User Interface High-Order Architectural Models

L.M.F. Carneiro M.H. Coffin D.D. Cowan C.J.P. Lucena*

Computer Science Department, University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
lmfcarne@neumann.uwaterloo.ca

Abstract

Many user-interface design models (usually expressed through architectural designs), expressed at different levels of abstraction, have been proposed in the literature. These models have been classified according to several criteria. The various design models are explicitly or implicitly derivable from a high order model which constitutes its specification. Existing classification schemes [21, 9, 16] do not reflect this derivation process.

A new classification scheme for user-interface architectural designs is presented in this paper. This new classification scheme is based on derivations of designs. We analyze the initial specification and the high-order design models which correspond to models proposed in the literature at various levels of abstraction. The goal of the present study is to discuss issues such as relevant properties of "design families" (related to design trajectories), specification notations to be used as initial design steps, and "implementation biases" induced by different design families. In particular, we put the Abstract Data View model [6] in perspective by reviewing its specification, presenting its high-order architecture, and comparing it with architectures at similar and lower levels of abstraction.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General – *System architectures*; D.2.2 [**Software**]: Software Engineering – *Tools and Techniques*; D.2.10 [**Software**]: Software Engineering – *Design*; D.2.m [**Software**]: Software Engineering – *Miscellaneous*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – *Evaluation/Methodology*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – *UIMS*;

General Terms: Abstract Data Types, Design, Interactive Applications, User Interfaces

Additional Key Words and Phrases: End-User Programming

1 Introduction

The design process is an activity aimed at the creation of system descriptions at increasing levels of detail. The result of this process can be expressed as a sequence of models. The order within

*L.M.F Carneiro, M.H. Coffin, and D.D. Cowan are with the Computer Science Department, University of Waterloo; C.J.P. Lucena is on leave from the Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Brazil; L.M.F. Carneiro holds a doctoral fellowship from CAPES (the Brazilian Research Council).

this sequence does not necessarily reflect the order in which the descriptions were obtained. The transition between two consecutive models within this sequence reflects the activity which starting from some initial model (the input or specification for the designer), produces a resulting model (the design of the current step).

A specification denotes a formal statement of the requirements and properties of a system, whereas a design (or implementation) is a formal statement of a structure which meets these requirements and which is usually expressed in terms of more primitive elements. It follows that the terms *design* and *specification* are closely associated: a specification is the input for the designer, whereas the design is its output. A design may in turn be used as a specification (input) for the next design phase. These transitions are the basic constituents of the design process.

Cockton [4] describes the differences between a model and an architecture. According to him, "models are the minimal form for software structures", and "architectures detail component roles and the interface between the components". Therefore, "a model is a starting point, an architecture is its first refinement". In this paper, we will talk about models, which we call architectural models. Architectural models can be seen as the real first refinement of a model.

Many user-interface design models (usually stated through architectural designs), expressed at different levels of abstraction, have been proposed in the literature [10, 11, 5, 15, 2, 12, 23]. The various design models are explicitly or implicitly derivable from a high-order model which constitutes its specification. Sometimes it is possible to trace back a given design model to the original specification. When it is possible, the design model has a corresponding high order architectural model. Existing classification schemes [21, 9, 16] do not reflect this derivation process.

A new classification scheme for user-interfaces architectural designs is presented in this paper. This new classification scheme is based on the analysis of derivations of designs. We analyze the initial specification (which may be formal or informal), and the high-order design models which correspond to models proposed in the literature, at various levels of abstraction. The goal of the present study is to discuss issues such as relevant properties of "design families" (related to design trajectories), specification notations to be used as initial design steps, and "implementation biases" induced by different design families.

In particular, we put the Abstract Data View (ADV) model [6] in perspective by reviewing its specification, presenting its high-order architecture, and comparing it with architectures at similar and lower levels of abstraction.

Some problems of existing architectural models are pointed out in the literature [24, 19]: application independence, overloading, and repetition of semantic constraints. The application independence problem refers to the problem of how changes in the non-user-interface application will affect the user-interface application, and vice-versa. *The overloading problem refers to operations that has different meanings in different contexts, such as a mouse click operation that has different meanings depending on the position of the mouse.* The repetition of semantic constraints problem refers to different components of the architecture enforcing the same semantic constraints over their objects. We will show that the ADV architectural model addresses these problems.

In this paper, we will use the word *system* to refer to user-interface application and non-user-interface application. The *user-interface application* encompasses the part of the system that deals with the user-interface. The *non-user-interface application* encompasses the functional and structural part of the system.

This paper is divided into six sections: introduction, a classification scheme, the ADV architectural model, comparing the ADV architectural model, high-order architectural models, and conclusion.

In the classification scheme section, we will use the classification scheme suggested in [16], and classify six architectural models presented in the literature according to this scheme. These models are: Monolithic, Client-Server, Seeheim, MVC, ALV, and PAC architectural models.

In the ADV architectural model, we will propose an architectural model for the ADV approach based on [6].

In the comparing the ADV architectural model section, we will compare the ADV architectural model to the six models described in the first section.

In the high-order architectural models section, we will divide the scheme into two levels: the high-order architectural model and the implementation architectural models. We will show how the ADV architecture model can be mapped into the implementation architectural models.

2 A Classification Scheme

Larson, in his book on tools for building user interfaces [16], suggests that the design decisions to be made when creating a user interface can be divided into five classes:

1. Structural
2. Functional
3. Dialog
4. Presentation
5. Pragmatic

These decision classes, which are described in the next few paragraphs, each represent one aspect of the design. We use a chess game to describe a possible result from each decision class.

The *structural* class specifies the structure of the conceptual objects which will be manipulated by the user through the user interface and the relationships among these objects. For example, the conceptual objects in a chess game would be the pieces, the position of the pieces on the board, and the players. The structure of the pieces is a set of the six types of pieces allowed in a chess game: Pawn, Rook, Knight, Bishop, Queen and King. The structure of the players is also a set, composed of the two possible players: Black and White. The rules of the game describe the constraints that need to be satisfied by the conceptual objects in order to move a piece.

The *functional* class enumerates the functions that can be applied to the conceptual objects and the syntax and semantics of these functions. For each function the input, the output, and possible errors are described. In the chess game pieces can be moved under the constraints usually defined for pieces in a chess game. An attempt to place a piece in an "illegal" position results in an error indication. Therefore, the chess game will have basically one function: the move function, which has a piece and its position as input, and generates a new position for the piece, or an error (because of a invalid move) as output.

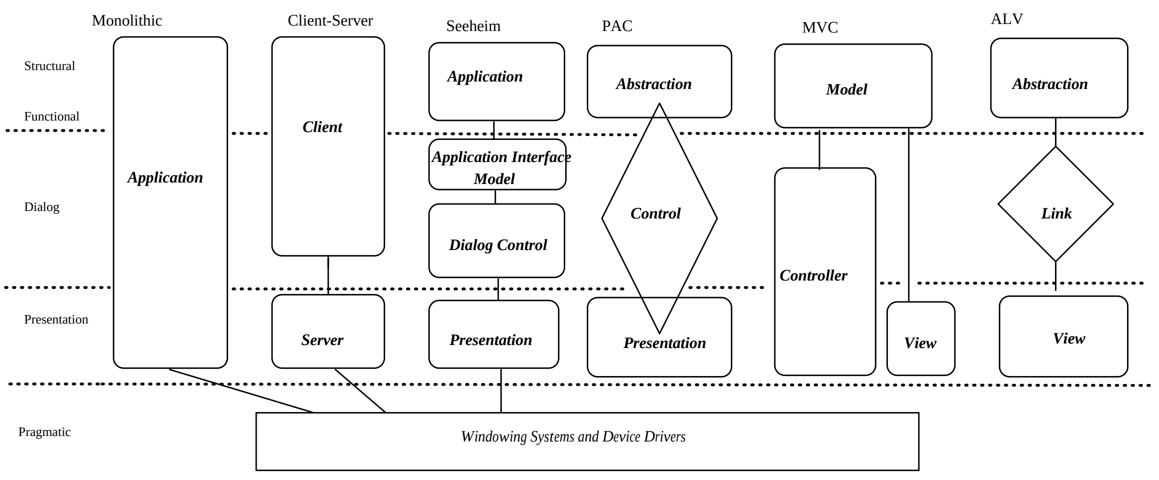

Figure 1: Architectural Models

The dialog class specifies the content and sequence of information exchanged between the user and the system. The information exchanged between the user and the system can be divided into two categories: the information exchanged between the user and the user-interface application, and the information exchanged between the user-interface application and the non-user-interface application. In the chess game there are five operations allowed for information exchange between the user and the user-interface application, and two operations allowed for information exchange between the user-interface application and the non-user-interface application. The input operations allowed for information exchange between the user and the user-interface application are: selecting the piece, moving the piece, releasing the piece. The output operations are: drawing the board, and drawing the piece. The operations allowed for information exchange between the user-interface application and non-user interface application are a request to move a piece and the result of the move.

The *presentation* class specifies the interactive objects that compose the user interface. The specification includes the algorithms that implement the visual aspects of the user-interface, and algorithms dealing with user-entry data and commands. The interactive objects can be input only, output only, or input-output objects. Thus, the presentation class can be divided into two subclasses: one dealing only with the input, and the other dealing only with the output. The input part will have the algorithms dealing with the user-entry data and commands; the output part will have the algorithms dealing with the visual aspects of the interactive objects. In the chess game, the interactive objects are: the board, an output object, and the pieces, which are input-output objects.

The *pragmatic* class contains all the decisions which are hardware dependent.

This taxonomy of design decisions provides a method for characterizing the different approaches to designing applications with user interfaces. Figure 1 shows six user-interface characterizations or architectural models [16]: monolithic, client-server, Seeheim, Presentation-Abstraction-Control (PAC), Model-View-Controller (MVC), and Abstraction-Link-View (ALV). The vertical axis of Figure 1 represents the five design decision classes.

The Monolithic architectural model does not segregate the design decisions. Everything from structural to presentation decision classes is implemented in a single component. Such an architectural model is not recommended for interactive systems—a system divided into user-interface application and non-user-interface application— or any other software system. Such a structure has many deficiencies which are documented in the literature [16].

The Client-Server [23] architectural model divides the single component of the monolithic architectural into two sub-components: the *Client* and the *Server*. There is a not a rigid division stating which design decision should be assigned to either the Client or the Server. This model is a communication model, rather than an implementation or logical model. It enforces how the Client and the Server should communicate with each other; it does not enforce any semantics for the Client and the Server components. Nevertheless, some User-Interface Management Systems (UIMSs) and user-interface toolkits [18] based on this architectural model enforce semantic constraints for the client and the server: the client implements the data structures and corresponding functions which support and manipulate the conceptual objects, and the server is responsible for presenting information from the client to the user, for converting data and commands entered by the user, and for sending the converted data to the client. In this division, the client also provides most of the support for the communication between the conceptual object and the user-interface application. Thus, the client will contain all decisions made in the structural, functional classes, and most of decisions in the dialog class; and the server will contain all decisions made in the presentation class, and a small segment of the dialog class, since it is responsible for the initial handling of an input command. This architectural model is the first attempt to divide the tasks in a user interface.

The Seeheim architectural model [11, 21, 10] divides the system into the *Application*, *Application Interface Model*, *Dialog Control*, and *Presentation* components, and assigns specific decision classes to each component. The Application contains both the structural objects and the functional core. The Application Interface Model contains part of the dialog-decision class. It is responsible for handling the communication between the user-interface application and the non-user-interface application. The Application Interface Model is viewed as the representation of the system from the viewpoint of the user-interface application, containing all the data structures and functions that belongs to the non-user-interface application that should be available for the user-interface application. It also contains a filter that is responsible for analyzing the data before it is sent to the non-user-interface application. The Presentation contains all the decisions made in the presentation class. The Dialog Control controls how much information the Presentation and the Application Interface Model need to know about each other. This control is done through accessing the states of the Presentation and the Application Interface Model. Thus, the Dialog Control contains part of the dialog decision class. This model has the advantage that it explicitly addresses the problem of dialogue independence [19]. Such a model also has some disadvantages [24, 16] primarily because of the introduction of layers into the model. Since there is no direct connection between the Application Interface Model and Presentation components, this model may enforce the same semantic constraint.

The MVC [15] architectural model is somewhat similar to the Seeheim model. The MVC model is based on the multi-agent paradigm [2]. This model divides the system into *Model*, *Application*, and *View* components. The Model and Application (from the Seeheim model) components are similar in that they both contain the structural objects and functional core. The View component supports output presentation only, and contains the output portion of the presentation class. The Controller component handles the user-entry of data and commands, and manages all the information exchanged between the user and the Model. Thus, the Controller spans both the dialog and presentation classes. In the MVC architectural model, the Model component communicates directly with both the display (View) and the input (Controller) components. According to [15], the typical interaction cycle involves the user taking some input action and the controller notifying the model to change itself. The model performs the necessary changes and broadcasts these modifications to its dependents. Views can interrogate the model to determine whether it is necessary to update the display.

The ALV architectural model also is somewhat similar to the Seeheim model. This model divides the system into *Abstraction*, *Link*, and *View* components. The Abstraction component of the ALV model is similar to the Application component of the Seeheim model in that they both contain the structural and functional objects. The View and Presentation are also similar in that they both contain the presentation objects. The Link component contains the dialogue objects. However, the dialogue objects in the ALV model have a different structure than the dialogue objects in the Seeheim model. Instead of using the callback approach [9], the ALV model uses the constraint approach [17, 22]. The Link component contains variables that are similar to active values, that is, variables that allow other objects to register functions with them. Thus, whenever the value of an active value changes, the function that is registered within it is called. Constraints are similar to active values in that both allow variables to register functions within them. However, it is the constraint-satisfaction system that will hold those active values, making active values more general and more decentralized.

The PAC architectural model [5, 2] is another example of a multi-agent based model. This architectural model structures an interactive system as a hierarchy of agents. The Abstraction contains both the structural objects and the functional core. The Presentation component is responsible for the perceivable behavior, that is, contains all the decisions made in the presentation class, and the communication between the user-interface application and the non-user-interface application, both input and output. The Control contains all the decisions made in the dialogue class, remembers a local state so the model supports multithread dialogue, and maintains relationships with other agents. The Control component has no correspondence with any component in the other architectural models. This model has the advantage of explicitly supporting multithread dialogue.

3 The ADV Architectural Model

The ADV model cleanly separates the application from the user interface. The ADV model is formalized using the VDM specification language [7, 1, 14].

A typical system that is based on the ADV architectural model consists of a collection of Abstract Data Types (ADTs) [13] that manage the data structures and the state of the application, a collection of ADVs that comprise the perceivable behavior—handling both views and events—, and a mapping between the ADVs and ADTs. The mapping associates the views of variables in the ADV with the state of the variables in the ADT. Thus, any change in a variable initiated from the ADV will also be modified in the ADT, and any change of a variable in the ADT will be reflected

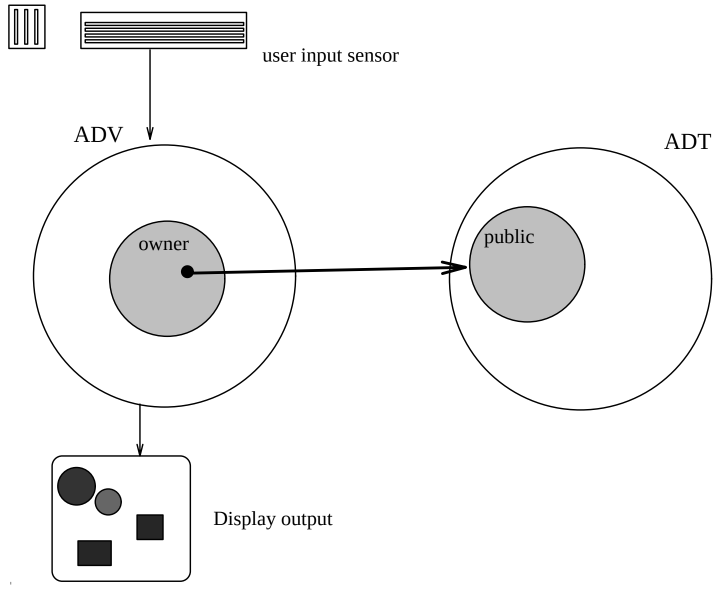

Figure 2: The ADV Architectural Model

in its view in the ADV. The changes in the variables in the ADV are usually a consequence of an event. The events are generated by input from devices operated by the user. The strategy used to ensure these modifications and changes is not enforced in the model. The diagram in Figure 2 illustrates the ADV, ADT, and the mapping that is shown as an arrow connecting the ADV to an ADT.

The ADT implements the design decisions made in the structural and functional decision classes. The ADT consists of a public interface and a body. In the body we describe the functions that are public, use public variables, as well as define variables and functions that are private to the ADT. The ADT is completely independent of the user interface, and in fact does not access any input or output. The ADV provides all the input and output functions and fully controls its associated ADT.

The ADV implements the decisions made in the presentation class and some of the decisions made in the dialog class. Therefore, the ADV implements all decisions concerned with the information exchanged between the user and the user-interface application, and all interactive objects.

The ADV model allows nesting of ADVs and ADTs. Both ADVs and ADTs can be viewed as objects. However, the model does not enforce any implementation-biased approach for the objects and the nesting mechanism. The nesting is simply the fact that an ADV object can include ADV objects. The same comment applies to ADTs; that is, an ADT object can include ADT objects.

In the ADV model, several ADVs can be associated with a single ADT; each can provide a different view of the ADT, or provide different control functionality. Since an ADT has no knowledge of input or output, it does not need to refer to any ADV. As a consequence, there is *not* a symmetrical arrangement in the ADT.

In the ADV architectural model the mapping, represented by the arrow in Figure 2, implements some of the decisions made in the dialog class. More specifically, the mapping between an ADV and an ADT is represented by the variable **owner**. The variable **owner** represents the information exchanged between the user-interface application and the non-user-interface application.

An ADV instantiation is associated with one and only one ADT instantiation. However, an ADT instantiation can be associated with several ADV instantiations. In other words, an ADV instantiation owns one and only one ADT instantiation; an ADT instantiation may be owned by several ADVs instantiations. This kind of relationship guarantees that a certain ADT can be viewed in different ways, but the different views are consistent with the single ADT.

Because the association relationship comes from the ADV, and not the other way around, the **owner** variable can be in the ADV for specification purposes. Roughly speaking, an ADV observes and manipulates the ADT which is its owner; from the point of view of the ADT, the observations are invisible and the manipulations are anonymous.

In systems based on the ADV architectural model, the control originates in the user-interface application rather than in the non-user-interface application, which avoids the callback spaghetti problem [20]. This placement of control is appropriate, since in highly interactive systems, the flow of control is determined primarily by the user of the system. The use of multiple ADVs, one for each independent view, makes the flow of control of each ADV quite simple: each ADV responds to a relatively small number of user-generated events and examines and manipulates one ADT.

4 Comparing the ADV Architectural Model to Non-Implementation-Biased Models

In Figure 3 we show the architectural models discussed before together with the ADV Architectural Model. The first difference between the ADV architectural model and the others is that the layers interact in a more structured fashion. In the ADV architectural model, communication between layers takes place when the lower layer (the ADV) executes synchronous invocations (procedure calls, effectively) to the upper layer (the ADT). An ADT never generates events that must be handled asynchronously by the associated ADV. In contrast, in the other models, a layer must not only handle synchronous invocations from the layer below, but also handle asynchronous invocations from the layer above. Handling asynchronous invocations is considerably more complicated than handling synchronous invocations since it requires error prone mechanisms such as signals, interrupts, or callbacks. Also, with an explicit mapping we enforce a one way communication, and, as a consequence, we have fewer interconnections, and the role and scope of the interface are defined unambiguously.

The need for asynchrony in the other models is due to the fact that the "official" locus of control is in the top-most layer: the non-user-interface application. This is consistent with "slightly-interactive" programs, which mostly compute but occasionally prompt the user for input. In highly interactive applications, however, the actual locus of control is in the bottom-most layer: the user. The tension between these two loci of control is the source of the complexity. The ADV architectural model avoids this tension by placing the main locus of control in the bottom-most layer. Fundamentally, the ADV model is based on the program waiting for the user rather than the user waiting for the program.

The difference between the ADV model and the Monolithic model is quite clear. In the monolithic architectural model there is no separation of the application and the interface at all—

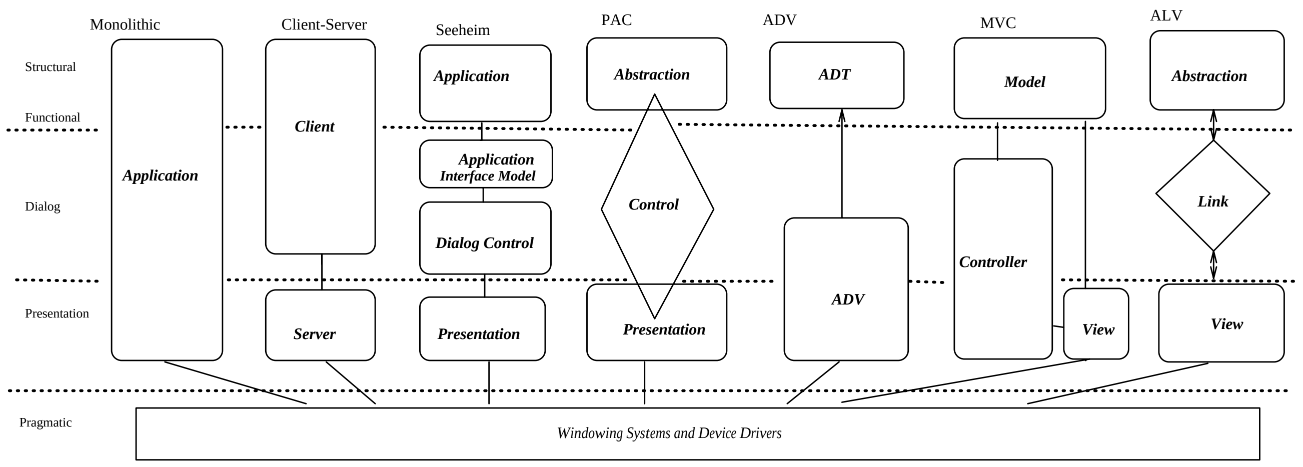

Figure 3: The ADV Architectural Model

conceptually or physically.

In some models that enforce semantic constraints to the Client-Server architectural model, all the decisions in the dialog class are made in the application. This forces the treatment of the communication between the user and the user-interface application to be done in the Client component. The Client comprises the decisions made in three levels: structural, functional and dialog levels. In other words, the Client—say non-user-interface application—has to deal with the treatment of all input data and requests made by the user, including issues that are of no interest to the non-user-interface application, such as ensuring that each character being input is not numeric, or ensuring two button presses are mutually exclusive. Handling input is the main reason for callback routines. An example of callback routines associated with widgets can be found in [9]. In our model, the dialog decision class is not at the non-user-interface application level, so all user-interface aspects are handled in the ADV.

In the Seeheim model, there is an interface between all decision classes levels. There is a lot of traffic between the components: each decision part must communicate, using messages, to the other decision part through these interfaces. In our model, this does not happen, because there is not any interface between any of the levels, making the specification of the communication between the components more explicit.

In the PAC model, the Control component resembles the Dialog Control component of the Seeheim model, containing all the decisions made in the dialog class. In the ADV model we saw that the decisions made in the dialog class are partly done in the Presentation component, and partly done by the mapping component (the **owner** variable). However, the multithread approach that is present in the Control component of the PAC model has no correspondence in the ADV model.

It seems, at first sight, that the ADV model is composed of two layers. We want to emphasize that this model is not layer-based. Also, we emphasize the separation of the user-interface application from the non-user-interface application. However, the word separation needs to be defined. We understand by separation as the conceptual separation of the user-interface application from the non-user-interface application, not necessarily the physical separation. The user interface is a frame around an application (non-user-interface application); the user interface cannot, and should not, ignore the non-user-interface application. On the other hand, the application should ignore completely the existence of this frame—or shell. Even more, the application should not be affected by the existence of a different shell around it. As the user interface has a tight connection with the application— as it should always have—, the user interface must know about the state of an application, not the other way around. For that purpose, the user interface should know which application is associated with it; this is done through the mapping.

5 High-Order Architectural Models

In the previous sections, we analyzed many user-interface design models, including the ADV architectural model. These models have been expressed at different levels of abstraction, and classified according to several different criteria. We classified the models according to the criteria presented in [16].

The various design models are explicitly or implicitly derivable from a high-order model which constitutes its specification. It is possible to trace a given design model to the original specification, which is at the root of the design process. We will classify the separation of the models as high-order architectural models and implementation architectural models. This new classification scheme is based on derivations of designs.

We can divide these architectural models into two groups: high-order architectural models, and implementation-architectural models. In the implementation-architectural models, implementation issues are enforced, and, consequently, if we remove the implementation details from the model, its components loose most of their semantics. In other words, the semantics of the components depends on the implementation details. For instance, if we remove the callback routines from the MVC model, most of the semantics of the Model, Controller and View components is lost. Generalization of implementation-architectural models is difficult to achieve, as they have their semantics tightly connected to implementation details. On the other hand, in the high-order architectural models, the semantics of the components are not enforced by implementation issues. Some of these models are formalized using specification languages, such as a state-machine description [8], or a VDM-like description [6].

Among the architectural models presented in this paper, we consider four as high-order architectural models:

- Client-Server
- Seeheim
- PAC
- ADV

In Figure 4 we present the architectural models divided into high-order architectural models and architectural models.

In this section, we will analyze how the ADV architectural model can be translated to other architectural models, which are considered to be implementation architectural models, that is, architectures that are implementation-biased. The same translation analysis could be done between other high-order architectural model and implementation models.

The ADV component of the ADV architectural model can be divided into three sub-components: description of the events—excluding the display event—, the display event, and the owner. The translation from the ADV architectural model to implementation architectural models is not always trivial, in the sense that the ADV and the ADT components may not have a direct mapping to the other architectural components as a whole.

The translation from the ADV architectural model to the X Toolkit implementation model [18] is shown in Figure 5. The X toolkit implementation model is a Client-Server based model. This means that the Client-Server Architectural model is trivially specialized to the X Toolkit implementation model: the client component of the Client-Server architectural model corresponds to the non-user-interface application, and the server component of the Client-Server corresponds to the widgets. The non-user-interface component of the X Toolkit registers callback procedures with the widget component. The widget component acts as a recipient of the event. That is, when an event occurs, the widget component registers what has happened, and calls the appropriate procedure in the non-user-interface application to perform the "what to do" part. In other words, when an event occurs, the system needs to know "what happened" and "what to do". Thus, in the X Toolkit implementation model the widget component contains the "what happened" procedures, and the non-user-interface component contains the "what to do" procedures. The non-user-interface component of the X Toolkit model also contains the functional core. The event sub-component of the ADV architectural model contains both procedures, that is, the events are completely treated in one place: there are descriptions of "what happened" and "what to do" procedures. Thus, the events and the display sub-components of the ADV model are mapped to both non-user-interface application and widgets components. The **owner** variable is implemented by the callback mechanism, and the ADT component is fully mapped to the non-user-interface application.

The translation from the ADV architectural model to the MVC implementation model is shown in Figure 6. As we saw, in the MVC model the presentation class is divided into two subclasses: one dealing with the output and the other dealing with the input. The output part deals with all the display aspects, and is handled in the View. The input part deals with all the user-data entry and commands, and is handled in the Controller. The Controller also deals with all decisions made in the dialog class. In the ADV model, both input and output decisions made in the presentation class are done in the ADV. Some of the decisions in the dialog class are done in the ADV, and others are delegated to the owner sub-component via the mapping. Therefore, the display sub-component of the ADV model is directly mapped to the View component of the MVC model, and the event sub-component of the ADV model is mapped to the Model and Controller components of the MVC model, for the same reason we translated the event sub-component of the ADV model to the non-user-interface and widgets components of the X Toolkit model. The mapping of the owner sub-component and the ADT component of the ADT model is trickier. We have to create the mapping in the model component of the MVC model in order to implement the ADT. The ADT can be just a subclass of the Model component. The mapping can be simulated by a method—which we have labeled the "to ADT" method. Thus, both this method and the callback routines together simulate the **owner** variable of the ADV model; the subclass ADT in the Model component of the

MVC model simulates the ADT component of the ADV model. In the ADV model, we can have many different views associated with an ADT. In the MVC model [15], we can also have many different views associated with a Model. Consequently, the translation of this feature is trivial.

The translation from the ADV architectural model to the ALV implementation model is shown in Figure 7. This mapping is simpler than the ones described before. The event and display sub-components of the ADV model are directly mapped to the View component of the ALV model, because the View component of the ALV model is responsible for treating all events generated by external devices. The ADT component of the ADV model is directly mapped to the Abstraction component of the ALV model as both contain only the structural objects and functional core of the system. The owner sub-component of the ADV model is mapped to the Link component of the ALV model as the Link component of the ALV model contains all the information related to the communication of the non-user-interface application to the user-interface application. In the ALV model, the link between the non-user-interface application and the user-interface application is done using constraints. Therefore, with this translation the owner subcomponent will be done using a constraint-based approach.

The selection of a certain translation scheme may depend on the type of the system, and sometimes ad hoc implementation architectural models might be preferable.

6 Conclusion

A new classification scheme for user-interfaces architectural designs was presented in this paper. This new classification scheme is based on the analysis of derivations of designs. We analyzed the initial specification and the higher-order design models which correspond to models proposed in the literature at various levels of abstraction. We used a classification scheme with two levels: high-order and implementation architectural models.

We discussed issues, such as relevant properties of design families, specification notation to be used as initial design steps, and implementation biases induced by different design families. The properties were discussed using the approach suggested in [16]. We compared some architectural models with each other, and with the ADV architectural model. We showed that there are specification notations that can be used in initial design steps, such as the state-machine and VDM specification languages. We presented two architectural models that are formalized in these specification languages: Client-Server and ADV architectural models.

We presented some architectural models that are implementation-biased, such as the MVC and ALV models, and, as a consequence, the generalization would be difficult to achieve if we do not consider the implementation details.

We put the ADV model in perspective by reviewing its specification, presenting its high order architecture, and comparing it with architectures at similar and lower levels of abstraction. The ADV architectural model proposed is map-based instead of layer-based. The layered approach is common in many of the architectural models proposed in the literature. We believe that, with this approach, we can express much better the relations that need to hold between the non-user-interface application and the user-interface application.

We also believe the ADV architectural model addresses most of the problems discussed in the literature, such as application independence, overloading, and repetition of semantic con-

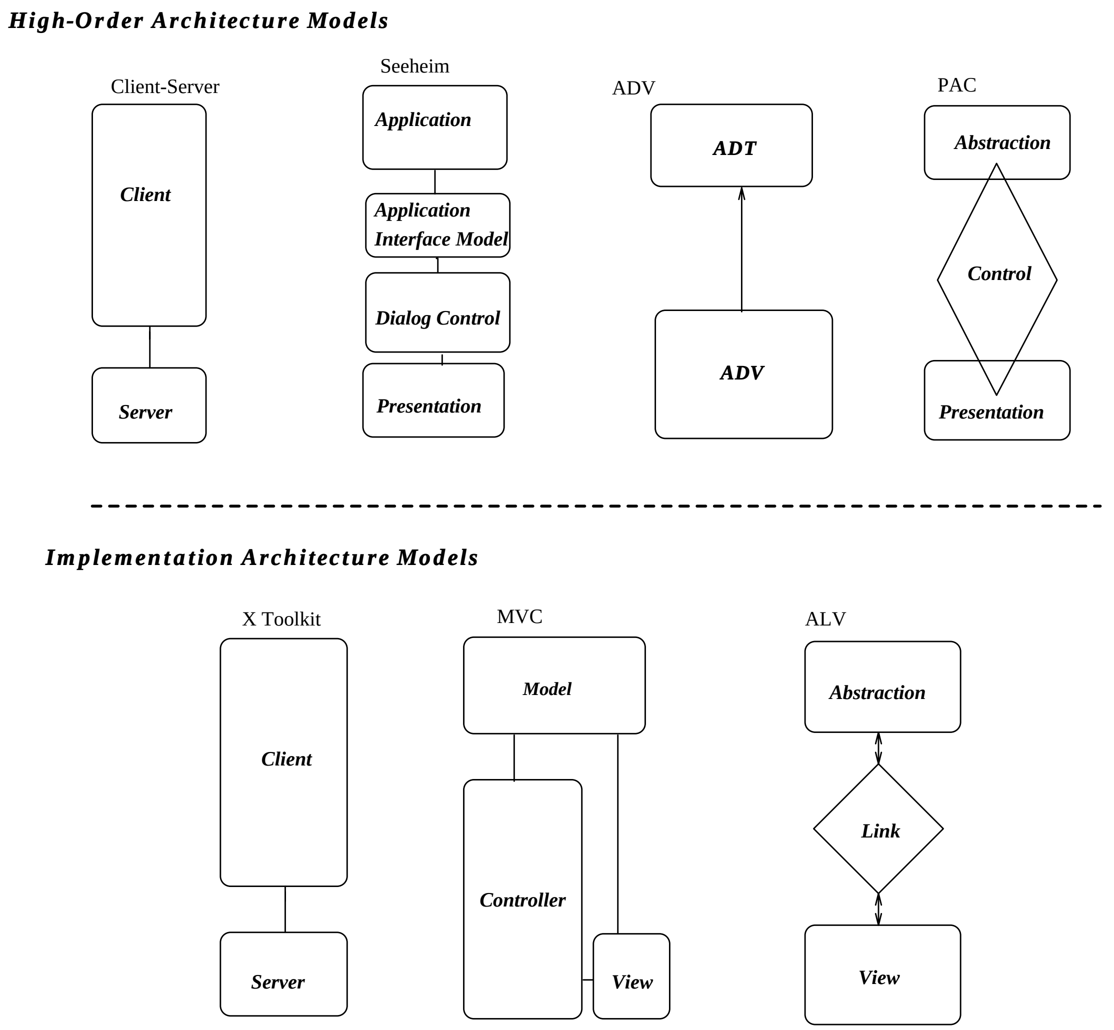

Figure 4: High-Order Architectural and Implementation Models

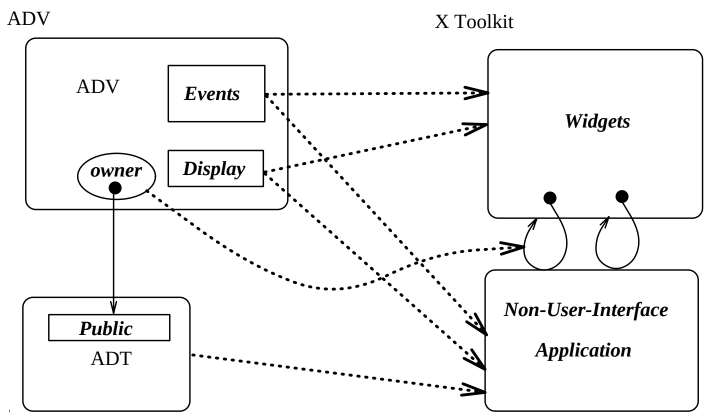

Figure 5: Translation from the ADV Architectural Model to a Client-Server Implementation Architectural Model

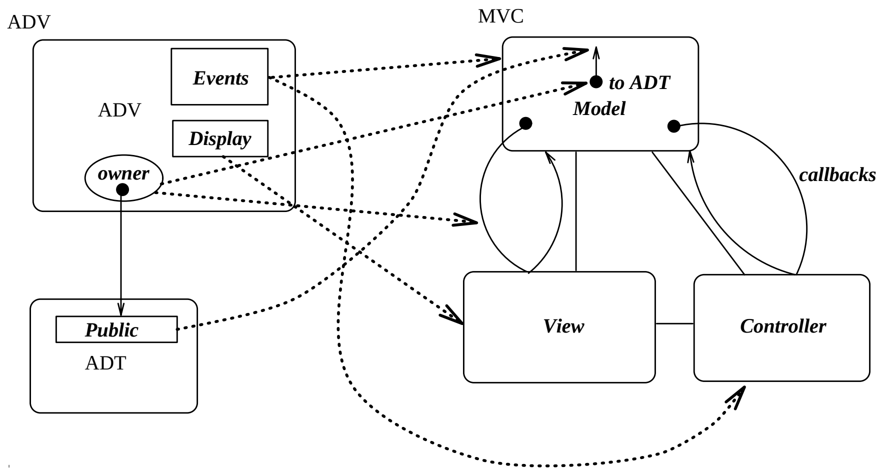

Figure 6: Translation from the ADV Architectural Model to the MVC Implementation Model

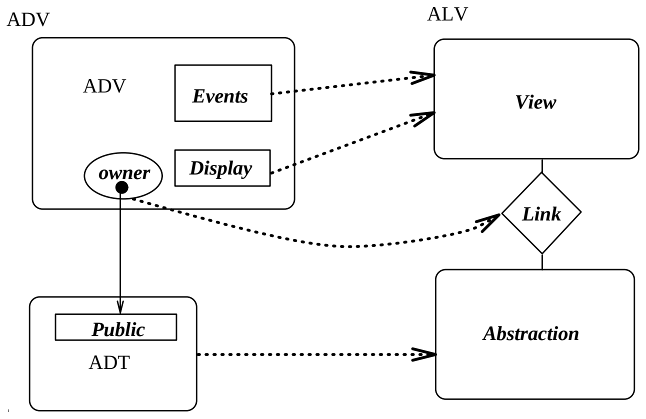

Figure 7: Translation from the ADV Architectural Model to the ALV Implementation Model

straints. The application independence is clear, as the application—referred to in this paper as non-user-interface application (ADT)—has no knowledge about its associated ADV. The repetition of semantic constraints problem is addressed by the ADV architectural model, as the ADV component has access to semantic information through the mapping—represented by the owner sub-component. The overloading problem is addressed by the ADV architectural model, as each object that composes the user-interface application has its own independent ADV representation, and, as such, has its own manipulation procedures.

Some problems are still not addressed by the ADV architectural model, such as the specification of concurrency, distributed problems [24], spatial and temporal relations among objects, and specification languages for interactive systems [3]. These problems are under study.

References

[1] Derek Andrews and Darrel Ince. *Practical Formal Methods with VDM.* McGraw-Hill Software Engineering Series, 1991.

[2] Len Bass and Joëlle Coutaz. *Developing Software for the User Interface.* The SEI Series in Software Engineering. Addison-Wesley, 1991.

[3] Luiza M. F. Carneiro. A Specification-based Approach to User-Interface Design. Internal report, University of Waterloo, December 1992.

[4] Gilbert Cockton. The Architectural Bases of Design Re-use. In D.A. Duce, M.R. Gomes, F.R.A. Hopgood, and J.R. Lee, editors, *User Interface Management and Design*, pages 15–34. Springer-Verlag, 1991.

[5] Joëlle Coutaz. The Construction of User Interfaces and The Object Paradigm. In J. Bézivin, J. M. Hullot, P. Cointe, and H. Lieberman, editors, *ECOOP'87 – European Conference on Object-Oriented Programming*, pages 121–130, June 1987.

[6] D.D. Cowan, R. Ierusalimschy, C.J.P. Lucena, and T.M. Stepien. Abstract Data Views. *Structured Programming*, 14(1):1–13, January 1993.

[7] John Dawes. *The VDM-SL Reference Guide.* Pitman Publishing, 1991.

[8] Stephen W. Draper and Donald A. Norman. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.

[9] James D. Foley. *Computer Graphics: Principles and Practices.* Addison Wesley, 1990.

[10] Mark Green. Design Notations and User Interface Management Systems. In Günther E. Pfaff, editor, *User Interface Management Systems – Proceedings of the Workshop on User Interface Management Systems held in Seeheim,FRG, November 1-3, 1983*. Spriger-Verlag, 1985.

[11] Mark Green. Report on Dialogue Specification Tools. In Günther E. Pfaff, editor, *User Interface Management Systems – Proceedings of the Workshop on User Interface Management Systems held in Seeheim,FRG, November 1-3, 1983*. Spriger-Verlag, 1985.

[12] Ralph D. Hill. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In *CHI'92 Conference Proceedings*, May 1992.

[13] C.A.R. Hoare. Proof of Correctness of Data Representations. In C.A.R. Hoare and C.B. Jones, editors, *Essays in Computer Science*, pages 103–115. Prentice Hall, 1989.

[14] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice Hall, second edition, 1990.

[15] Glenn E. Krasner and Stephen T.Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *JOOP*, pages 26–49, August 1988.

[16] James A. Larson. *Interactive Software – Tools for Building Interactive User Interfaces*. Yourdon Press Computing Series, 1992.

[17] Wm Leler. *Constraint Programming Languages*. Addison-Wesley, 1988.

[18] Joel McCormack and Paul Asente. An Overview of the X Toolkit. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, pages 46–55, October 1988.

[19] Brad A. Myers. User-Interface Tools: Introduction and Survey. *IEEE Software*, pages 15–23, January 1989.

[20] Brad A. Myers. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. In *UIST – Fourth Annual Symposium on User Interface Software and Technology*, pages 211–220, 1991.

[21] D. R. Olsen, Jr. Presentational, Syntactic and Semantic Components of Interactive Dialogue Specifications. In Günther E. Pfaff, editor, *User Interface Management Systems – Proceedings of the Workshop on User Interface Management Systems held in Seeheim,FRG, November 1-3, 1983*. Spriger-Verlag, 1985.

[22] Michael Sannella, Bjorn Freeman-Benson, John Maloney, and Alan Borning. Multi-way versus One-way Constraints in User-Interfaces: Experience with the DeltaBlue Algorithm. Technical report 92-07-05, University of Washington, July 1992.

[23] Alok Sinha. Client-Server Computing. *Communications of the ACM*, 35(7):77–98, July 1992.

[24] Paul J. W. ten Hagen. Critique of the Seeheim Model. In D.A. Duce, M.R. Gomes, F.R.A. Hopgood, and J.R. Lee, editors, *User Interface Management and Design*, pages 3–6. Springer-Verlag, 1991.