The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development

Angel R. Puerta

Abstract

Model-based interface development works on the following central premise: given a declarative interface model that defines all the relevant characteristics of a user interface, then comprehensive, automated, user-interface development environments can be built around such model. Yet, the high potential of this technology has not been realised because all interface models built so far are partial representations of interfaces, cannot be readily modified by developers, are implicitly tied to their associated development environment, or, importantly, are not publicly available to the HCI community.

The Mecano Project is a research effort that aims to overcome such limitations. It encompasses two phases: (1) The development of a comprehensive interface model available as a resource to the HCI community, and (2) the implementation of a open model-based development environment based on such an interface model. In this paper, we report on the first phase of the project. We present the Mecano Interface Model (MIM), and its associated interface modelling language (MIMIC).

We describe a metalevel paradigm for interface modelling that overcomes problems of flexibility and completeness. The paradigm is also unique in that it not only models the user interface but also models explicitly the design process of the interface. This allows the construction of software tools that operate on the design process as well as on the interface elements. MIM has been validated via a variety of paper-based interfaces.

Keywords

Model-based interface development, interface models, user interface design.

Introduction

The paradigm of model-based interface development has attracted a high degree of interest in the last few years due to its high potential for producing integrated user interface development environments with support for all phases of interface design and implementation.

The basic premise of model-based technology is that interface development can be fully supported by a generic, declarative model of all characteristics of a user interface, such as its presentation, dialogue, and associated domain, user, and user task features. As depicted in fig. 1, with such model at hand, suites of tools that support editing and automated manipulation of the model can be built so that comprehensive support for design and implementation is possible. Typically, users of model-based environments (i.e., interface developers) refine the given generic model into an application-specific interface model using the tools available within the environment. A runtime system then executes the refined model as a running interface.

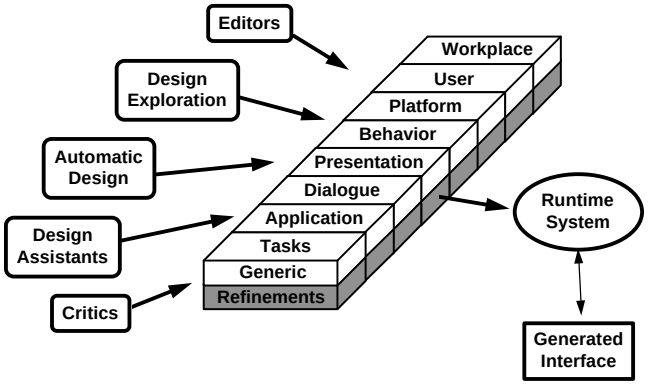

Figure 1. The model-based paradigm. Design tools operate on a generic interface model to produce an application-specific refined model that is then executed by a runtime system.

The benefits of model-based development are manifold. By centralising interface information, model-based systems offer support, within a single environment, for high-level design as well as for low-level implementation details. Global changes, design visualisation, prototyping, consistency of resulting interfaces, and software engineering principles in general are much improved over currently available tools, such as interface builders, which offer only partial and localised development support. Over the past few years, several model-based systems [Foley91, Johnson95, Puerta94b, Szekely93, Vanderdonckt93] have demonstrated the feasibility of the model-based approach.

Despite all the potential shown, model-based technology is struggling to find its way out of the laboratories. This is due mainly to the absence of one of the key elements needed by the technology to truly prosper. The two central ingredients for success in model-based systems are: (1) a declarative, complete, and versatile interface model that can express a wide variety of interface designs, and (2) a sufficiently ample supply of interface *primitives*, elements such as push-buttons, windows, or dialogue boxes that a model-based system can treat as black boxes. The need for the first ingredient is clear: without a vocabulary rich enough to express most interface designs, the technology is useless. The second ingredient is also critical because model-based approaches fail if developers are required to model too low-level details of interface elements—a problem painfully demonstrated by the erroneous modelling abstraction levels of some early model-based systems.

Whereas there is little question that good sets of interface primitives are available in most platforms, researchers have fallen short of producing effective interface models. The problems with current interface models can be summarised as follows:

- *Partial models*. Models constructed up-to-date deal only with a portion of the spectrum of interface characteristics. Thus, there are interface models that emphasize user tasks [Johnson95], target domains [Puerta94b], presentation guidelines [Vanderdonckt93], or application features [Szekely93]. These models generally fail when an interface design puts demands on the model beyond the respective emphasis areas.
- *Insufficient underlying model*. Several model-based systems use modelling paradigms proven successful in other application areas, but that come up short for interface development. The Entity-Relationship model, highly effective in data modelling, has been applied with limited success in interface modelling [Janssen93, Vanderdonckt93]. These underlying models typically result in partial interface models of restricted expressiveness.
- *System-dependent models*. Many generic interface models are non-declarative and are embedded implicitly into their associated model-based system, sometimes at the code level. These generic models are tied to the interface generation schema of their system, and are therefore unusable in any other environment.
- *Inflexible models*. Experience with model-based systems suggests that interface developers many times wish to change, modify, or expand the interface model associated with a particular model-based environment. However, model-based systems do not offer facilities for such modifications, nor the interface models in question are defined in a way that modifications can be easily accomplished.
- *Private models*. Interested developers or researchers wishing to obtain a generic interface model from one of the currently available model-based systems, quickly find that there is no executable version of an interface model that is publicly available,

or even obtainable via a licensing agreement. The inability to produce an interface model fit for distribution to third parties is one of the major shortcomings of model-based technology.

1. The Mecano Project

To address the limitations described above, we started at the beginning of 1995 *The Mecano Project*. This project draws from our own experience building Mecano [Puerta94b]a model-based system where interface generation is driven by a model of an application domainand from our examination of several model-based systems built in the past few years. The project encompasses two phases:

- *Phase one: The interface model*. In this phase, we define a generic interface model with a high degree of completeness, portability, and independence from a corresponding model-based system. The interface model is to be available as a resource to the HCI community.
- *Phase two: The model-based environment*. In this phase, we implement a model-based environment that supports interface generation based on the phase-one interface model. The system is to embody an open architecture so that third-party developers can contribute their own tools to the environment simply by adhering to the vocabulary and definitions of the phase-one interface model.

In this paper, we present the results of phase one of The Mecano Project. We first introduce the interface modelling language MIMIC and explain a metalevel approach to writing interface models that overcomes problems of completeness and flexibility in interface models. Subsequent sections describe in detail the grammar and features of MIMIC. Through an example, we show how the generic Mecano Interface Model (MIM) is written using MIMIC, and how a specific sample interface is defined with this language. We conclude by detailing our approach towards validation of MIMIC and MIM, by examining related and future work, and by presenting a set of conclusions.

2. A Metalevel Approach to Modelling

The requirements of completeness, flexibility, and system independence of an interface model are very difficult to achieve within a monolithic structure for interface modelling, as is the case with current model-based systems. Even the most elaborate interface model will run into difficulties if changes or extensions are needed. Furthermore, the idea that a single generic interface model that can express most interfaces can be defined is debatable at best, and certainly contrary to experience gathered with the use of model-based systems.

The key reasons why interface models lack flexibility are first that they were not designed expressly with the intention of being changed once implemented, and second, but perhaps more importantly, that they lack an explicit description of the organisation and structure of the model components. Without such description, it is difficult to understand the role played in an interface design by the different interface elements being modelled, and it is also hard to visualise the relationships among those elements. As a consequence, tools cannot be built to support the model expansion process, and manual changes are coding exercises usually only accessible to the original designers of the interface model.

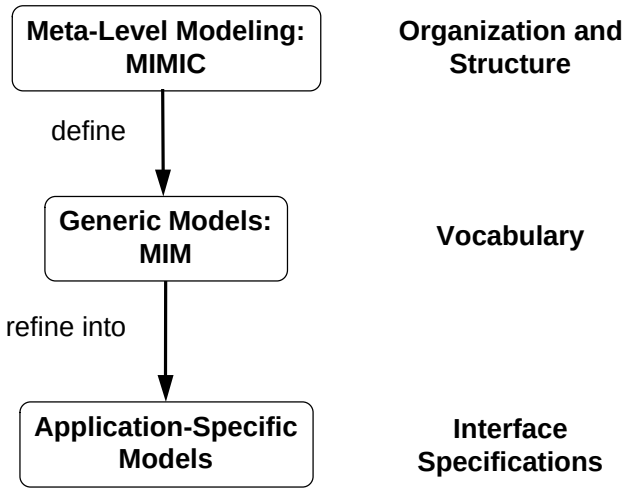

Figure 2. A multilevel approach to interface modelling. MIMIC defines roles, organisation, and structure of interface model components. MIM is a generic model whose structure follows the MIMIC definitions. Interfaces are refined from MIM into application-specific models.

In the Mecano Project, we overcome the various limitations of current interface models by means of a modelling approach at multiple levels of abstraction, as shown in fig. 2. The result is an interface modelling language, called MIMIC, that can be used to express both generic and application-specific interface models. We also provide one generic model called the Mecano Interface Model (MIM). The MIMIC language follows the following principles:

- *Explicit representation of organisation and structure of interface models*. MIMIC provides a metalevel for modelling that assigns specific roles to each interface element, and that provides the constructs to relate interface elements among themselves. There is no fixed way to relate elements, so developers are free to build their own schema (e.g., building a Petri Net of dialogue elements).

- *No single generic model*. We have discarded the idea that a single, all-encompassing generic interface model can be built successfully as previously assumed. Instead, MIMIC supports the definition of generic interface *models*. We provide one such generic model in MIM and our model-based system will support that generic model. However, we envision that developers, and the HCI community in general, will produce a number of such generic models, or extensions of generic models, that are suited for specific user tasks, application domains, or given platforms.
- *Explicit interface design representation*. Interface models written with MIMIC will define not only interface elements, but also characteristics of the design process for the modelled interface. This is a feature lacking in all previous schema for interface modelling, but it is a crucial one if we are to give developers access to and control of the automated processes of interface generation in model-based systems.

3. The Mimic Modelling Language

MIMIC is an object-oriented modelling language that follows the general principles of C++, and is in fact implemented in C++. Thus, the MIMIC grammar is said to "bottom-out" on the C++ grammar. Inheritance and typing are similar between both languages.

3.1 The Sample Interface

Within the confines of this paper, it would be difficult to discuss an example of a complete interface built with MIMIC and MIM. Therefore, we have opted for presenting a simple, but artificial, domain that can be used to highlight features of MIMIC and to illustrate the building of interface models with MIM. Our validation of MIMIC, described in a later section, examined more realistic application domains. The sample interface is shown in fig. 3. The interface controls the firing of a cannon in a ship. The user must load and aim the cannon using the controls provided. The interface must enforce the restriction that firing cannot take place until the cannon has been properly loaded and aimed. The two numeric fields in the interface are used to specify in degrees the rotation at the base of the cannon (max. 360 degrees), and the firing angle (max. 85 degrees).

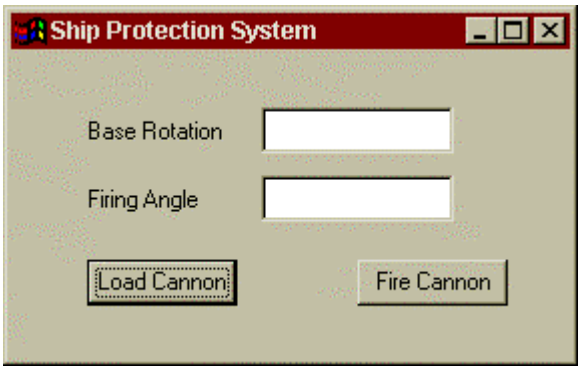

Figure 3. The ship protection system. Operators can fire a cannon only after it has been properly loaded and aimed.

3.2 Keys for Reading the MIMIC Grammar

In reading through the example shown in the following sections, a number of conventions must be observed. The BNF grammar is *abbreviated* to save space and improve readability. In particular, keywords and separators are not detailed, nor are some of the less interesting categories. The use of some of these should be obvious from reading the actual interface model. In addition, the following keys should be noted:

\+ means one or more instances of a category
* means zero or more instances of a category
** means zero or more *unique* instances of a category

Finally, in our example application-specific interface model, items marked **bold** highlight generic MIM-defined elements that are being referenced by the application-specific model.

3.3 Top Level Categories

```
<interface>              ::=  <interface-definition>*
                              <model-component>+
<interface-definition>   ::=  <interface-attribute>   |
                              <interface-relation>
<interface-attribute>    ::=  <attribute>
<interface-relation>     ::=  <relation>
<model-component>        ::=  <user-task-model>       |
                              <domain-model>          |
                              <presentation-model>    |
                              <dialog-model>          |
                              <user-model>            |
```

<design-model>

An interface is made up of one or more model components. There is no requirement for an interface to have all types of model components neither there is a limitation on the maximum number of components of each type that an interface can have. If an interface has not defined all types of components then it may or may not be *operational*. An operational interface is one that can be implemented as a running program by a runtime system. If an interface has more than one definition for a type of model component, then it may be operational at any one time with just one of the defined instances of the particular component type. Allowing multiple model components with the same role is useful when examining *what-if* scenarios and when dealing with portability. Interface definitions specify attributes and relations, which we will examine later, that apply to the interface as a whole. For our example, here is the top-level section of the model:

```
INTERFACE ship-protection {
  INTERFACE-DEFINITION is-a mecano-interface-model
```

The interface model is defined as a subclass of the Mecano Interface Model, MIM, thus inheriting all the attributes and relations defined for that particular generic model. MIM includes elements that support the 2-D, form- and dialogue-based interaction that our sample interface requires. Throughout the example, applied MIM elements are highlighted with bold font.

3.4 Global Categories

There are a number of global categories defined by MIMIC. Many, such as *name* and *value*, should be intuitive to the reader and will not be described. The key global categories that deserve the most attention are *relations*, *attributes*, and *conditions*. We shall see examples of the use of these categories when we examine the model components in our example. In this section, we only present the definition of those categories.

```
<relation>              ::= <relation-definition>   |
                            <relation-statement>
<relation-definition>   ::= <name>                  |
                            <allowed-class>
<relation-statement>    ::= <name>
                            <object>
```

A relation is the main mechanism to establish links among the objects defined in an interface model. A relation definition establishes the *nature* of a relation between objects (e.g., an *is-a* relation). The defined relation is one-to-many and specifies the classes that can be the target of the relation. Relations are typically defined at the top level of generic interface models, or at the top level of the components of generic

models. The scope of a relation is limited to the class where it is defined and to any children of that class. In contrast to a definition, a relation statement *applies* a defined relation to existing objects.

```
<attribute>              ::= <attribute-definition>  |
                             <attribute-value>
<attribute-definition>   ::= <name>
                             <value-definition>
                             <attribute-feature>**
<value-definition>       ::= <value-type>
                             <canonical-form>
                             <allowed-values>*
                             <default-value>
<attribute-feature>      ::= <feature-definition>    |
                             <feature-value>
<feature-definition>     ::= <name>
                             <value-definition>
```

An attribute is a characteristic, or property associated with a class or object in an interface model. An attribute definition establishes the type and features of an attribute in an interface model. Attributes are typically defined at the top level of generic interface models, or at the top level of the components of generic models. The scope of an attribute is limited to the class where it is defined and to any children of that class. Attributes can have attribute-specific features that are similar in nature to regular attributes, but that do not allow the definition of additional features within the feature itself. An attribute value assigns the values of a defined attribute and the values of that attribute's features.

```
<condition>      ::= <precondition>       |
                     <postcondition>      |
                     <initial-condition>
```

A condition is a Boolean expression that has a temporal quality. Conditions are used to specify the applicability at any given time of an activity, such as a user task or a command, or to specify the state of such activity. A precondition must be satisfied before an activity can be undertaken. A postcondition is satisfied after an activity has been completed. An initial condition is satisfied as soon as an activity is started.

3.5 The User-Task Model Component

```
<user-task-model>        ::= <name>
                             <user-task-definition>*
                             <user-task>+
<user-task-definition>   ::= <task-attribute>  |
                             <task-relation>
<user-task>                ::= <name>
```

```
                                <task-relation>*
                                <goal>
                                <subtask>*
                                <execution-order>
                                <condition>*
                                <task-attribute>**
<subtask>              ::=  <user-task>
```

A user-task model is a collection of hierarchically-ordered user tasks. A user task is a definition of an activity that a user desires to perform. A task has a final purpose, or goal (a Boolean expression), and may be decomposable into several subtasks. The subtasks are performed according to an execution order under given conditions. Note that the semantics of the hierarchy built with this model component are left to the interface developer. Thus, the hierarchy of user tasks may constitute a GOMS model, or it may constitute some type of activity graph. The user task model for our example is as follows:

```
USER-TASK-MODEL protection-tasks{
    USER-TASK-DEFINITION
      is-a mecano-user-task-model
    USER-TASK ProtectShip {
      GOAL (fire-cannon TRUE)
      SUBTASK (load-cannon aim-cannon fire-cannon)
      EXECUTION-ORDER sequence}
    USER-TASK load-cannon {
      GOAL (load-cannon TRUE)}
    USER-TASK aim-cannon {
      GOAL (aim-cannon TRUE)}
    USER-TASK fire-cannon {
      GOAL (fire-cannon TRUE)
      PRECONDITION
        (load-cannon == TRUE && aim-cannon == TRUE)
      POSTCONDITION (load-cannon == FALSE) }}
```

The task of firing the cannon is decomposed into three subtasks that should be executed in sequence. Note, however, that the developer has chosen not to enforce the sequence in full by not specifying any conditions for the subtask aim-cannon. Thus, the model actually allows users to aim first and then load the cannon.

3.6 The Domain Model Component

```
<domain-model>       ::=  <name>
                          <domain-definition>*
                          <domain-object>+
<domain-definition>  ::=  <domain-attribute>   |
                          <domain-relation>
```

```
<domain-object>      ::= <name>
                         <domain-relation>*
                         <domain-attribute>**
```

A domain model is a collection of hierarchically-ordered domain objects that define all the objects in a domain along with their relationships. A domain object represents any entity in a given domain. Domain objects are characterised through domain-specific relations and attributes. Here is the domain model for our example:

```
DOMAIN-MODEL ship-cannon-system {
    DOMAIN-DEFINITION is-a mecano-domain-model
    DOMAIN-OBJECT cannon {
      DOMAIN-ATTRIBUTE load-state {
        TYPE BOOLEAN
        ALLOWED-VALUES (loaded empty)
        DEFAULT-VALUE empty}}
    DOMAIN-OBJECT aim-coordinates {
      DOMAIN-ATTRIBUTE base-rotation {
        TYPE FLOAT
        ATTRIBUTE-FEATURE range (0 360)}
      DOMAIN-ATTRIBUTE firing-angle {
        TYPE FLOAT
        ATTRIBUTE-FEATURE range (0 85)}}}
```

The domain model defines the relevant objects of the domain. The range attribute is defined in the Mecano Interface Model from which this model inherits attributes.

3.7 The Presentation Model Component

```
<presentation-model>       ::= <name>
                               <presentation-definition>*
                               <presentation-element>+
<presentation-definition>  ::= <presentation-attribute>  |
                               <presentation-relation>
<presentation-element>     ::= <name>
                               <presentation-relation>*
                               <presentation-attribute>**
```

A presentation model is a collection of hierarchically-ordered presentation elements. A presentation element represents any entity associated with an interface presentation, such as windows, displays, buttons, and other widgets. Presentation elements can be either abstract or concrete as defined by the interface designer. Abstract presentation elements are useful when dealing with portability issues. Presentation elements are characterised through presentation-specific relations and attributes. The presentation model also defines the characteristics of a presentation, such as layout and general guideline styles. A partial view of the presentation model for our example follows:

```
PRESENTATION-MODEL cannon-presentation {
    PRESENTATION-DEFINITION
      is-a mecano-presentation-model
    PRESENTATION-DEFINITION follows-style normal
    PRESENTATION-DEFINITION
      uses-medium (windows95 vdt)
    PRESENTATION-DEFINITION uses-mode graphical
    PRESENTATION-ELEMENT application-window {
      PRESENTATION-RELATION is-a window
      PRESENTATION-ATTRIBUTE type dialog
      PRESENTATION-ATTRIBUTE
        title "Ship Protection System"
      PRESENTATION-ATTRIBUTE font MS-Sans-Serif-8
      PRESENTATION-ATTRIBUTE border 2
      PRESENTATION-ATTRIBUTE dimensions (200 150)
      PRESENTATION-ATTRIBUTE is-resizable NO}
    PRESENTATION-ELEMENT fire-button {
      PRESENTATION-RELATION is-a push-button
      PRESENTATION-RELATION
        align-horizontal load-button
      PRESENTATION-RELATION
        belongs-to application-window
      PRESENTATION-ATTRIBUTE font MS-Sans-Serif-8
      PRESENTATION-ATTRIBUTE label "Fire Cannon"
      PRESENTATION-ATTRIBUTE dimensions (40 20)}}
```

The sample presentation model defines a GUI in Windows95. Note the heavy use of MIM elements in the presentation model (noted in **bold**), a level of use that should be typical in most interfaces. Each element of the interface is defined via attributes and relations. Note that some elements have an absolute window position while others are positioned via alignment relations. This keeps in line with the general philosophy of model-based systems where designers work at higher levels of abstraction by means of primitives. In this case, developers avoid working at the layout level of grids and guidelines.

3.8 The Dialogue Model Component

```
<dialog-model>        ::= <name>
                          <dialog-definition>*
                          <command>+
<dialog-definition>   ::= <dialog-attribute>   |
                          <dialog-relation>
<command>             ::= <name>
                          <dialog-relation>*
                          <goal>
```

```
                                <subcommand>*
                                <execution-order>
                                <interaction-technique>+
                                <response>*
                                <condition>*
                                <dialog-attribute>**
<sub-command>           ::=  <command>
<interaction-technique> ::=  <relation>
<response>              ::=  <initial-response>    |
                             <final-response>
<initial-response>      ::=  <relation>
<final-response>        ::=  <relation>
```

A dialogue model is a collection of hierarchically-ordered user-initiated commands that define the procedural characteristics of the human-computer dialogue in an interface model. A command is a definition of a user-initiated activity that a user desires to perform. A command has a final purpose, or goal (a Boolean expression of arbitrary complexity), and may be decomposable into several subcommands. The subcommands are performed according to an execution order under given conditions.

Commands are executed via interaction techniques and may produce one or more system responses. An interaction technique is a special class of relation that links an existing command with a specific technique for interaction that is carried out via one or more presentation elements. Thus, performing an interaction technique, such as a mouse click, on a presentation element, such as a push button, is equivalent to executing the command within which such interaction technique is specified. A response is another special class of relation that defines a system reaction to a user action. Responses have a temporal element that determines at what point during the execution of a command the responses take place. An initial response occurs immediately after its corresponding command is initiated. A final response occurs immediately after its corresponding command is completed satisfactorily (i.e., the command goal has been achieved).

As with user tasks, the semantics of the hierarchy of commands are left to the designer who can work with a number of dialogue description schema by using the dialogue model component. The following is the dialogue model for our example.

```
DIALOG-MODEL ship-protection-dialog {
  DIALOG-DEFINITION is-a mecano-dialog-model
  COMMAND launch-application {
    GOAL (fire-cannon TRUE)
    SUBCOMMAND (load-canon aim-cannon fire-cannon)
    EXECUTION-ORDER sequence
    INITIAL-RESPONSE disable fire-button
    FINAL-RESPONSE disable fire-button
```

```
    INITIAL-CONDITION (load-cannon == FALSE)
    INITIAL-CONDITION (aim-cannon == FALSE)
    INITIAL-CONDITION (fire-cannon == FALSE)
    POSTCONDITION (load-cannon == FALSE)
    POSTCONDITION (aim-cannon == FALSE)}
  COMMAND load-cannon {
    GOAL (load-cannon TRUE)
    INTERACTION-TECHNIQUE
      left-mouse-click load-button}
  COMMAND aim-cannon {
    GOAL (aim-cannon TRUE)
    INTERACTION-TECHNIQUE edit-float
      (base-rotation-editbox firing-angle-editbox)
    FINAL-RESPONSE enable fire-button}
  COMMAND fire-cannon {
    GOAL (fire-cannon TRUE)
    INTERACTION-TECHNIQUE
      left-mouse-click fire-button
    FINAL-RESPONSE disable fire-button
    PRECONDITION
      (load-cannon == TRUE && aim-cannon == TRUE)
    POSTCONDITION (load-cannon == FALSE)
    POSTCONDITION (aim-cannon == FALSE)}}}
```

The dialogue model follows closely the user task model. Note that whereas the requirement that the cannon not be fired until is loaded and aimed is enforced using enabling and disabling of buttons, the sequence of "load-cannon" before "aim-cannon" is not actually enforced by any system response or interaction technique. The parallelism between user-task models and dialogue models in MIMIC is not coincidental. We consider the user-task model the driving paradigm for the interaction dialogue and expect that automated tools in our model-based system will exploit such parallelism.

3.9 The Design Model Component

Although the model components shown so far in our example seem to capture a full description of the interface, there is in fact a wealth of information that remains implicit in those components and that is crucial if we desire to automate the refinement of interface models.

For example, why were push buttons used to operate the cannon? What is the connection between user tasks, domain objects, and presentation elements? These and many other similar questions are integral part of an interface design, yet interface models have failed to capture it. The design model component in MIMIC is used for exactly that purpose.

```
<design-model>       ::= <name>
                         <design-definition>*
                         <design-mapping>+
<design-definition>  ::= <dialog-attribute>   |
                         <dialog-relation>
<design-mapping>     ::= <relation>
                         <mapping-condition>*
```

A design model is an unordered collection of design mappings. The mappings establish design relationships among interface objects. The applicability of a mapping may be subject to a number of mapping conditions (Boolean expressions). Here is a partial view of the design model for our example:

```
DESIGN-MODEL ship-protection-design {
  DESIGN-DEFINITION is-a mecano-design-model
  DESIGN-MAPPING presentation-assignment
    (FLOAT editbox)
  DESIGN-MAPPING presentation-assignment   (BOOLEAN push-button)
  DESIGN-MAPPING task-domain-link
    (load-cannon cannon.load-state)
  DESIGN-MAPPING task-domain-link
    (fire-cannon cannon.load-state)}
```

This view shows that two different user tasks need access to the same attribute in the domain object. As a consequence, two interface elements are made available to the user to perform those tasks. The interface elements are push buttons as determined by the presentation assignment of the type FLOAT of the load state of a cannon.

When modifying designs, developers often change not interface elements per se but rather the rationale for the existence of those elements. Thus, if a developer does not wish to use push buttons in the ship protection interface, it may be more appropriate to operate on the design model to change the presentation assignments than on the presentation model itself.

3.10 The User Model Component

```
<user-model>         ::= <name>
                         <user-definition>*
                         <user>+
<user-definition>    ::= <user-attribute>   |
                         <user-relation>
<user>               ::= <name>
                         <user-attribute>**
                         <user-relation>*
```

A user model is a collection of hierarchically-ordered users. A user is a description of the characteristics of an individual user or of those of a stereotype of a user group. The user model is not intended to be a description of the mental state of a user. Our example does not have a user model component.

4. Model Validation

To validate the MIMIC modelling approach, and to refine MIM, we conducted a process of writing a variety of application-specific interface models. We aimed more at breadth than at volume of interfaces examined. Both members of our group and outside contributors were given the MIMIC language specifications along with a current version of MIM, and were asked to write an application-specific model of their choice based on an existing interface. Examples worked out ranged widely in size from toy domains as the one shown here, to subsets of commercial applications such as Microsoft Word and Netscape Navigator.

While some developers had trouble initially with the semantics of MIMIC, most changes in the long run were accommodated by modifying, or extending, MIM. Typically, developers would find the need to define relations or attributes at the application-specific level that we would later on incorporate into MIM. Yet, in other instances, developers would suggest defining MIM relations in a different way from what was being provided. This experience solidifies our belief that multiple generic interface models (i.e., multiple MIM) will be necessary eventually. During the next phase of The Mecano Project, we expect to continue the validation process, this time from a software-supported, as opposed to manual, point of view.

5. Implementation Issues: Automating Model Building

In the second phase of The Mecano Project, we implement a model-based environment, called Model-Based Interface Designer (Mobi-D), that supports interface generation based on the phase-one interface model. The main components of Mobi-D can be seen in fig. 4. The system has three main features:

- *User-centred interface development in an integrated and comprehensive environment*. Developers build interfaces manipulating abstract objects such as user tasks and domain objects. The production of presentation styles and dialogues is automated in most part by the environment.
- *Transparent modelling language*. Developers do not need to know the MIMIC modelling language, just the roles of the different components of an interface model. The environment tools provide the interactive functionality needed to complete model editing operations without having to read or write in the MIMIC language.

- *Open architecture*. Third-party developers can enhance the environment by incorporating their own design tools. Such tools need only to adhere to the MIMIC language. This feature is key in supporting machine-learning and other techniques for user-task automation.

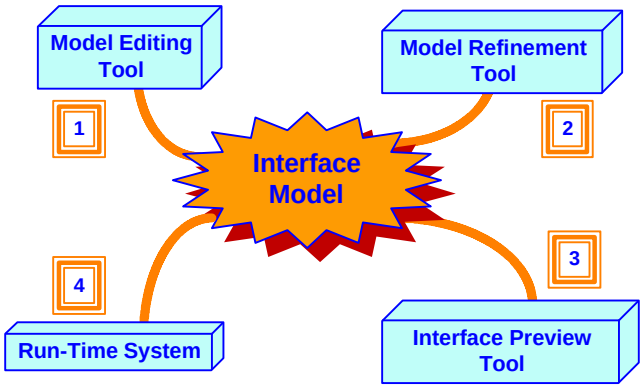

Figure 4. The Mobi-D development environment.

6. Related Work

There are a number of model-based systems that have been developed over the past few years. In general, they all suffer from the limitations outlined in the introduction to this paper. We will highlight here their contributions more than their shortcomings. In general, interface models mentioned here are subsets of MIMIC, do not support explicit design layers, and do not separate levels of abstraction as MIMIC does.

ADEPT [Johnson95] drives interface generation entirely from models of the user tasks. It applies a multistep refinement process that methodically links tasks to abstract interface elements, then to concrete interface elements that can be assembled into a running interface. Interfaces generated by ADEPT have a high degree of portability thanks to the use of abstract interface elements. Another successful task-based system is TRIDENT [Vanderdonckt93] which has an excellent knowledge base of design guidelines that are consistently applied during interface generation. Parts of the TRIDENT interface model are based on the ERA paradigm, on which the GENIUS system is based as well [Janssen93]. GENIUS, however, does not define an interface model but rather models certain dialogue and data elements for interface generation purposes.

UIDE [Foley91] provided one of the earliest attempts at building an interface model. The model was mainly a presentation component augmented by data and dialogue

constructs. The system demonstrated the high potential for the automation of interface generation from models.

A related system is HUMANOID [Szekely93] which builds interfaces around application models and makes use of pre-defined presentation templates to solve layout generation problems. Both of these systems are now being combined into a new generation system called MASTERMIND [Neches93]. MASTERMIND shares some of the goals of The Mecano Project and will certainly overcome many of the problems of its predecessors. We believe, however, that its associated interface model and modelling language [Szekely95], built as a single, all-encompassing structure, will suffer from similar limitations to those outlined at the beginning of this paper.

ITS [Wiecha90] has been used successfully at the commercial level. Its approach is particular in the sense that it supports team development, and that it takes an organised view at the use of rules for interface generation.

Conclusion

We have presented a modelling approach for user interfaces that overcomes many of the limitations of previous approaches in model-based systems. We implement a metalevel paradigm for interface model building with a top level that defines the organisation and structure of interface models, a generic level that defines the vocabulary for model building via generic interface models, and an application-specific layer where interfaces are modelled.

We introduced the MIMIC modelling language for interfaces, and the generic Mecano Interface Model, MIM. MIMIC includes as one of its interface roles, a design model component that explicitly states the relationships among the different elements of an interface.

We have validated our modelling approach by writing a variety of interfaces in MIMIC with the support of MIM. The modelling language is to be supported transparently by a model-based development environment, called Mobi-D, featuring an open architecture.

Acknowledgements

Special thanks to David Maulsby who provided extensive commentary on this paper. Our thanks to all the reviewers for their thoughtful comments. This work was supported by the US Government under the Defense Advanced Research Program Agency (DARPA).