricted applicability. We identify a level-of-abstraction mismatch in interface models, which we call the mapping problem, as the cause of the limitations in the usefulness of model-based systems. We propose a general computational framework for solving the mapping problem in model-based systems. We show an implementation of the framework within the MOBI-D (Model-Based Interface Designer) interface development environment. The MOBI-D approach to solving the mapping problem enables for the first time with model-based technology the design of a wide variety of types of user interfaces.

Keywords
Model-based interface development, interface models, knowledge-based user interface design, user interface development tools

INTRODUCTION
Model-based systems for user interface development [7] exploit the idea of using a declarative interface model to drive the interface development process. An interface model represents all the relevant aspects of a user interface in some type of interface modeling language. Model-based systems provide the software tools that build and refine the interface model to produce a user interface. Objects typically included in a comprehensive interface model are user tasks, domain elements, users, presentation items, and dialog structures. Because of the nature of the objects in an interface model, model-based systems suffer from a challenging level-of-abstraction problem. On one hand, there are purely abstract units in an interface model, such as a user task (e.g., "get customer's name"). On the other hand, there are very concrete units, such as scrollbars and pushbuttons, which are part of an interface presentation.

The development cycle of model-based systems normally starts by building an abstract model. This can take the form of a user-task model, a domain model, or an integration of both. The tools in model-based systems then attempt to produce automatically a concrete interface design (i.e., a presentation and dialog) from the abstract representation. Researchers have found limited success with this approach. Some model-based systems have proven reasonably effective in narrow target application domains including, for example, automatic generation of forms, or automatic generation of dialog boxes for database access [2, 5]. However, no technique has been shown to be applicable at a general level. The main reason for this shortcoming is that we do not have a general computational framework to bridge the abstract-to-concrete gap in interface models. We lack an understanding of what features and attributes of the abstract elements in an interface model are relevant in creating a link to the concrete elements in that model. We call the problem of linking abstract and concrete elements in an interface model *the mapping problem*. Solving the mapping problem in a general sense is essential for the construction of model-based systems of wide applicability in user interface design.

In this paper, we present an initial general solution to the mapping problem in model-based interface development. We identify salient features in abstract elements of an interface model and discuss its potential in determining mappings to concrete elements. We also identify the most important types of mappings in user interface design and describe the variables that can affect the decision of mapping one object to another in an interface model. We have found that the mapping problem in model-based systems defies automation because of the number of variables that can impact each possible mapping. Instead of automation, we propose that model-based systems provide tools that allow developers to interactively set the mappings. The tools should assist the developer in pruning the design space of potential mappings into a manageable

set. Finally, we present a set of prototype tools that support interactive mapping of abstract to concrete objects in an interface model within the MOBI-D (Model-Based Interface Designer) development environment [7]. Our general strategy for attacking the mapping problem enables MOBI-D to target a wide range of application domains and user-interface types.

INTERFACE MODELS

An *interface model* is an ordered collection of all the relevant elements of a user interface. Interface models are declarative and are written in an *interface modeling language*. The elements of an interface model are grouped into *model components*. The basic components are the user-task model, the user model, the domain model, the presentation model, and the dialog model. Interface models are referred to as *partial models* if they include just some of the basic components and as *comprehensive models* if they include all of the basic components. A brief synopsis of each of the components is as follows:

- *User-task model.* A user-task model is a description of the task to be accomplished by the user of an application through the application's user interface. Individual elements in the user-task model represent specific actions that the user may undertake. Information regarding subtask ordering (e.g., sequence, unordered, optional sequencing) as well as conditions on task execution is also included in this model.
- *Domain model.* A domain model defines the objects that a user can view, access, and manipulate through a user interface. In nature, it is very similar to an application's data model but it is also intended to explicitly represent the attributes of objects and the relationships among the various domain objects. Therefore, domain models are more ontological in essence than data models.
- *User model.* A user model represents the different types of users of a target application. It is not a cognitive model but a definition of the attributes and roles of users.
- *Presentation model.* The presentation model is a representation of the visual, haptic, and auditory elements that a user interface offers to its users. For example, a presentation element might be a window that contains additional elements such as widgets that appear in that window. The presentation model also includes presentation attributes, such as font styles and orientation of button groups. On its own, the presentation model is only a static collection of sensory elements.
- *Dialog model.* The dialog model defines the way in which the presentation model interacts with the user. It represents the actions that a user may initiate via the presentation elements and the reactions that the application communicates via those same elements.

THE MAPPING PROBLEM

The main function of a model-based interface development system is to provide the software tools that allow developers to construct user interfaces by means of creating and refining an interface model. In general terms, developers use model-based systems to first define one or more of a user-task model, a domain model, and a user model. The model-based systems then attempt to generate from those model-components presentation and dialog models that are then converted into an executable interface specification (i.e., a running user interface). A knowledge-based process supports the generation of the user interface.

The success of model-based systems has been limited. On one hand, there are systems that can generate specific-type interfaces (e.g., form-based interfaces, database interfaces) [2, 5] with a high degree of automation. On the other hand, none of the knowledge-based approaches for interface generation used by model-based systems is applicable beyond its intended narrow target domain nor can they be generalized to other targets. Furthermore, interfaces produced by these systems look all fairly similar and developers have little flexibility to change them in any of its fundamental aspects.

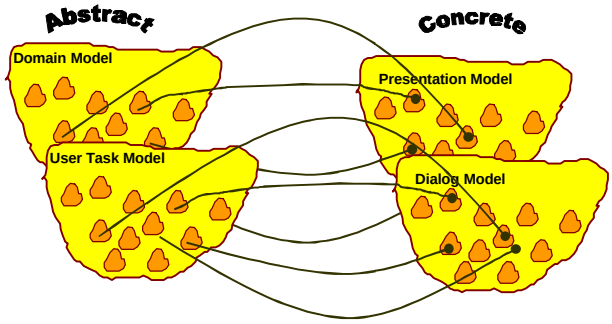

Figure 1. The mapping problem in interface models.

To understand the roots of these limitations, it is necessary to study the nature of interface models. For the purpose of our discussion, we will divide the components and elements of an interface model into *abstract* and *concrete* categories. We call concrete those elements of a running user interface that a user can access directly. Thus, windows, push buttons, mouse clicks, and audio are concrete elements. In an interface model, concrete elements can be found in the presentation and dialog model components. In contrast, we call abstract those elements of a running user interface that the user can access only indirectly, or not at all. Thus, user tasks and data objects are abstract elements. In an interface model, abstract elements can be found in the user-task, domain, and user model components. Note that the distinction is somewhat arbitrary. If a text field widget in a user interface displays the value of a data object (e.g., a string

value) we say that the user has direct access to the text field widget but indirect access to the data object.

Under the abstract/concrete point of view, the process of generating a user interface in a model-based system can be seen as that of finding a concrete specification given an abstract one. For example, given user-task *t* in domain *d* find an appropriate presentation *p* and dialog *D* that allows user *u* to accomplish *t*. Therefore, the goal of a model-based system in such a case is to link *t*, *d*, and *u* with an appropriate *p* and *D*. We call this challenge of linking the abstract and concrete elements *the mapping problem* in model-based systems, as depicted in Figure 1. Furthermore, we claim that the limitations of model-based systems are due to the lack of a general solution to the mapping problem for all interfaces, or at least to the lack of availability of a general framework to search for solutions to the mapping problem for individual interfaces.

Model-based systems that generate specific-type interfaces in essence embed into their knowledge-based approach a single way, or single method, of mapping abstract to concrete elements. Clearly, any user interface design that requires a set of mappings that cannot be produced by that single method is therefore unrealizable. However, the challenge of developing general, or multiple, mapping methods is considerable given the level-of-abstraction mismatch in interface models. This difficulty has led some researchers to determine that the future of model-based systems is simply to target specific-type interfaces [9]. We will show here, however, that a general framework for mapping can be developed and supported via software tools in a model-based system. With such a framework, developers are able to design a multitude of types of user interfaces.

TYPES OF MAPPINGS

Before deciding what type of software tools can best support solving the mapping problem, it is important to analyze whether there are certain types of mappings among the elements in an interface model that have particular importance. Clearly, if for a given user interface design it is potentially meaningful to map any abstract interface model element to any concrete one, then we would probably be facing a nearly insurmountable computational problem. Fortunately, the intrinsic nature of each interface model component determines to a large extent the kind of mappings that are possible to and from that component. In this section, we identify the most important types of mappings in model-based interface design.

Task-Dialog Mappings

Earlier, we described user-task models as an ordered collection of elements representing user tasks. User task models are typically arranged into hierarchical task/subtask decompositions. Groups of tasks can be arranged by order of execution (e.g., sequential, parallel). Conditions can be attached to the execution of tasks and input/output requirements can be specified for any task or subtask. Therefore, user-task models provide two types of information about tasks: structural and procedural.

Dialog models define the conversation between users and the interface. These models establish a navigation schema, define what are the accepted user actions, and determine what interaction techniques are applicable in each instance of a user action. Therefore, dialogs are procedural in nature. This establishes a parallelism with the user-task model and hints at potential mappings between the two model components. Some of the potential mappings are (1) task execution-order to navigation order, (2) conditions on task execution to enable/disable states in the dialog, and (3) input/output requirements for tasks to input/output requirements for command execution.

Task-Presentation Mappings

A presentation model defines the parts of a user-interface presentation as well as the arrangement and grouping of those parts. As such, it is structural in nature and suggests possible mappings to the structure of user tasks. In particular, task/subtask decompositions should map (likely not one-to-one) to part/subpart hierarchies in a presentation. In addition, subtask groupings in the user-task model should map to subpart groupings in the presentation model. An example of this would be the assignment of a task group to a window where a user can complete all the subtasks included in the group.

Domain-Presentation Mappings

Elements in a domain model possess *attributes* that are often relevant to presentation element selection. Clearly, the nature of a domain object is the determining factor in how to present and make that object available to a user in an interface. A domain object typically possesses a number of *attributes*, including a *data type*. This suggests that the mappings between a domain model and a presentation model are essential in any user interface design. The mappings establish, for example, what widget should be used to display the value of an integer-type object. Not only the associated data type may influence this selection but also other attributes such as range, or minimum and maximum values can play a role.

Task-User Mappings

So far we have looked at mappings across levels of abstraction but mappings restricted to just the abstract or just the concrete levels in an interface model are also possible. A user model may specify a number of types of users of the target interface. Each user may be involved in all tasks in a user-task model, or just in a subset of these tasks. The assignment of users to tasks is a mapping process. If two or more types of users are assigned to the same subgroup of tasks, this has a *multiplier* effect on the

mappings to presentations and dialogs from task and domain models. In effect, each user may require a different presentation and dialog and appropriate mappings should be established between the abstract and concrete levels in the interface model to account for those requirements.

Task-Domain Mappings

An interface model must also define what objects are involved in the completion of the tasks represented in its user-task model component. Thus, it is necessary to map objects to tasks in an interface model. The assignment of objects to tasks has an *integration* effect on the mappings between the presentation and dialog model components as discussed next.

Presentation-Dialog Mappings

Clearly, the presentation elements and the dialog elements in an interface model must be linked to each other to specify a running user interface. As we discussed, there are important links between tasks and dialogs and between domains and presentations. Naturally, we should expect that the mapping of domain objects to tasks would influence how the presentation and dialog elements are mapped among themselves. In general, we would expect that presentation-dialog mappings parallel the corresponding task-domain mappings.

AUTOMATION ISSUES

To achieve an automated general solution to the mapping problem, it would be necessary to first identify all the variables that affect the setting of the various types of mappings. If we then are able to quantify or qualify these variables, it might be possible to implement knowledge-based approaches that solve the mapping problem for each instance of design of a user interface.

Unfortunately, it does not appear that we are able to satisfy the prerequisites named above. Mappings in an interface model are affected by the same basic factors that affect the design of user interfaces. Therefore, many of these factors are intangible and resist a computational implementation. Foremost among these factors are human creativity and artistic qualities. An interface layout design can be as much an expression of the visual artistic capabilities of a designer as it is a rational selection of widgets based on data type definitions.

Even when a purely rational approach is used, developers can face a multitude of choices for each mapping to be set. There might be several widgets in a toolkit library that would be effective to display a domain object, or there might be numerous ways of distributing tasks among windows, or there might be different ways to group widgets in a window, and so on.

The approach of model-based systems for specific-type interfaces has been to remove all issues of creativity and art from the mapping process, and to severely limit the choices that can be made in a rational manner. It is not surprising then that the interfaces produced by these systems are restricted to one look and feel, and to one design pattern. And it is even less surprising that the methodology for interface generation used in these model-based systems cannot be generalized since it does not account for a majority of the factors that affect mapping.

The ideal approach to automate the solutions to the mapping problem would be one that supported the choices of rational interface design as well as the creativity of artistic interface design.

THE MOBI-D APPROACH

MOBI-D (Model-Based Interface Designer) is a general-purpose model-based interface development environment. It supports the definition and refinement of comprehensive interface models via a number of software tools including model editors, task-based interface builders, user-task elicitation tools, and knowledge-based design assistants. The functionality of MOBI-D and its development cycle are described elsewhere [6, 7]. In this paper, we will concentrate on describing the solution to the mapping problem implemented by MOBI-D.

The approach of MOBI-D is not to embed into the system any particular method of setting the mappings. Instead, MOBI-D allows interface developers to directly access and set the mappings according to their needs. It can also provide knowledge-based assistance to developers in setting the mappings. The goal is to provide a general computational framework that supports a rational approach to solve the mapping problem for individual interface designs without limiting the freedom that interface designers need to explore design options.

In order to support the inspection and setting of mappings, MOBI-D extends the definition of an interface model to include a new model component called the *design model*. A design model is a declarative representation of all the mappings in a user interface. In the same manner that having declarative representations of user-tasks, domains and so forth enables the construction of supporting software tools, so it does the design model. MOBI-D includes tools for direct manipulation of mappings and for knowledge-based assistance in setting mappings. We will examine here two of the tools: a design model editor and an intelligent mapping assistant.

Design Model Editor

Figure 2 shows the design model editor in MOBI-D. In the left pane, developers can inspect the entire contents of each basic model component in the MOBI-D interface model (i.e., user-task, domain, user, presentation, and dialog). In the right pane, developers can view the current

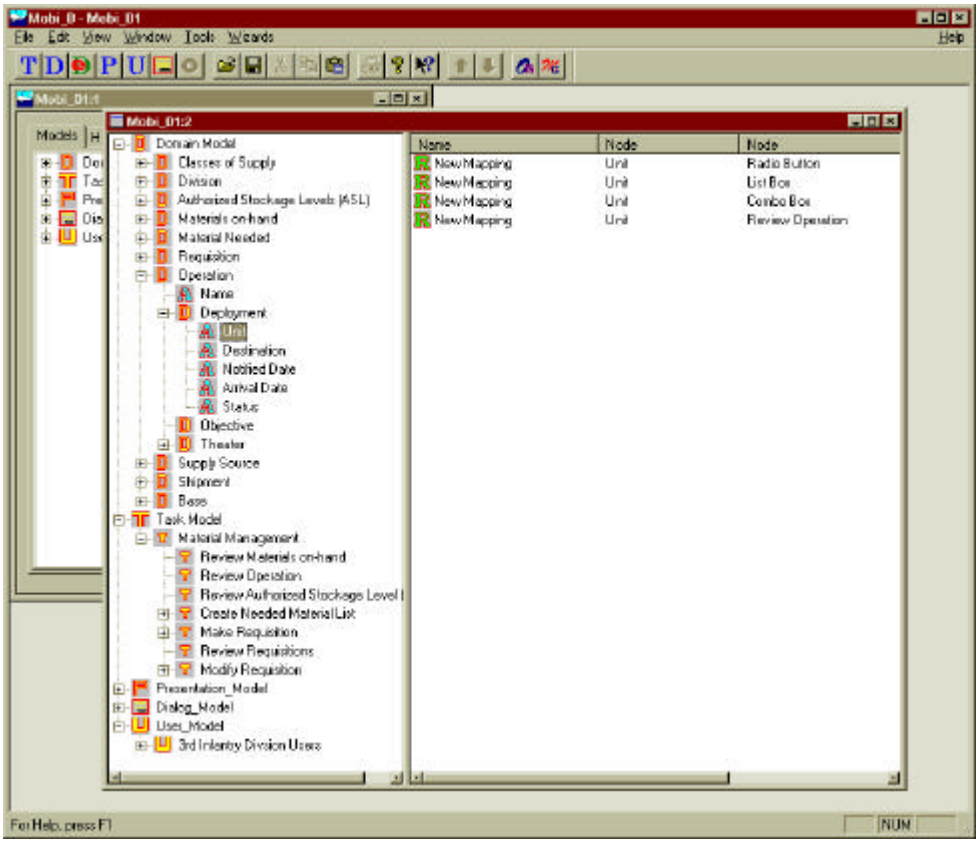

Figure 2. The MOBI-D design model editor.

mappings for a selected element on the left pane. The tool is designed for maximum freedom. To set a mapping, a developer simply drags an element of any of the model components and drops it onto the intended target element. The semantics of establishing a mapping are dependent on the type of elements involved. For example, mapping a user type to a user task means such user type performs the target user task. In contrast, mapping a domain object to a user task means that such object is used in performing the target user task. Developers can annotate and further specify the nature of a mapping.

TIMM: The Interface Model Mapper

The design model editor is effective in allowing developers to set any type of mappings they wish to set. However, it provides no guidance in how to set mappings. Typically, developers use the design model editor to set same-level-of-abstraction mappings (e.g., users to user-tasks and domain objects to user tasks). For the more complex abstract-to-concrete mappings, MOBI-D provides a decision-support tool called TIMM (The Interface Model Mapper).

In the process of building user interfaces with MOBI-D, developers first define user-task, domain, and user models for the target interface. These models are integrated via mappings done in the design model editor. At the time of deciding what presentation and dialogs should be used given the integrated model built, developers are faced with a myriad of options in mapping from the abstract level to the concrete level in the interface model. The role of TIMM, as depicted in Figure 3, is to assist developers in navigating the design space of abstract-to-concrete mappings. TIMM can prune the design space of mappings down to a manageable set that the developer can then explore to make final decisions. The types of mappings that TIMM currently supports are domain-to-presentation, task-to-dialog, and task-to-presentation.

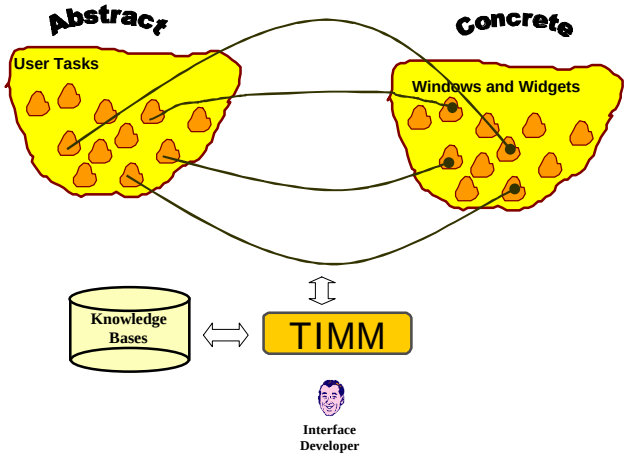

Figure 3. A role of the Interface Model Mapper (TIMM).

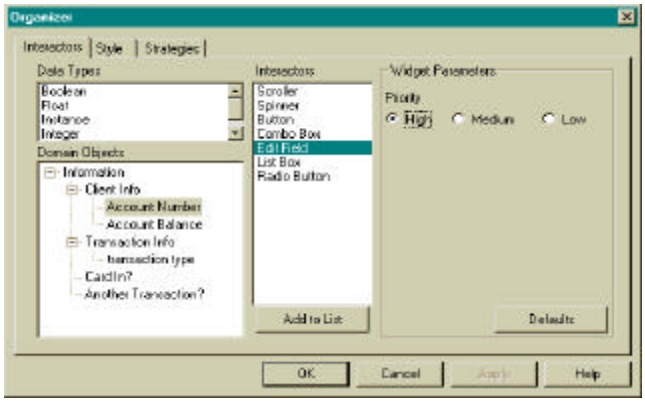

Figure 4. Domain-to-Presentation mappings in TIMM

Figure 4 shows the TIMM facility to inspect and set domain-to-presentation mappings. The lower left pane is a view of the domain model while the center pane is a list of presentation elements (called *interactors* in MOBI-D). By selecting an object in the domain model, developers can view and modify what interactors are appropriate to display and access that particular domain object. TIMM uses a knowledge base of interface design guidelines to examine the attributes of a domain object and build a list of potential interactors. The interactors in the list may be arranged in order of *priority*, that is how desirable is it to use one interactor versus another given the current set of interface design guidelines.

Because of the special role of the data type attribute in determining domain-to-presentation mappings, TIMM provides a view (upper left pane) to inspect and modify global mappings between data types an interactors. A change in one of these mappings affects the list of potential interactors for all objects in the domain model of that data type. Developers are free to set their own mappings or to accept the mappings created by TIMM. Advanced users of TIMM can also access and modify the knowledge base of interface design guidelines. In this manner, TIMM affords complete freedom to developers to sets the mappings but can also reduce the design possibilities to reasonably sized set that developers can deal with effectively.

Figure 5 shows the TIMM facility to inspect and set task-to-dialog and task-to-presentation mappings. Item on this panel are either structural (task-to-presentation) or procedural (task-to-dialog) in nature. An example of a procedural item is the sequence enforcement setting. In a user-task model a group of subtasks may be specified as a sequence. But at the user interface level, the sequencing can be enforced in several ways For example, in a sequence of *n* tasks the interface may hide completely the areas for completion of subtasks *2-n* until the user completes subtask *1*. Alternatively, the interface may leave visible but disable the areas for subtasks 2-n. A third option could be to enable all areas but to display an error message if a subtask is left out. Using TIMM, a developer can specify what global *strategy* to follow to set task-to-dialog mappings for task execution.

An example of a structural item is the setting for "number of windows". Given a user-task model, which is a task/subtask decomposition, there are multiple options as to how to split those tasks among, say, windows in a user interface. User-task models in MOBI-D are tree structures with varying levels of depth. The slider present in TIMM to adjust the number of windows has discrete values matching the depth level of the current user-task model tree. Using the slider, developers set a strategy for dealing with task-to-presentation mappings for grouping in an interface model.

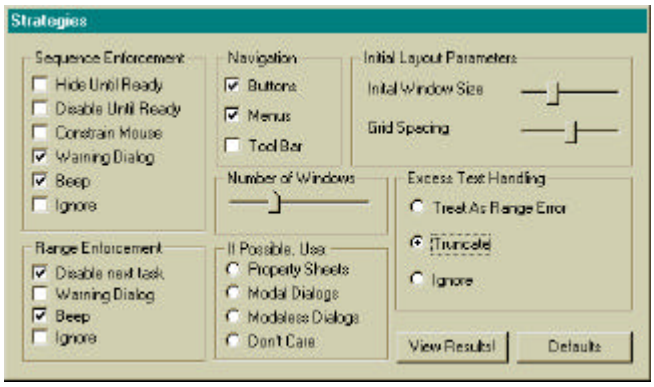

Figure 5. Mappings from tasks to dialogs and presentations in TIMM.

Once developers finish working with TIMM, they move to a different MOBI-D tool to complete their interface design. This tool is a task-based interface builder similar to commercial interface builders but customized for each particular interface design according to the sets of mappings and the strategies developed in TIMM [7]. Using this tool, developers make the final design decisions and set the concrete-to-concrete mappings between design and presentation.

As it can be seen, the tools in MOBI-D allow developers to first set abstract-to-abstract mappings, then assist in jumping from the abstract to the concrete levels, and finally direct the setting of concrete-to-concrete mappings. The result is the ability of developers to undertake a wide variety of interface designs in MOBI-D.

SAMPLE INTERFACES

Figure 6 shows one of the screens of an interface for logistics activities (e.g., reviewing stocks, requesting shipments, and planning levels of supplies) in a military theater of operations. The interface adapts its views to users of different ranks and modifies its dialog and sequencing according to changes in situation changes (e.g., bad weather delaying a shipment). Up until now, this type of interface design was out of the reach of model-based systems. It has multiple presentation modalities, complex widgets (e.g., 3-D viewers, and interactive maps),

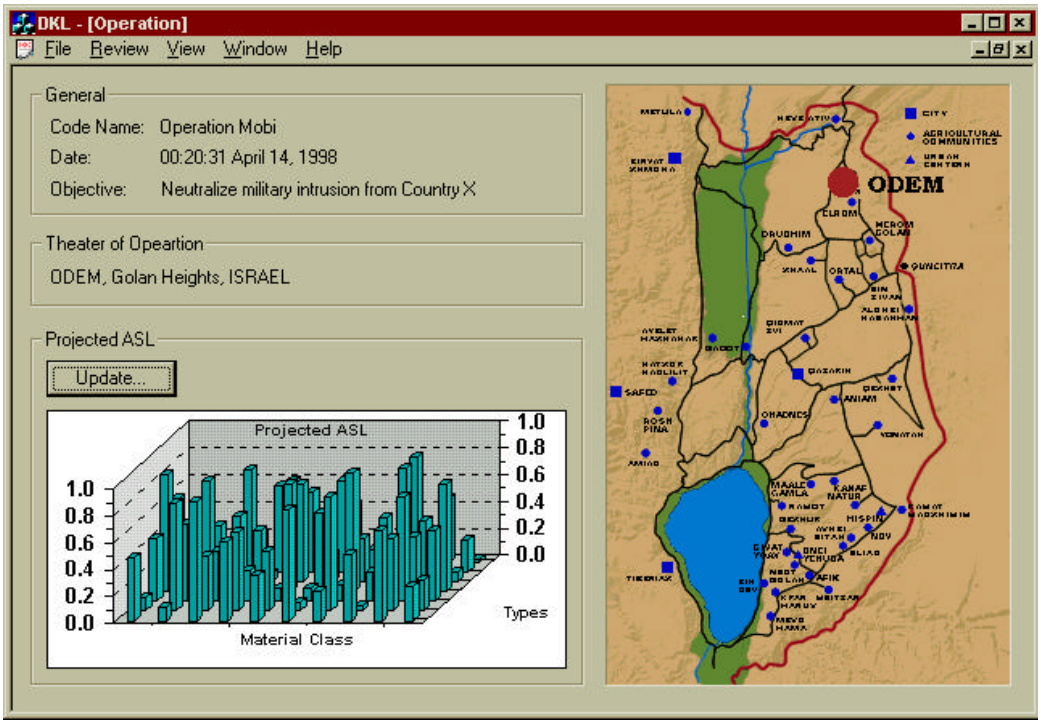

Figure 6. An adaptive logistics interface designed with MOBI-D.

and support complicated tasks. However, with MOBI-D this type of design is enabled thanks to the ability to freely map the abstract elements of the interface model with the concrete ones.

RELATED WORK

Over the years, there have been a number of model-based systems built. Here we introduce some of the most relevant ones. In general, these systems share these limitations when compared to MOBI-D: (1) focus on automatic generation of interfaces and therefore in a single mapping method, (2) failure to represent mappings as declarative elements of the interface model, (3) lack of interactive tools that allow developers to explicitly inspect and set mappings. As a result of these limitations, these systems are restricted to specific-type interface designs and fail to solve in any general sense the mapping problem.

UIDE [1] is one of the first model-based systems to be built. It included a partial interface model where presentations were generated from data models. An algorithm assigned data types to widgets and laid them out on a canvas. Therefore, UIDE dealt mainly with domain-to-presentation mappings and did so in an automated way. Another system that exploited domain-to-presentation mappings automatically was Mecano [5]. This system used domain models to generate form-based interfaces but lacked any notion of a user-task model.

ADEPT [3], FUSE [4], Tadeus [8], and Trident [11] all embed various forms of task models. The philosophy of these systems was to try to automate as much as possible the interface generation process from a user-task model. Therefore, the systems embed their mapping method into their knowledge-based approach. It should be noted that although ADEPT does not explicitly represent mappings, it does identify that the are abstract and concrete levels of abstraction in an interface model.

HUMANOID [10] applied templates to bridge the domain-to-presentation mappings, which proved to be an effective way of building some types of interfaces. However, the approach cannot be generalized outside of its intended scope.

CONCLUSIONS

We have presented a general framework to solve the mapping problem in model-based interface development systems. We identify the nature of the mapping problem as one of bridging levels of abstraction in an interface model. By explicitly representing mappings in an interface model, by providing tools that allow developers to set and inspect the mappings, and by affording developers knowledge-based approaches to prune the design space of potential mappings, the MOBI-D interface development environment enables the design of a wide variety of user interfaces previously unattainable using model-based technologies.

Clearly, MOBI-D deals with only a few of the interesting mapping situations in any user interface design. However, the MOBI-D environment defines a new approach and philosophy to model-based systems, one that potentially can lead to a much wider use of the technology.

ACKNOWLEDGMENTS

The work on MOBI-D is supported by DARPA under contract N66001-96-C-8525. We thank Hung-Yut Chen, Eric Cheng, James J. Kim, Kjetil Larsen, David Maulsby, Justin Min, Dat Nguyen, David Selinger, and Chung-Man Tam for their work on the implementation and use of MOBI-D.

REFERENCES

1. Foley, J., *et al.*, *UIDE-An Intelligent User Interface Design Environment*, in *Intelligent User Interfaces*, J. Sullivan and S. Tyler, Editors. 1991, Addison-Wesley. p. 339-384.
2. Janssen, C., Weisbecjer, C., and Ziegler, J. *Generating User Interfaces from Data Models and Dialogue Net Application*, in Proc. of *InterCHI'93*. 1993: ACM Press.
3. Johnson, P., Wilson, S., and Johnson, H., *Scenarios, Task Analysis, and the ADEPT Design Environment*, in *Scenario Based Design*, J. Carrol, Editor. 1994, Addison-Wesley.
4. Lonczewski, F. *Providing User Support for Interactive Applications with FUSE*, in Proc. of *IUI97*. 1997: ACM Press.
5. Puerta, A. and Eriksson, H. *Model-Based Automated Generation of User Interfaces*, in Proc. of *AAAI'94*. 1994: AAAI Press.
6. Puerta, A. R. *The MECANO Project: Comprehensive and Integrated Support for Model-Based Interface Development*, in Proc. of *CADUI96: Computer-Aided Design of User Interfaces*. 1996. Namur, Belgium.
7. Puerta, A. R. A Model-Based Interface Development Environment. *IEEE Software*, (14) 4, July/August 1997, pp. 40-47.
8. Schlungbaum, E. *Individual User Interfaces and Model-Based User Interface Software Tools*, in Proc. of *IUI97*. 1997: ACM Press.
9. Szekely, P., *Reflections on Beyond Interface Builders: Model-Based Interface Tools*, in *Readings in Intelligent User Interfaces*, M. Maybury and W. Wahlster, Editors. 1998, Morgan Kaufmann. p. 507.
10. Szekely, P., Luo, P., and Neches, R. *Beyond Interface Builders: Model-Based Interface Tools*, in Proc. of *InterCHI'93*. 1993: ACM Press.
11. Vanderdonckt, J. M. and Bodart, F. *Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection*, in Proc. of *InterCHI'93*. 1993: ACM Press.