Display of Multi-Attribute Data Using a Presentation Description Language

by

Jonathan Kemble

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

August 24, 1994

Approved:

Professor Craig E. Wills, Major Advisor

Professor David C. Brown, Major Advisor

Professor Robert E. Kinicki, Department Head

Abstract

In order to make large applications that manage multi-attribute data usable, they must have an effective user interface. Application data and data relationships must be displayed in a manner that is useful for a particular user while still following principles of user interface design. A User Interface Management System (UIMS) is an application independent data presentation system which isolates the interface portion of the application and can allow a high level of customization. A presentation description language can be used to control the UIMS and allow maximum flexibility. This thesis investigates a UIMS controlled by a language that allows a user to easily describe the application data and data relationships at a high level of abstraction. The UIMS uses this language to structure application data and augment it with properties. A rule-based system then uses the augmented data along with graphical design knowledge to determine the content, layout and details of the interface used to display the data. Finally, a graphical interface is generated to present the data. A system to provide this functionality was designed and implemented. Experiences with the system showed this approach to be valid and provided ideas for future work.

Acknowledgments

I would like to thank my advisors, Professor Craig Wills and Professor David Brown for the guidance they provided during this thesis. Their ideas, advice and support are deeply appreciated. Thanks go to Digital Equipment Corporation for providing the means to complete this work. I would also like to thank Professor Lee Becker for reading this thesis.

In addition, I would like to thank Bertram Dunskus for putting up with me for a year and providing help with Framemaker. Thanks also to Jeremy Medicus and Chris Mangiarelli for providing me with a place to sleep, to Brian, Aaron, John, Steve and Sharra for not having too much fun without me, and to a cast of thousands for supporting me over the years.

List of Figures

List of Tables

Chapter 1 Introduction

It is a well known problem that the user interface portion of an application can take a significant proportion of the total programming effort. A system that automatically generates an interface for an application can help greatly. This thesis investigates a system that will automatically create an interface based on a high level description of application data.

1.1 Construction of Interfaces

The user interface portion of an application can be considered as a separate entity from the application itself. The application stores the data and implements the operations on the data, whereas the interface simply displays the data and manages user actions. Some basic features of an interface for an application include:

- the ability to handle a wide variety of data types,
- use of display attributes such as color and position to present the application data, and
- support for individual user viewing preferences.

Most graphical interfaces are created by directly interacting with the windowing system. The **windowing system** is the part of an operating system that allows applications to graphically present information in a window structure. Displays are typically built as a hierarchy of graphical elements known as **widgets**. Different windowing systems typically provide a similar set of widgets. Examples of widgets are scroll bars, buttons and editable text items. These widgets can be filled directly with application data. Widgets are normally combined to form higher level widgets called **windows** or **dialogs**. This grouping of widgets partitions application data and actions into conceptually related sections. The combination of all the widgets that are created by an application is known as a **widget tree**.

The interface typically reacts to user events by notifying the application through a callback mechanism. When a user performs an action on a widget, the interface system notifies the application which then invokes the associated callback function for that widget. This call-back function performs the requested operation and updates the widgets in response to the request. Figure 1 shows this interaction.

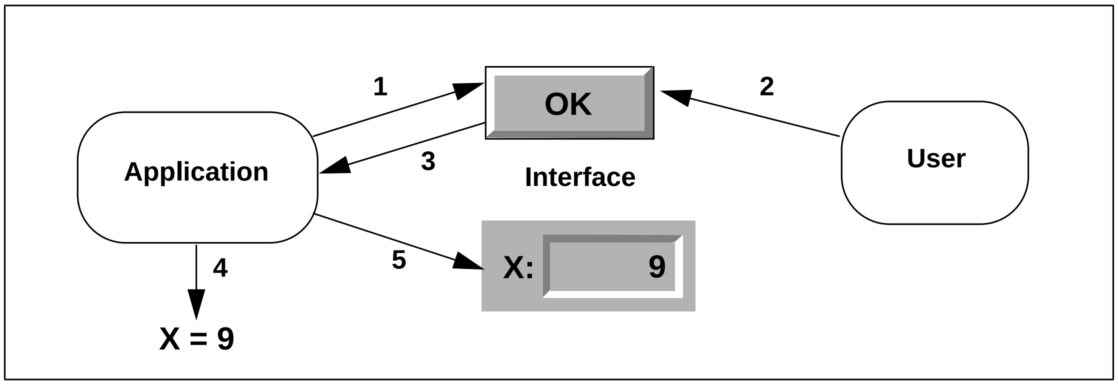


Figure 1: Interaction of Application, Interface and User

The first step in the interaction is the creation of the button widget by the application (1). Next, the user presses the button (2). The windowing system notifies the application through a callback function that it was pressed (3). The application responds by updating its internal data (4) and then displaying this data in another widget (5).

In a typical windowing system there can be many widgets that an application must manage. Though the interface can be regarded as separate, the widgets do not inherently provide any management functions. The application must manage widgets itself.

1.2 Human Computer Interaction

In order for a large application to be usable, it must have a clear and effective interface. The interface should be consistent and predictable while effectively using the resources it has available. To ensure these goals are achieved an application can make use of interface guidelines. These guidelines provide hints on layout, style and the use of display attributes such as color, position or size. Guidelines can also specify the characteristics of interaction, such as the ordering or availability of actions. Using guidelines helps to ensure that all parts of the interface are effective and consistent.

1.3 User Interface Management System

Constructing and interacting with a large widget tree while conforming to a set of guidelines can be taxing for the application programmer who must implement this functionality. The application must create and manage potentially many widgets, respond to and update each of the widgets as needed, and do so in the manner specified by the guidelines of good interface design. To remove much of the burden of dealing directly with the interface, the concept of the **User Interface Management System** (UIMS) was developed. A UIMS is a system designed to automatically generate an interface from an input description and manipulate it for an application. It encapsulates the interface generation portion of an application and allows the application to interact at a higher level than if it were manipulating the widgets directly. It contains its own interface guidelines and automatically applies them when creating interfaces.

The major components of a UIMS are outlined in the Seeheim model [Pfaff 85]. Figure 2

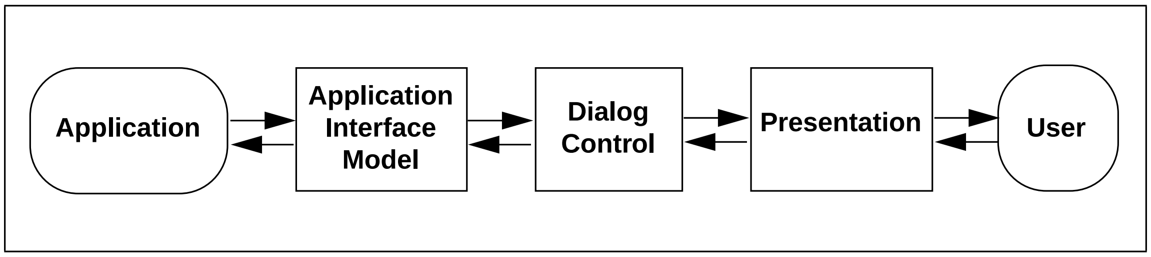


Figure 2: Seeheim Model of UIMS

shows these components. The major components of the model are the application interface model, the dialog control, and the presentation. The application interface model is the UIMS representation of the application data and actions. While the application manages the actual data and performs the commands requested, the UIMS must have some knowledge of the structure of the data and commands available to it. The dialog control section handles the interaction with the user. It determines what data to present to the user and what actions to allow the user to perform on the data. The presentation section handles the actual display of the data to the user, including the layout and choice of widgets to use in the display. A UIMS that only handles the display of data and does not manage interaction with the user is known as an **automatic presentation system**.

1.4 Programming Languages

More powerful UIMS can handle complex data structures and display requirements. Adapting a new application interface model for each application may be impractical. Instead, a UIMS may provide a language for describing the application data. Using a human readable language allows a human designer to specify data and display characteristics quickly and easily. A **presentation description language** allows a user to specify what data and relationships are to be displayed by a UIMS and what the characteristics of the interface should be. Different users can maintain separate descriptions which filter and display data to their own requirements.

1.5 Rule Systems

The interface design knowledge encoded in a UIMS can be stored as a set of rules. A **Rule-Based System** (RBS) is a system which incorporates a set of rules to encode human knowledge [Hayes-Roth 92]. A rule is a situation-action pair that is triggered when the situation described in the rule is present. If the rule is fired, its action is carried out by the system. An RBS determines which rules to trigger and fire. Rules can have different uses. A system with many rules may separate them into groups, each with a particular domain or function. In the case of a UIMS, the rules may be separated into categories such as layout or widget choosing rules.

1.6 Motivation

An example of an application that creates large sets of related data objects and requires an effective user interface is the TENNIS system [Brown et al 94]. TENNIS is a computer network service ease evaluation and estimation tool developed at WPI with support from Digital Equipment Corporation. It consists of a multi-agent expert system that reads a network description and produces comments and evaluation data concerning the ease of servicing the network. The system produces a wide variety of interrelated data and this output must be related to a potentially complex network description.

The TENNIS system generates two main types of output, messages and components. Components are nodes in the computer network. Examples of components are computers, network devices, and network connections. The components typically contain a set of information such as their name, position and status. The messages are text strings from agents within the expert system. The messages typically come from a specific agent and can refer to a component within the network. Some messages also provide numeric values that indicate an evaluation of the network.

The base system does not have an integrated user interface, but rather uses a defined protocol for receiving user requests and transmitting output. Because TENNIS takes little input and produces a large amount of output, an automatic presentation system would be beneficial.

Data from TENNIS must be presented by grouping similar items into sets, such as network components and messages. Relationships between items in dissimilar sets, such as when a message refers to a particular component, must also be displayed. These potentially complex relationships within the data must be described to the UIMS if it is to properly present them. This suggests that a presentation description language would be useful.

1.7 The Thesis

The central theme of this thesis is that an automatic presentation system can be generated that is driven entirely by data and descriptions of the data. This idea is investigated through the design and implementation of a system named COURT.

A literature survey was performed to examine existing work in the area of UIMS. The COURT system was designed, implemented and evaluated. COURT operates on a set of data and a description of the data expressed in a presentation description language. The end result is an interface designed to represent the data and relationships within the data. The generated prototype system has been tuned to work with TENNIS.

The remaining chapters are structured as follows: Chapter 2 examines existing work in the area of UIMS and compares these systems. Chapter 3 discusses areas of improvement in existing work and outlines the goals of this work. Chapter 4 presents the design decisions made while developing COURT. Chapter 5 details the portions of the design that were implemented in COURT. Chapters 6 and 7 provide an example of the use of COURT and an evaluation of the system, respectively. Chapter 8 lists the conclusions reached and possible future work that can be done in this area.

Chapter 2 Related Work

Existing research in automatic generation of interfaces has made progress toward decoupling the interface from the application and allowing the user to control it through a language. The following systems can all be described in terms of the UIMS model. One major differentiating characteristic between the systems is the amount of interface building knowledge within the system. The systems will be presented in roughly increasing amount of interface design knowledge, grouped by the general architecture of the system. After examining the systems, they will be compared based on their input requirements.

2.1 Interface Toolkits

Interface toolkits contain the least amount of interface design knowledge. These systems generally form a thin layer between the windowing system and the application, providing simplified access and increased portability. By combining widgets into prefabricated windows, the application's job of interface construction is partially simplified. However, an application must normally fill widgets with data and maintain them itself. Some examples of toolkits are XIT [Herczeg et al 92] and GINA [Arens et al 91]. Although these systems allow a high degree of flexibility, they operate at a low abstraction level and as such are designed for software developers, not end users.

2.2 Rule-Based UIMSs

Rule based UIMSs take application data or rough interface description as input, transform it via the application of rules, and produce a more concrete interface description as output. The input can be a description of the application data objects, the application actions, a rough interface specification, or any combination of these. This input is normally used as the basis for a working design used in the system. The rules used by the system trigger based on certain patterns in the working design. When fired, a rule will normally further constrain the working design by either adding interface elements, or by limiting the possible choices for interface elements. Rules usually have a particular purpose, such as layout of windows, enforcing a certain style, or adding consistent functionality. The output of a rule system is either a complete interface, or an interface description that can be used by an external application to create a working interface.

Examples of rule-based systems are described in the following.

2.2.1 ITS

ITS [Wiecha & Boies 90] is a rule-based system designed to provide consistent styles in dialogs. An application provides a rudimentary interface description as input to ITS. This interface description specifies items to be displayed, choices available to the user, and actions to invoke when choices are made. Style rules in ITS match on specific portions of the interface description and transform them by adding more specific descriptions to the specification.

For example, an interface description containing a set of items for the user to choose among may be transformed into an interface containing a title and two selectable text items. Figure 3 shows this transformation graphically. On the left side are the interface elements provided in the description. On the right are the widgets generated by the rules, and the location of the original elements. The Vertical Group and Horizontal Group widgets provide layout policies for their child widgets. The interface structure produced is then supplied to the interface implementation portion of ITS.

The use of rules allows for a reusable style knowledge base. Modification of the rules will

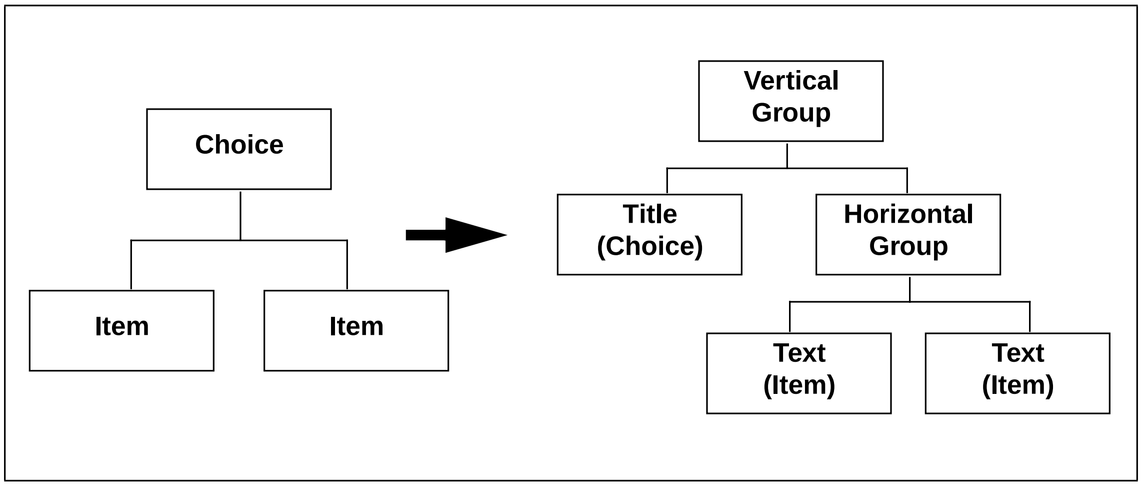


Figure 3: Transformation of Interface Description in ITS

modify all future interfaces generated by the system. ITS incorporates useful rules for deciding what widgets to create based on the input description. However, ITS does not provide for the display of application data; it only supports user interaction. In addition, the application must partition the interface elements to determine the content of windows. ITS concentrates on refining the formatting of the display.

2.2.2 Integrated Interfaces

Integrated Interfaces [Arens et al 91] is a interface design system which also uses rules to map from an application domain model to interface domain model. Objects in the application domain model and interface domain model are represented in the NIKL frame system. The rules to map from application to interface are stored in a propositional logic system named PENNI. The rules are separated into high level rules that provide a presentation style for a particular set of data, and low level rules that map application data objects into interface objects. The presentation design works in three phases: realization, selection, and redescription. Realization examines a data object’s attributes to determine to which application object class it belongs. Selection determines the most effective interface encoding to represent objects. Redescription maps data values to interface objects.

Integrated Interfaces was developed for the domain of naval war ship situation display. Because this domain contains potentially complex data objects and presentation requirements, many of the rules are application specific. Rules exists to map specific data objects such as ships into icons that represent their state. In addition, special cases of presentation requirements are encoded in sets of rules which are enabled when these requirements are specified.

Mapping from an application model to an interface model allows either of these models to change while the system remains the same. Integrated Interfaces also incorporates the duties of determining the content of windows. Though the system attempts to derive as much information as possible from the data alone, the complex domain requires application specific rules. This domain dependence makes the system harder to be adapted to other domains.

2.2.3 UIDE

UIDE [Foley et al 89] automatically designs interactive interfaces using design and style transforms. Transforms are essentially rules that are designed to provide consistent style and functionality in interface. The input to UIDE is a frame-based description of the application data objects along with their attributes, and the actions available on the data objects. A frame based representation means that the objects are instances of classes arranged into a hierarchy. Each object has a number of attributes, and some of these attributes may refer to other objects. The transforms provide consistent style by adding constraints and attributes to an interface description. Figure 4 shows the main components of UIDE.

One interesting aspect of UIDE is the automatic addition of interaction techniques based on the data types. The system can automatically add an object selection mechanism for objects that can be provided to actions. By examining the object types and the action input

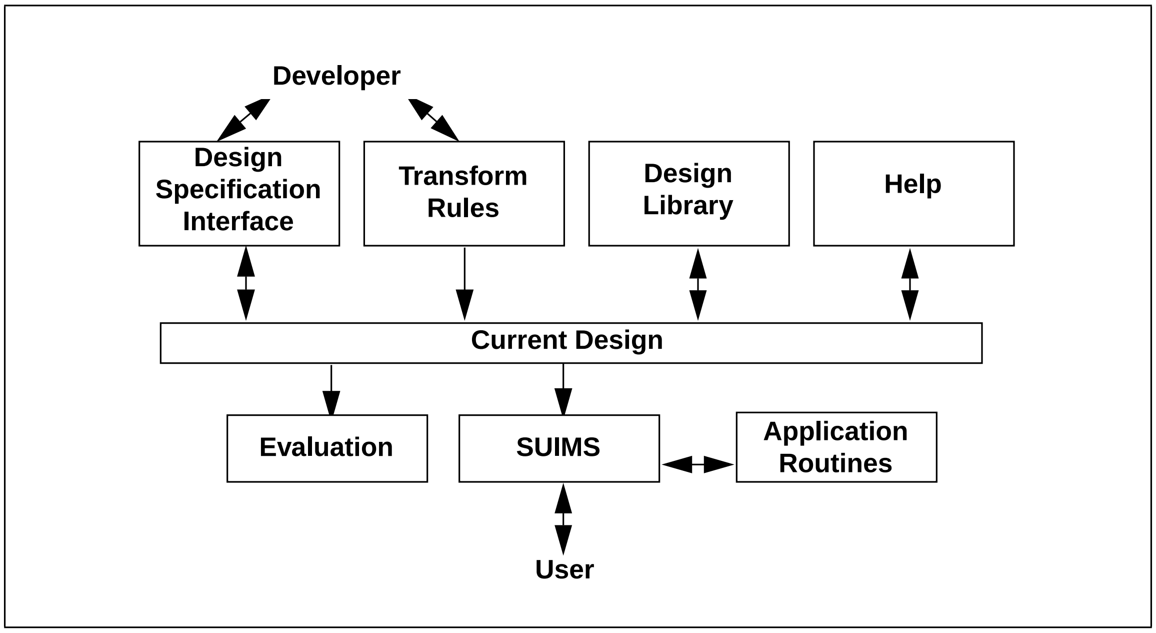


Figure 4: UIDE System Components

requirements, the system can determine what objects can be selected together. In addition, the system can automatically generate context sensitive help using the action preconditions and postconditions.

2.2.4 DON

DON [Kim & Foley 90] is a rule based system based on UIDE [Foley et al 89]. DON creates an interface for an application. Rules are partitioned into knowledge bases and applied by the organization and Presentation managers as they are activated. Figure 5 shows the major system components. A high level specification is provided by the interface designer. This specification provides information about the application data and actions, including attributes of data and actions, and pre-conditions and post-conditions of actions. The conceptual design knowledge combined with user preferences is sent to the Organization Manager. The Organization Manager creates a rough interface design by grouping major actions and data items together. Data and actions are grouped into windows by their inputs, function, or outputs. The output of the Organization Manager is then transformed by the

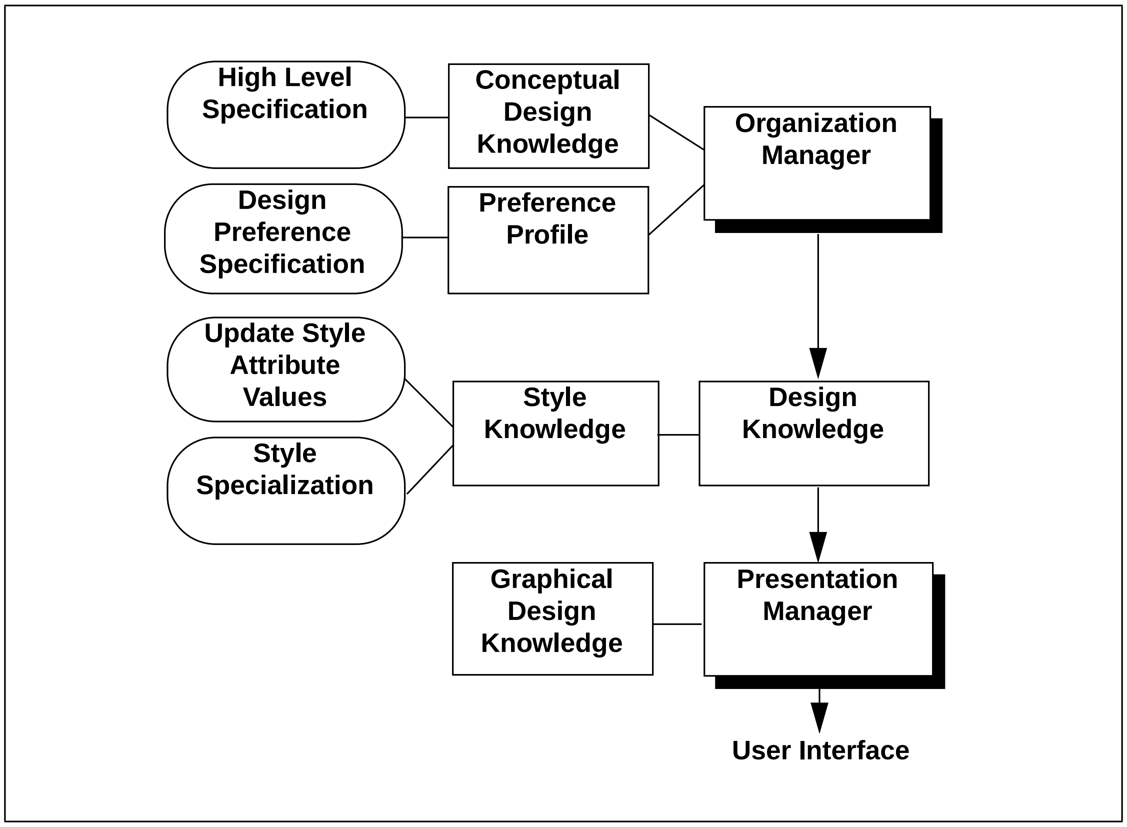


Figure 5: DON System Components

style knowledge to conform to a particular style. The interface designer can modify the style rules by specifying values for attributes in general, or by specializing an object within the hierarchy to a certain data type. The Presentation Manager then creates a detailed design by applying graphic design knowledge to the intermediate design.

Breaking the interface design process into rough design and detailed design phases forms a clean decomposition of function. Domain independent conceptual design knowledge, style knowledge, and graphical design provide a well rounded and reusable set of knowledge.

2.2.5 Humanoid

Humanoid [Szekely 90] produces an interactive display using a template matching process. Predicate subsumption is used to match an application object tree against a set of templates which specify the widgets to represent the application data. Figure 6 shows the use of Humanoid's template matching. A template contains a listing of the data types it can present along with the widget used to present them. For each application data object, the closest matching template is selected and the widgets specified in the template are generated. Because the data items specified by the widget may not be leaves of the application data tree, the process can be repeated for each data item specified in a template. Templates can inherit from other templates to easily allow specialization. The use of templates allows the system to create interfaces for data that can change at runtime.

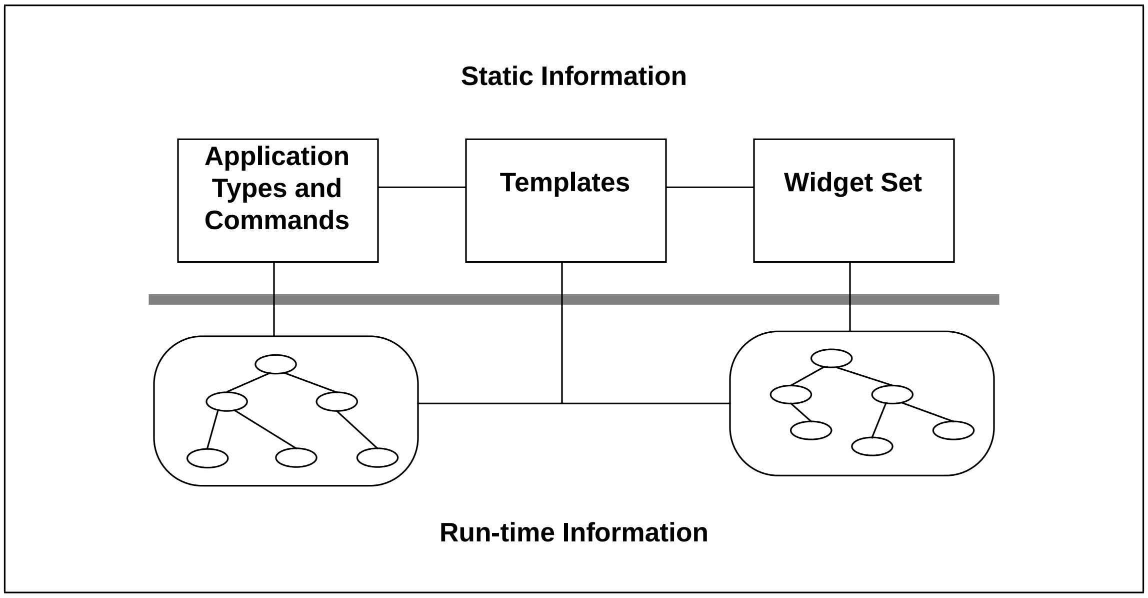


Figure 6: Use of Templates in Humanoid

2.3 Search

Search is an AI technique of finding a solution to problem by continually generating a candidate solution, evaluating its suitability, modifying parameters used to generate a candidate, and generating a new candidate. APT [Mackinlay 91] uses a depth first backtracking search architecture to create a presentation for a set of data. APT first partitions a set of data into smaller units for display. It then uses evaluation criteria to select a graphical encoding for the data. Composition is then performed to combine the representations of the partitioned data. This algorithm can be recursively applied to a set of data, making the search depth first.

If any decision proves unusable at a later stage, the system can backtrack to the decision point and retry using an alternate. A decision is unusable if either an encoding cannot be selected, or encodings cannot be composed. Selection and composition are controlled by expressiveness and effectiveness evaluation criteria. Expressiveness criteria checks that a graphical encoding can represent all the data that is needed. Effectiveness criteria evaluates how easily a person can perceive the data represented by an encoding. Figure 7 shows

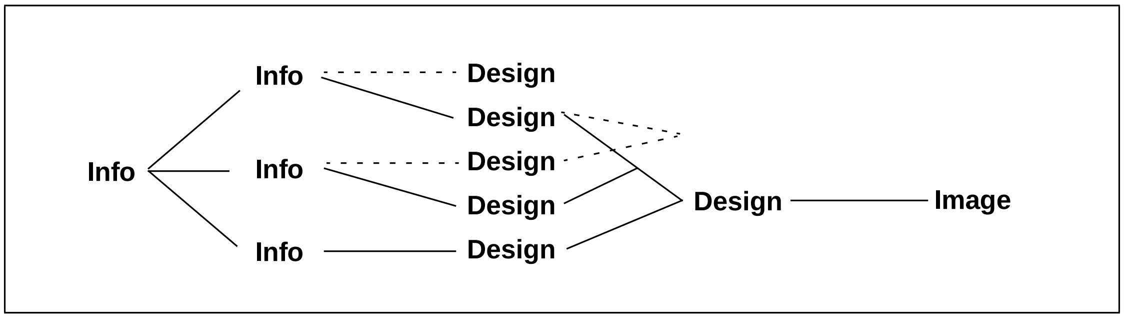


Figure 7: APT's Search Architecture

APT's search architecture. The solid lines represent the final design. The dotted lines represent backtracking during decision making.

The use of search as an interface design technique bypasses the issue of splitting the design process into rough design and detailed design phases. Layout is split between the data partitioning and composition of encodings. Expressiveness and effectiveness evaluation criteria provide a powerful means for selecting graphical encodings.

2.4 Demonstration and Direct Manipulation

Demonstration systems allow a user to directly specify an interface by either providing a portion of the design characteristics, or by specifying an example of the design. Typically these systems will provide a graphical interface in which the interface designer can provide design examples. Two relevant systems are GOLD and SAGE.

2.4.1 GOLD

GOLD [Myers et al 94] automatically generates business charts from examples provided by the user. A user draws a graphical element, maps it to a data element in a set, and the system extrapolates this example to include all the data elements in the set. A number of graphing heuristics are used, including heuristics for inferring characteristics of data in a table, determining the type of chart to use, creating chart axes, mapping data to graphical objects, and neatening charts.

2.4.2 SAGE

SAGE [Roth et al 94] is a system designed to create non-interactive presentations of data. The interface designer provides design directives to the system through a direct manipulation interface named SAGEBRUSH. The user creates a partial design to encode a set of data. Prototype interfaces can be stored and retrieved in SAGEBOOK. The design directives along with application data are then provided to the SAGE system which creates the interface and maps data to the widgets. Figure 8 shows the system components in SAGE.

SAGE goes beyond a simple demonstration system, however. It incorporates a knowledge base which allows it to map data to interface objects without any examples from the user. It maintains a rich representation of the graphical elements, along with a comprehensive description of the application data [Roth & Mattis 94]. Application data is characterized in many dimensions, including set ordering, domain of membership, relational characteris-

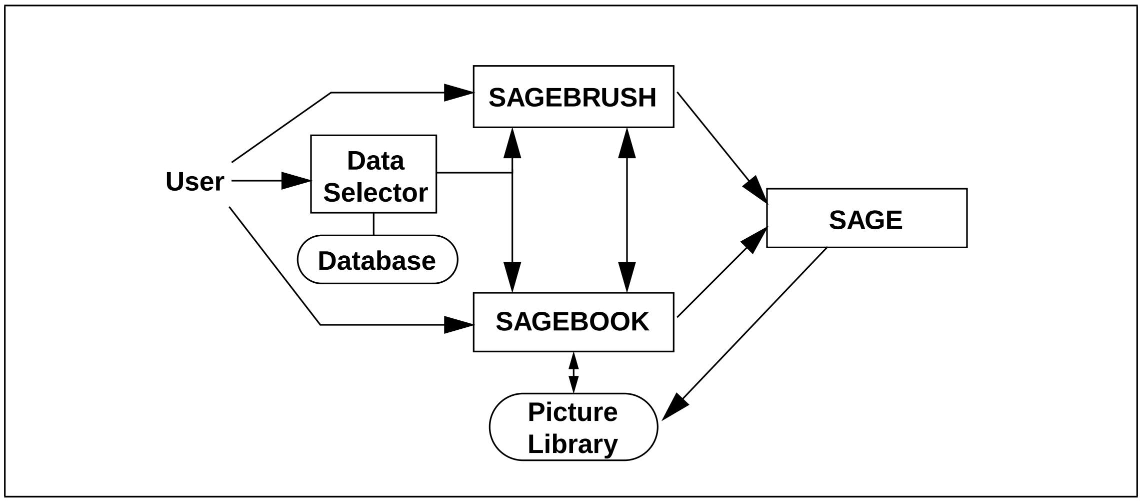


Figure 8: SAGE System Components

tics, cardinality, etc. With a large amount of information about the data, SAGE is able to determine what graphical elements will best represent data.

This description of data is analogous to a description of the actions available in an application. Application actions are described by their function, operands and preconditions. These descriptions allow an interface design system to group the actions and provide an interface for accessing them. When producing a presentation of application data, an equivalent description is needed to allow the system group and represent the data.

2.5 Intelligent Interfaces

Intelligent interfaces use knowledge bases to process and display data. These systems are capable of generalizing patterns of user interaction, and dynamically modifying the interface to suit the immediate task. The systems often incorporate a user model which allow them to extract meaning from the interaction. Examples of intelligent interfaces can be found in [Sullivan & Tyler 91]. Though these systems are able to dynamically generate interfaces with little or no work from the user, it is difficult to customize the attributes of the interface.

2.6 Comparison of Systems

Examining these systems together reveals characteristics for differentiating them. The following list summarizes these characteristics.

- *Intelligence in system*. This characteristic measures the type and amount of interface design knowledge present in a system. Interface toolkits have no knowledge, rule based systems have varying amounts of knowledge, and intelligent agents contain the most knowledge.
- *Interactiveness of interface produced*. This distinguishes between systems that produce interactive displays from those that produce presentations of data only.
- *Abstraction level of interface specification*. For systems that allow the user to specify a partial interface design, this interface specification can be at a low or high abstraction level. A low abstraction level means the user supplies detailed interface elements such as widgets, encoding mechanisms, or window layouts. A high abstraction level means the designer specifies properties or constraints of actions.
- *Abstraction level of data specification*. The abstraction level of the data can vary from being low, in which case only the data values are supplied, to high in which case the data is augmented with semantic and relational information.
- *Method of input*. The input method can vary between procedure calls made from within the application, to a description provided in some standard language, to examples provided by an interface designer through a graphical interface.

This thesis is concerned with the abstraction level of the input provided to an interface design system. Sorting the existing systems by the interface and data description abstraction level characteristics reveals a distinct grouping of the systems. The systems separate into five main groups, indexed below by the type of input they take. Table 1 summarizes this grouping of systems.

Table 1: Comparison of Existing Work

Input Type	Example Systems
Concrete Interface Components	**XIT**
Abstract Interface Components	**ITS, JADE, Mickey**
Application Data and Actions	**Integrated Interfaces, UIDE, DON, Humanoid, UofA*, VUIMS**
Abstract Data	**SAGE, APT**
Concrete Data and Interface Components	**GOLD, GARNET**

2.6.1 Concrete Interface Components

These systems take as input a low level description of the interface to produce. These system are designed to simplify access to the underlying windowing system and not to apply any design knowledge to the input description. A typical input description for a system like TENNIS would include a list of windows in which to display messages or network components, along with the contents of the windows. The systems are mainly interface toolkits, such as XIT.

2.6.2 Abstract Interface Components

These systems apply layout or style knowledge to a more abstract interface description. The input abstraction level lies somewhere between a listing of the contents of a window, and a rough window layout, possibly including hints about style. An input for TENNIS may specify the messages that should be grouped together, along with some hints about the style of the display, such as a list. The systems apply knowledge to determine the exact layout of the window and the style of presentation and interaction. These systems can be summarized as layout and style enhancing systems. Some systems that fall into this category are ITS, JADE [Vander Zanden & Myers 90], and Mickey [Olsen 89].

2.6.3 Application Data and Actions

From a description of the application data and actions that can be performed on the data, these systems produce an interactive interface. The application data description typically includes some kind of class hierarchy into which application objects can be placed. Network components in TENNIS fit nicely into a hierarchical classification. The description of the actions often includes information about their function, operands and preconditions. The system can then determine the content, layout, and style of the interface generated. Some example systems include Integrated Interfaces, UIDE, DON, Humanoid, UofA* [Singh & Green 89], and VUIMS [Pittman & Kitrick 90].

2.6.4 Abstract Data

This category of systems input a description of application data and produce a non-interactive presentation of the data. Because the interface is strictly for presentation, these systems are generally the automatic presentation systems. The data description can range from a set of data values with no other information, to a set of data values augmented with properties, semantic and relational information. Describing the TENNIS data in an abstract way would include descriptions of the meaning of messages, such as important or dangerous, along with descriptions of the component attributes, such as status or position. The system applies layout and encoding rules to the data to produce an effective display. Some example systems are SAGE and APT.

2.6.5 Concrete Data and Interface Components

These systems take as input a set of application data and a partial interface description. They typically have demonstrational interfaces that allow a designer to specify by example how data is to be mapped to interface elements. A partial interface description is built and the system applies heuristics to complete the design. For example, to produce a TENNIS interface the designer would select a interface object such as a window and bind it to a data element such as a message. The system would extrapolate this specification by adding all messages into the window. Because the knowledge in the system is mainly focused on extrapolating a partial design, the input data description can be weak. Example systems of this type are GOLD and GARNET [Myers et al 90].

2.6.6 Summary of Comparisons

The systems that take interface components as input require that the application provide some form of rough interface design, such as determining the content or layout of windows. The toolkit systems which take as input concrete interface descriptions are not useful as UIMS since they provide little automation of the interface generation. Systems such as ITS and Mickey which process the abstract interface descriptions concentrate on style and layout within a window and do not determine the contents of a window. These systems are also limited in their ability to automatically create displays.

The demonstrational systems which take data and interface components as input are limited since they require an interface designer to provide information as the interface is generated.

Systems such as DON and Humanoid are more capable of creating interfaces from the application data and action descriptions. Many systems have been developed to provide this functionality.

Systems designed to create presentations based on the data only are interesting because they must perform all of the interface design themselves. Since no information is supplied about the interface itself, the system must determine content and layout from the information provided about the data.

This comparison of existing work reveals a lack of development in the area of automatic presentation system driven by abstract data descriptions.

2.7 Summary

Previous work done in the area of automatic generation of user interfaces has generated a number of systems. These systems vary in their method of operation, their input requirements, and the output created by them. A comparison of the systems reveals there are relatively few systems that are able to create interfaces based on a description of the data only. This thesis focuses on an automatic presentation system whose input requirements are data and a description of the data. The goals of such a system are outlined in the next chapter.

Chapter 3 Problem and Goals

Existing systems have made progress toward automating the generation of interfaces. These systems incorporate design knowledge and are capable of producing effective displays. However, there are many deficiencies in these systems that need to be addressed. This thesis attempts to improve on previous systems by minimizing the amount of interface specification in the input data while still creating an effective display of the data. In this chapter, the deficiencies of the existing work are examined and the goals of this thesis are described.

3.1 Deficiencies in Existing Work

Each type of system discussed makes progress in automatically generating interfaces. Each type of system is also open to improvement.

Interface toolkits operate at too low a level of abstraction to be useful as automatic interface generators. Though the systems provide a consistent presentation style in their interfaces and provide some predefined layouts, few constraints are enforced on the interaction style or window layout of the resulting interface. The application must provide most of the interface design itself, and must manage it entirely. A more automated system is required.

Systems that provide refined window layout and style, such as ITS and Mickey, contain useful heuristics for improving window effectiveness, but also do not provide sufficient functionality. Application data is not automatically displayed, aside from items grouped into windows in the input specification. The designer must provide the window contents.

Systems such as DON and Humanoid, that produce an interface based on a description of application data and actions, contain knowledge for determining window contents, layout and style. These systems contain a limited amount of graphical design knowledge, but do not focus on presentation layout. The application data description includes limited semantic information. The action descriptions contain information that can be used to determine grouping and layout based on function and operands, but the data has no such descriptions.

Automatic presentation systems such as SAGE and APT are specifically designed for presentation of application data. These systems contain knowledge for producing windows to represent the application data effectively However, the input description does not always provide much semantic information about the data. Though SAGE can allow data to be characterized, APT allows only limited descriptions of the application data. Display design knowledge is used to partition data, but this could be made easier if the data description included semantic information.

Demonstration systems require only interface elements and data objects to be linked together. Little semantic information is supplied on the data, and the interface design process is not automatic.

None of the systems described in the previous chapter allows for easy modification of what data is to be displayed. Most systems specify that the application produces a set of data and *all* the data must be displayed. The filtering of data is external to the interface generation system. In addition, some systems allow style preferences to be expressed, but the content and layout is normally generated automatically based on rules. The content and layout of data should be easily modifiable to allow for multiple users to view the same data differently.

3.2 System Goals

The goal of this work is to investigate an automatic presentation system that designs an interface based on semantic information about application data. A list of more specific goals follows:

- *Allow description of the interface at a high abstraction level.* The designer must be able to specify characteristics of the interface in an abstract manner. The designer should not have to build a complete interface using low level interface objects, but rather be able to describe the attributes of the interface.
- *Allow description and filtering of the data displayed.* The designer must be able to describe the types of data generated by the system and form abstract views of the data. Relationships between the data should also be expressible. The designer should have the ability to attach meaning or importance to any particular type of data or data relationship. These descriptions will be used to determine how the data is displayed.
- *Allow multiple users with differing requirements.* Different users will have different requirements on what data should be displayed, and how the data should be displayed. The system must allow multiple users to easily view the same data.
- *Incorporate human computer interaction guidelines.* The system should incorporate guidelines governing good interface design when creating the interface. General principles concerning interface object attributes such as size, color and position should be followed in order to create an effective and efficient display.
- *Allow easy modification of style*. Following interface design guidelines still allows for the system to produce interfaces in its own style. The output style should be easily modifiable to allow alternate displays should it be desired.
- *Decouple the interface from the application.* The interface should be loosely coupled with the application. The application should not be required to manipulate the interface at a low level. The interface should manage itself as much as possible, only send-

ing user requests to the application.

- *Generate the interface to present the data.* Though the designer may describe attributes of the display, the system should be able to generate a widget tree using the data to be presented and the descriptions of the data. The system should determine the best interface object to use to display a certain data object, and the best interface object attributes to represent relationships.

3.3 General Structure

Figure 9 shows the overall functionality of the COURT system. The general structure is

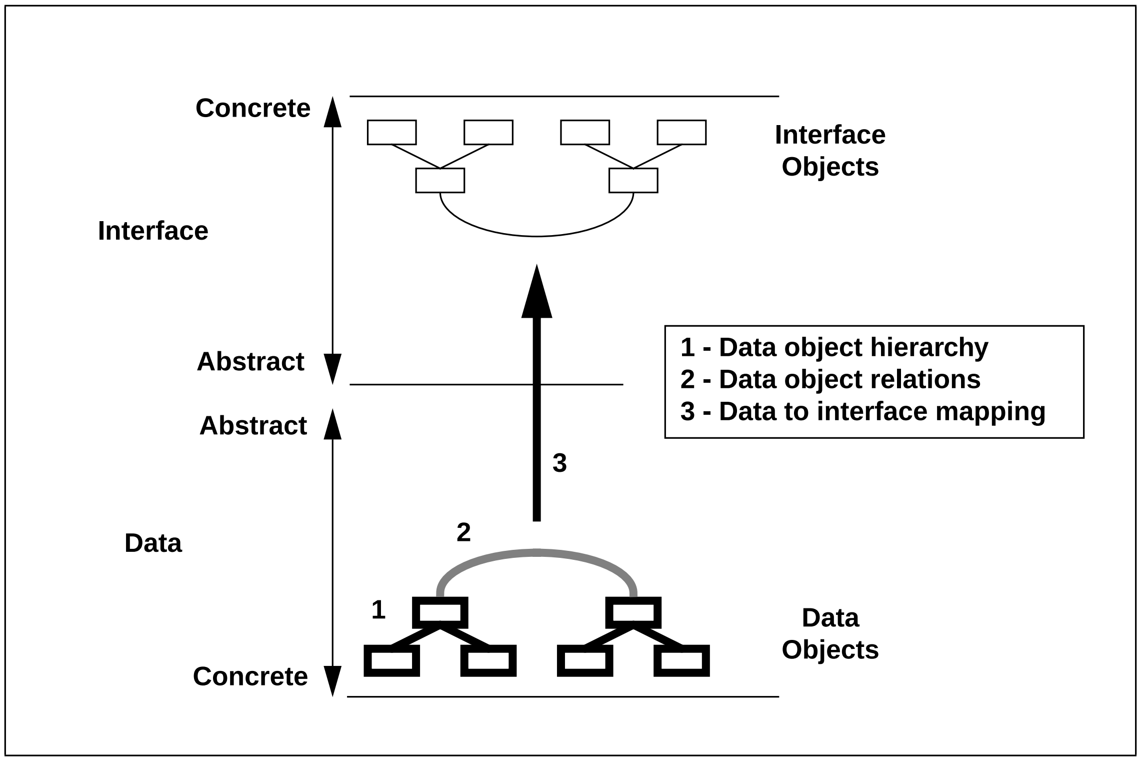

Concrete
Interface
Abstract
Abstract
Data
Concrete
Interface Objects
1 - Data object hierarchy
2 - Data object relations
3 - Data to interface mapping
3
2
1
Data Objects

Figure 9: Data and Interface Abstraction Levels

the mapping of data objects to interface objects. Data is described as a hierarchical structure with relationships among objects. The data objects are augmented with properties that will describe the semantics of the data. The hierarchical structure, the relationships, and the properties together are used to determine the content and layout of windows used to represent the data.

3.4 A System Driven by Data

Many of the system goals can be satisfied by requiring the designer to provide only data descriptions to the system. The data descriptions can be at a high abstraction level, and can include filtering. A sufficiently rich data description provides the system with structural, relational and semantic information. The interface can be generated entirely based on this information about the data. Modifying the data descriptions directly affects the generated interface. As the data is described differently, the resulting interface is modified to accommodate these changes. COURT is designed using the philosophy that a sufficiently rich description of the data is all that is required to generate an effective interface.

3.5 Comparison of COURT With Other Work

COURT focuses on minimizing the amount of interface description that needs to be provided to a UIMS. Like other systems, COURT builds an display based on interface design knowledge. It takes an abstract description of application data as input, meaning it falls into the category of an automatic presentation system. It differs from past work by allowing a description of data that can provide a large amount of semantic information and allow easy modifications of viewing parameters for multiple user types.

Table 2 shows how COURT relates to the existing systems.

3.5.1 Advantages Over Other Systems

COURT has some advantages over other systems. Because no interface specifications are required, the application and interface designer are freed from any consideration of interface-specific issues. The only input from the interface designer is a description of the data.

Table 2: Comparison of Existing Work and COURT

Input Type	Example Systems
Concrete Interface Components	**XIT**
Abstract Interface Components	**ITS, JADE, Mickey**
Application Data and Actions	**Integrated Interfaces, UIDE, DON, Humanoid, UofA*, VUIMS**
Abstract Data	**SAGE, APT, COURT**
Concrete Data and Interface Components	**GOLD, GARNET**

The use of data descriptions allows the designer to easily change the structure and relationships of the data, and thus the resulting display. Changing the structure of the data in other systems may require additional updates to the interface descriptions provided. In COURT, changes to the data descriptions are reflected in the interface with no additional work.

The data description provided to COURT can be changed to suit the needs of different end users. The same data can be visualized in different ways depending on what is most important for particular classes of users. Other UIMS do not take into account the differing needs of users.

3.6 Summary

The goals of this thesis are to explore the creation of an automatic presentation system that inputs data descriptions written at a high level of abstraction. Using semantic, structural and relational information about a set of application data, the system will automatically produce a display which conforms to a set of user interface guidelines. The use of a data description allows the resulting interface to be modified easily and provides an easy means for creating alternate displays for different user types. A system designed to meet these goals is described in the next chapter.

Chapter 4 System Design

COURT was developed with an example application in mind to ensure that the system remained practical, and to help examine design issues. The TENNIS system was used as an example of an application that provides a set of multi-attribute data to be presented to the user, but has no tightly coupled interface. The TENNIS system is a multi-agent expert system which provides an evaluation of a computer network with regards to its ease of service. The primary output is in the form of textual messages from agents. Most of the examples provided in this chapter are from the TENNIS system. For a detailed description of TENNIS, see [Brown et al 94].

4.1 Overall Design

The system goals dictate that the system be designed as a UIMS that focuses on the presentation of data. Returning to the Seeheim model shown in Figure 2, it becomes apparent that the components most relevant to COURT are the application interface model and the presentation components. The application interface model component provides all information that is required about an application. The presentation component determines the display that should be generated to represent a set of data. Because COURT focuses on the description of data and its presentation, these two components are fundamental to the system design. Because COURT does not concentrate on interaction with the user, the dialog control component of the Seeheim is not as important.

The system takes as input a set of application data, a description of the data, and a set of rules used to transform the data objects to interface objects. It produces as output an interface to present the application data. Figure 10 shows a simple system block diagram. The

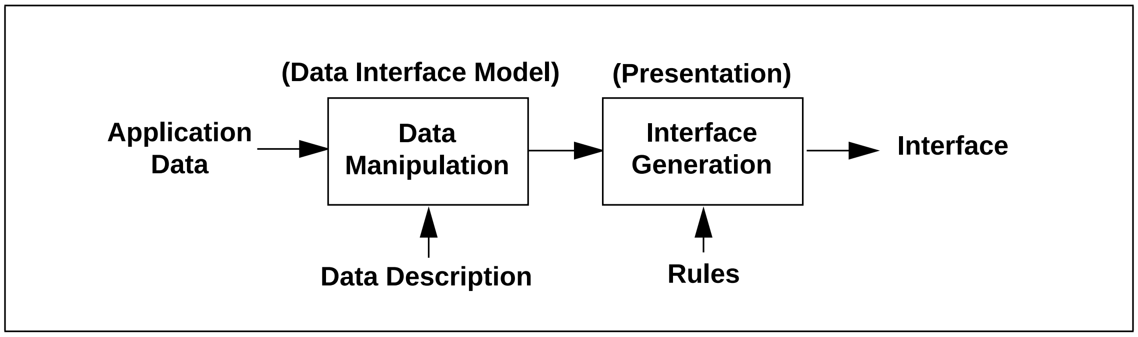


Figure 10: COURT Simple System Block Diagram

major functions of the system are the manipulation of the data to match the description, and the transformation of the data into an interface. These two functions are examined in more detail.

4.1.1 Data Manipulation

Data manipulation is the process by which data is read into COURT and converted into a form that represents the data description. Before it can be manipulated, the data and its description must first be read. The application data is read by the **application interface**, and the data description is read by a **parser**. The data is then converted into a form that COURT can understand. This is known as **recognizing** the data. The data is then grouped into a more complex structure representing the description. This grouping is known as **augmenting**.

4.1.2 Interface Generation

The transformation of the data into an interface involves the use of rules that match on particular data structures and create corresponding interface objects. The use of rules to create an interface has been shown to be effective in previous work [Roth et al 94], [Kim & Foley 90]. The transformation of data into an interface is known as interface generation. The transformation can be roughly broken into two main phases, the **rough design** and **detailed design**. Rough design determines the layout of the interface, and detailed design determines the exact interface objects to use. This approach is shown to work in [Kim & Foley 90].

4.1.3 System Structure

Figure 11 shows a more detailed system diagram. The parser, recognizer and augmentor are contained in the data manipulation half of the system. The interface generation half includes the rough and detailed design phases. The diagram shown includes more detailed information about each of the system components. These details will be described in later sections of the thesis.

The general flow of information through the system is as follows: The parser reads a data description provided in a script by an interface designer. The parser builds an internal representation of the data description in the form of data element and abstraction classes. The recognizer reads application data and classifies it, based in the information provided by the parser. Data is abstracted into a tree structure and augmented with properties. The interface generation section then applies design rules to create interface objects which in turn create graphical elements.

The following sections examine the system components in more depth.

4.2 Application Data

There are a number of requirements on the representation of the application data. They can be summarized as follows:

- *Flexibility in representation.* The representation must allow as many applications as possible to be used with the system. This requirement means the representation must be flexible and must be able to handle all the types of data an application may produce.

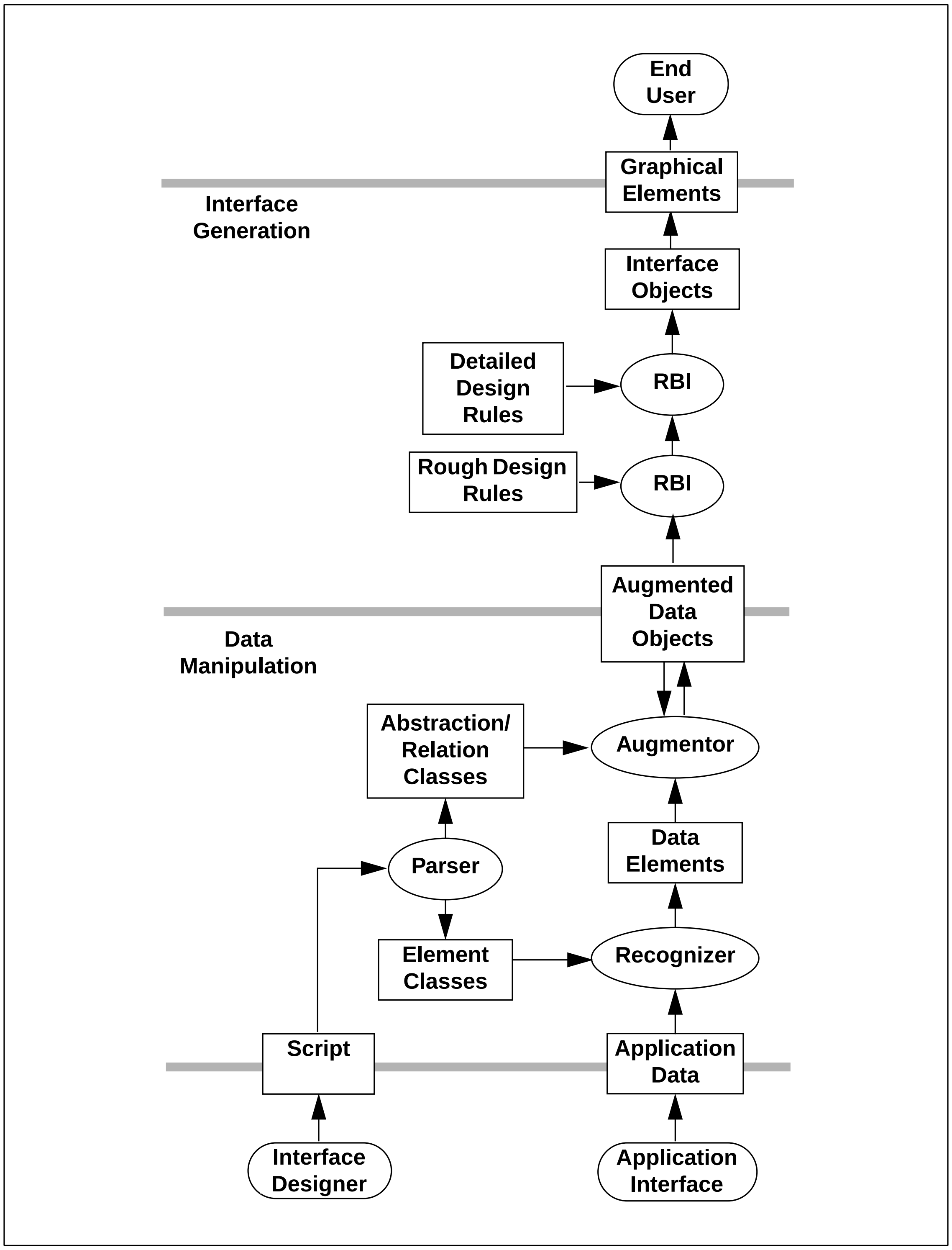


Figure 11: COURT System Structure

- *Easy combination into larger structures.* The data in COURT is combined into larger and more complex structures. Thus, representing the data in manageable units makes it easier to combine.
- *Contain little description information.* The data should contain as little description as possible. The data description provided by the interface designer should contain all the information about the data.
- *Incorporate object-oriented principles.* The data should be structured according to object-oriented principles. The most important aspect of this is encapsulating information.

Based on these requirements the application data is represented in small units called **data elements**. A data element represents a group of data items that form a single piece of information. Using this approach, all data items that are related together are encapsulated into an element. Combining the data items allows them to be more easily referred to, while keeping the data element size small allows them to be easily combined in later stages. A typical data element from the TENNIS system may be a network component such as a computer, or a message from an agent.

Using small data elements allows COURT to work with a wider variety of applications. If COURT required large complex data structures as input, additional processing would be required to assemble application data into this format. Any complex structures present in existing application data can more easily be broken into the smaller units required by COURT than vice-versa.

The individual data items contained in a data element are called **data attributes**. A data attribute contains a single value from an application. In order to allow for the various types of data that an application may produce, a data attribute supports a number of data types.

Applications may produce strings, numbers, letters, or almost anything imaginable. In order to keep the data types to a practical size, a subset of types was chosen for use in the system. These were boolean, numeric, and string. A boolean type may be either TRUE or FALSE. A numeric type may be a decimal or integer number. A string type is a sequence of characters. These data types can be used to represent most data produced by applications. In addition to a type, a data attribute has a name and a value. The name allows the attribute to be referred to directly, which is useful when building larger structures.

An example of a data element and its associated data attributes comes from TENNIS. A comment from an agent may contain attributes such as the agent name, the point of view, the network component in question, the text of the comment, and a timestamp. The timestamp is a numeric attribute, and the others are textual.

Data elements as read from the application contain no additional descriptive information. All descriptions of the data elements come from the script. Allowing the application to provide data descriptions makes it more difficult to modify the automatically generated interface via the script.

A data element read from an application contains only names, types and values. In order to create a display of the data elements, much more information is required about the data. COURT must have a description of the data elements and attributes themselves, as well as knowledge of how they can be combined and how they relate to each other. In order to remain application independent, information about the data should be provided externally [Roth & Mattis 94].

Information about the data elements and attributes provides a local view of the data. This may include semantic information, and information about the ranges or types of data contained in an element. This information is provided in the form of **data characteristics**.

Information about the groupings of a set of elements provides a global view of the data. This can be groupings of data elements that are related by content or by interaction. COURT represents this information using abstractions and relations.

Characteristics, abstractions and relations are examined in more detail below.

4.2.1 Data Characteristics

Data elements by themselves provide little or no semantic information. In order to build an informative display of these elements, more information is required. The system must be able to determine what a particular data element represents. Information about the meaning or the types of values can be obtained either externally, or by examining the data. The types of information that are required are broken into semantic information, which provide indications as to the meaning of individual data attributes, and information about a set of data, such as the number and types of values present.

4.2.2 Properties

Semantic information that needs to be expressed may include the domain of the data and the interpretation. A numeric value may represent a time, but this could be either an absolute time, or a span of time. The data semantics can be expressed using properties. **Properties** are markers that can be attached to data attributes to indicate their meaning. Properties available in COURT are *Position*, *Difficulty*, *Size*, *Time*, *Mass*, *Status*, *Label*, *Order*, *Temperature*, *Age* and *Type*. Each of these properties is used later when determining how to encode a data attribute.

The properties in COURT are system defined. A user supplied data description cannot specify properties that the system does not already understand since these properties are directly used by the design rules.

4.2.3 Statistical Information

Aside from semantic information, much can be gathered about a set of data by examining it statistically. A set of data has minimum, maximum and average values, as well as a set of unique values present in the data, called an enumeration. An **enumeration** is a list of the unique values for a particular attribute in a set of data elements.

For example, a set of message elements from TENNIS may each have a text string to represent the agent from which they originated. Though there can be hundreds of messages, they may only come from five or six agents. The enumeration of the agent attribute for this data set would reveal this low number of unique values. Rather than displaying this agent string for each message, it may be more efficient to only display it once for each set of messages.

4.2.4 Data Range

One important piece of information that can’t be determined statistically is the range within which a data value may fall. Different element types within a set of application data may have different attributes, though some of these attributes may lie within the same scale. For instance, TENNIS may provide a set of components each with a position coordinate, and a set of connections, each with a pair of coordinates. Though the data attributes with the two kinds of elements may be entirely different, the values can actually be displayed together since they fall in the same range.

This information is provided to COURT by means of a range specifier. A **range** is an identifier that is attached to a data attribute. It can be reused for data attributes with the same range.

4.2.5 Data Groupings

Data characteristics provide information about the individual data attributes. Characteristics is useful for determining how these attributes are to be displayed. However, characteristics do not provide information on how to group data elements. **Grouping** data elements is the combination of elements for the purpose of providing information about them. The grouping of elements allows the system to determine layout. By specifying that certain data elements are similar and belong together, COURT determines how the data elements are to be related in the display.

It makes sense to distinguish two ways of combining data. One way is to combine data based on similarity. This is a grouping of elements that are related in some aspect, such as their function, style, origin, etc. TENNIS creates messages that can all be grouped together, along with components which can all be grouped.

Another type of grouping is one concerned with interaction. Two elements can be grouped together if they are somehow engaged in a relationship. The elements may be different, but may have some relation. For example, the messages elements that TENNIS can create may refer to certain component elements.

These two data groupings are represented by abstractions and relations.

4.2.6 Data Abstraction

An **abstraction** is a grouping of elements that are similar in some aspect. It represents a strong grouping of data elements. The concept of abstraction involves a generalization of a description. An abstraction of a concept encompasses other concepts which can be grouped into the same general category. An abstraction combines elements that all contain a common ingredient. In COURT an abstraction is a grouping of elements that can be placed together in a generalization. The grouping represents a set of elements that are somehow similar.

For example, all the messages created by TENNIS can be grouped into an abstraction. Messages could also be placed into smaller abstractions by grouping them by their agent of origin, though this would be a lower level abstraction, since each abstraction would include only a subset of messages. These smaller abstractions represent slightly less general abstractions than an abstraction of all messages. However, further abstracting all these smaller abstractions into one large abstraction would result in all the messages being grouped together.

The primary use of an abstraction in COURT is to guide the layout of the display. Abstractions represent a logical and conceptual grouping of data, which is best reflected in the display through the use of location.

4.2.7 Data Relationships

A **relation** is a grouping of elements that engage in some interaction. It represents a weaker grouping than an abstraction. In addition to grouping elements, relations have the an additional parameter which specifies the type of the relationship. This is equivalent to a data attribute property.

In TENNIS, a relation may exist between messages and components. Messages can *refer to* components. This relationship is not appropriate for an abstraction, since these elements are not similar. A relation represents this connection between elements.

The relation represents a weaker binding of data, and, as such, it will not affect the display as significantly as an abstraction. Relation information will primarily affect interface attributes. The relation between two elements will be shown through the consistent use of attributes such as color and texture.

4.3 Presentation Description Language

COURT provide a language in which the interface designer can express the data descriptions. This language is named VOLLEY. The data description contains all of the information required to create an interface from an application's data. The description must include the application data elements, their characteristics, and the groupings that can be formed with them. The language must meet the following requirements:

- *Expressible by an interface designer.* The language should be easy to learn, read and write. Because changes may be frequent during application development, and the description may have to change for alternate user types, the language must be easily learned by an interface designer.
- *Describe data elements and characteristics.* Application data elements along with their attributes and characteristics must be described using the language. The characteristics are the properties and the ranges.
- *Describe data abstractions and relations.* Abstractions and relations of data elements must be described using the language. The contents of the each type of grouping must be described.
- *Easily machine processable.* COURT must parse the language into an internal representation, so it must be machine readable and well structured for parsing.

In order to make the language easily read and written by an interface designer, it was designed to use as many english words as possible, and contain minimal punctuation. To make the language machine processable it was structured to be expressible as a context free grammar. Examples of data descriptions written in VOLLEY appear in the descriptions below, as well as in Chapter 6. The VOLLEY grammar is expressed in BNF in Appendix A.

Aside from being machine and human readable, the language must have the ability to express the data elements, characteristics, abstractions and relations. These four items form a natural break in the information the language must express. The VOLLEY language consists of four declarations that express each of these descriptions. They are the data element description, the data element characteristics, the abstraction description and the relation description. These declarations are described in more detail below.

4.3.1 Data Element Classes

Data elements are the basic units of information provided by an application. An application can produce many kinds of data elements, but by default there is no identifying marker associated with them. The only difference between the kinds of data elements that an application can generate are the types of data attributes they contain. In order to refer to a particular kind of data element it is required that they have a marker attached to them. In order to specify this, the VOLLEY language provides the ability to specify a data element class. A **data element class** provides information about the kinds of data attributes a certain style of data element has. When data is received from the application, it is matched against the data element classes described. A data element class specifies the kinds of data attributes it will contain. These data attribute specifications are known as **data attribute classes**. A data attribute class specifies a name and a data type.

The VOLLEY declaration of a data element class may look like the following:

```
element Component has
     string Name, Function
     numeric X, Y
     numeric Health
end
```

The above declaration specifies a data element class named `Component`. This class contains two string attributes named `Name` and `Type`. It also contains three numeric values named `X`, `Y` and `Health`.

The use of the data element classes is described further in the description of the Recognizer.

4.3.2 Data Element Characteristics

Data element characteristics provide information on the semantics of individual data elements. The characteristics maintained by COURT are the properties and the range information. These characteristics both describe data attributes. In order to express characteristics for a data attribute, the VOLLEY language must allow the specification of a data element class, a data attribute class, and either a property or a range. To provide this, VOLLEY has a `characteristics` declaration. To aid in writing and parsing the language, the `characteristics` declaration specifies a single data element class within which many data attributes can have characteristics specified.

An example of a VOLLEY declaration of data element characteristics is as follows:

```
characteristics of Component
     Name properties label
     Function properties type
     X, Y properties position
     X range netmapX
     Y range netmapY
     Health properties status
end
```

In this example, the `Name` attribute has a `label` property attached to it. `Function` determines a `type` of `Component`, and `X` and `Y` specify `position`. The `X` and `Y` coordinates each have a range specifier, namely `netmapX` and `netmapY`. These range specifiers can be arbitrary identifiers. The importance of range specifiers comes when they are repeated in the characteristics of other attributes. They are used to link together values in the same range. Other data elements such as connections may contain attributes whose range is also netmapX and netmapY. This tells COURT that although these data attributes are from different element, they contain values that can be displayed together, for instance on the same axis.

Properties are used mainly by the rough and detailed design components, to be described later.

4.3.3 Data Abstraction Classes

Data element classes and characteristics describe the application data in a local manner. The abstraction and relation classes are used to group elements to provide a more global view of the data. As described previously, a data abstraction consists of a set of data elements. In order to specify these abstractions to COURT, it is useful to create an abstraction class. An **abstraction class** specifies a kind of abstraction that can be created.

The data elements contained in an abstraction class can be varied. Some abstractions may contain only one element from a class, whereas others may contain all the elements of a class. In addition, it is useful to allow constraints on elements included in a class. To provide this flexibility, the abstraction class declaration is broken into two parts, the references and the constraints.

The **references** part of an abstraction class declaration specifies a list data element class names. Data elements from each class listed can be included in one of two ways. A set of abstractions can be created, each of which contains one element from the element class, or a single abstraction can be formed which includes all elements from a class. This is known as the **quantity** of the elements included in an abstraction. Each data element included can be named in the references section to allow the constraints to refer to them later.

The **constraints** part of an abstraction class declaration specifies a limitation on which elements are included in an abstraction. The constraints are in the form of conditional expressions. A constraint specifies a data attribute of one of the element classes listed in the references section. The attribute can be compared to literals, or to attributes in other element classes listed in the references.

This system of references and constraints can be confusing, but it provides significant power in grouping elements. In order to clarify the operation of the references and constraints, a series of examples will be shown.

The first example shows a grouping of all elements from one class, `Component` (defined above), into an abstraction.

```
abstraction NetworkComponents has
     all Component
end
```

This specifies that all elements from the `Component` class be abstracted into a `NetworkComponents` group. The second example shows the specification of a single element in an abstraction class, and the use of constraints in a class.

```
element Computer has
     string Name
     string ServerName
end

abstraction Client has
     one Computer
     where
     ServerName of Computer <> “”
end
```

Because the quantity of `Computer` elements in the `Client` abstraction class is one, this declaration specifies that a new abstraction instance is to be created for each instance of the `Computer` data element class whose `ServerName` attribute is not empty. The third example shows the use of naming elements in a reference, as well as constraints against other data elements referenced.

```
abstraction DependentHosts has
     all Client Clnt
     one Computer Srv
     where
     ServerName of Clnt = Name of Srv
end
```

The `DependentHosts` abstraction class creates a set of abstraction instances which each contain a set of `Client` elements and a single `Computer` element. The element classes specified in the references section are named (`Clnt`, `Srv`) so that they can be used in the constraints section. The abstraction constraint specifies that all `Clnt` elements (of class `Client`) contain the name of the `Srv` (of class `Computer`) element in their `Server-`

`Name` string attribute. This abstraction is designed to combine together a computer which is a server, and all of its clients. As can be seen from the example, the references can also specify abstractions as well as data elements. This allows for the creation of a data tree.

The use of data abstractions is discussed further in the section describing rough interface design.

4.3.4 Data Relation Classes

Data relations are much like abstractions, only they refer to a maximum of two elements. A **data relation class** specifies the data elements involved in a relation, along with the relation type. The requirements for a relation class declaration, like that of the abstraction class, dictate that there be a reference and a constraint section. The references can only specify two data element classes. Because there are fewer classes involved in a relation than in an abstraction, there is no need to name the classes. Therefore the class name is the only name used to refer to the classes referenced in a relation, as opposed to the reference class naming provided in the abstraction. The syntax and semantics of the constraints section of the declaration are identical to the constraints section of the abstraction class.

The following example shows the use of a relation.

```
relation
     Message refersto Component
     where
     CompName of Message = Name of Component
end
```

In this example it is assumed that the `Message` class contains a `CompName` attribute and the `Component` class contains a `Name` attribute. A relation instance is created for each `Message` and `Component` pair in which the `Message`'s `CompName` attribute is the same as the `Component`'s `Name` attribute.

4.3.5 VOLLEY Script

The data description must be stored externally to the system and used when reading and classifying application data. The description must be persistent across runs of the system, and must be editable by the interface designer. To provide these capabilities, the description is stored in a **script** which is provided to the system just before the application data is read. The use of a script allows alternative descriptions of the data to be easily substituted. This enables different users of COURT to easily provide different descriptions of the data.

4.4 Parser

The data description must be read by the system for use while processing the application data. The script parser is used to read the script and store a representation of the classes described in it. The data element classes and the data characteristic declarations must be stored and provided to the recognizer when it runs. The abstraction and relation classes must also be stored and provided to the augmentor when it runs.

4.5 Application

The application must provide data in the form that COURT can understand, namely as a set of data elements. Any large structures that the application produces must be broken down into data elements. The application must be able to convert all its data into the basic data types supported by the data element structure. COURT mainly operates on sets of data and as such, applications that produce data in sets are best suited to work with COURT. Once data elements are created from the application data, they are output to the recognizer for further processing. The application interface is the only part of the COURT system that is specific to a particular application.

4.6 Recognizer

The application data elements are provided to the system with no semantic information. Before any abstractions or relations can be generated, the data elements must be recognized as belonging to a particular class. In addition, the data characteristics must be appended to each data element. This is the job of the recognizer.

The recognizer uses the internal representation of the script that was supplied by the parser. The recognizer has two primary duties: determining the class of a data element and adding the data characteristics to a recognized element.

Determining the class of an element is done by simply looking for a match of the element among all the defined element classes.

Adding the data characteristics is also reasonably straightforward. The data characteristics are specified for a particular attribute within a data element. When an element is recognized, the data characteristic declarations that match the element class are retrieved. The data characteristics are then appended to the appropriate attributes within the recognized element.

After application data elements have been recognized they are ready to be sent to the augmentor. The elements are stored in a holding area for pickup by the augmentor.

4.7 Augmentor

The augmentor processes application data to apply the abstraction and relation classes contained in the script to the data it has read. The input to the augmentor consists of the recognized application data, along with the abstraction and relation classes declared in the script. The augmentor consists of two parts, the abstractor and the relator. The abstractor adds abstraction classes, while the relator adds relation classes. The output of the augmentor consists of data objects grouped into the abstractions and relations. This structured set of application data is known as the **augmented data objects**.

4.7.1 Abstractor

The abstractor builds abstractions based on abstraction classes defined in the script. An abstraction is a group of data objects. Data objects are either elements or abstractions. An abstraction class specifies the classes of objects that are contained in the abstraction, and a set of constraints that must be satisfied by the objects in order to be included in the abstraction.

The available abstraction classes are retrieved from the information provided by the parser. For each abstraction class defined in the script and processed by the parser, the abstractor attempts to build a set of abstraction instances. These instances must contain the elements referred to by the abstraction class, but must also satisfy the constraints placed on the instances by the abstraction class.

Because the abstraction classes may specify that an abstraction contains other abstractions, the resulting abstraction instances may be formed into a tree structure. This tree would contain data elements as its leaves and data abstractions as its nodes.

After the abstraction classes have been created they are sent to the relator for the addition of relation groupings.

4.7.2 Relator

The relator must instantiate the relation classes defined in the script. As for an abstraction, the relation classes specify references to data object classes, which can be abstraction classes or data element classes.

The relator must take the set of relation classes read by the parser and build instances of these classes from the data provided by the abstractor. These relations are added to the data tree produced by the abstractor.

Figure 12 shows a sample data tree after it has been augmented by the abstractor and rela-

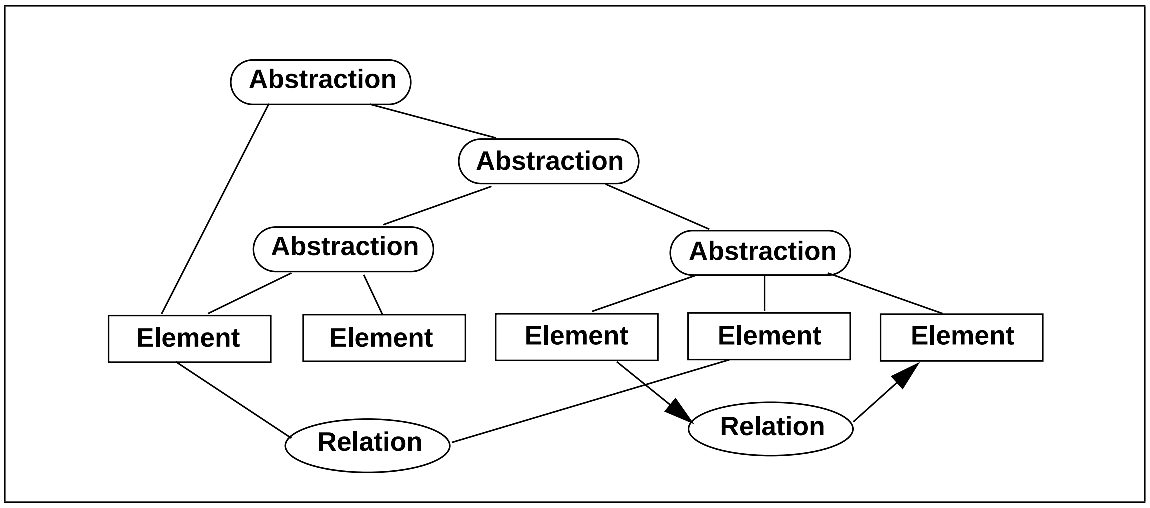


Figure 12: Augmented Data Tree

tor. The abstraction and element instances are shown here as a tree structure. The relations are shown as additional links among nodes in the tree. Figure 13 shows an augmented data object tree that may be generated after TENNIS data is abstracted and related. In both figures, the rectangles represent data elements, the rounded rectangles are abstractions, and the ovals are relations.

4.8 Interface Generation

The interface generation portion of the COURT system must take the augmented data objects from the augmentor and create an interface to display these objects. The major actions

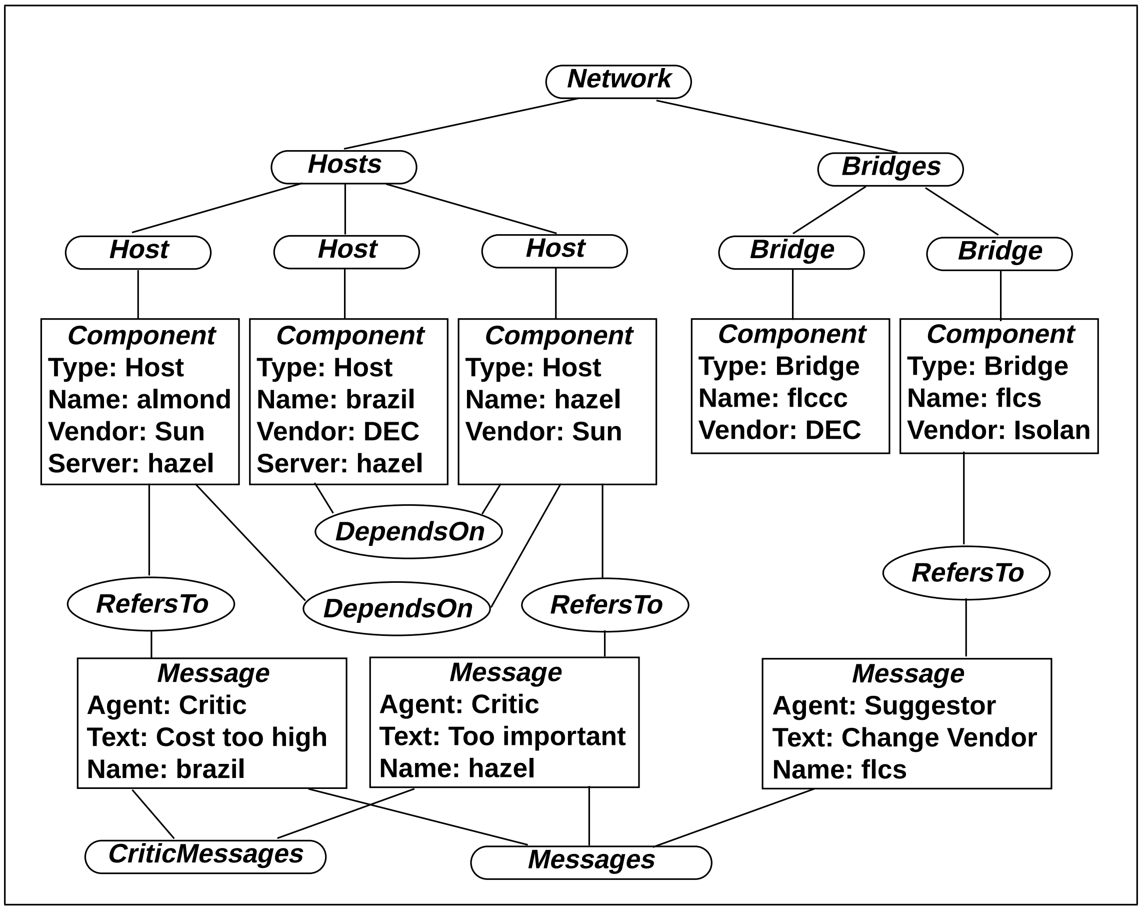


Figure 13: Augmented Data Objects from TENNIS

performed when generating the interface are partitioning the data, laying out the data within windows, and creating the low level widgets to display the data. [Kim & Foley 90] More specifically, the requirements of the interface generation are:

- *Examining the data to find patterns.* The data must be examined to determine additional characteristics that were not provided in the script. This can include information about the scale of the data and the number of unique values in a data set.
- *Partitioning the data into windows.* The data elements must be partitioned into separate windows. Presenting all the data within a single window is not effective for most applications. The data should be broken logically into separate windows.

- *Laying out the windows.* The windows that contain the data must be laid out relative to one another in a logical manner. This is accomplished by positioning and sizing the windows in the display.
- *Laying out the data within the windows.* Within a window the widgets created must be laid out to represent the data and the structure and relations in the data.
- *Determining how to represent data within windows.* The rule system must also determine which widgets should be used to represent particular kinds of application data depending on its characteristics.

The first requirement is concerned with analyzing the data to be displayed. The second and third requirements are concerned with producing the display on a higher level. The major concern is the content and layout of windows, which constitutes a rough design. The last two requirements are concerned with the layout and representation of data within windows, which is detailed design.

The design requirements can be examined in terms of "heuristic classification" as described by Clancy [Jackson 90]. Figure 14 shows the data analysis, rough design and detailed de-

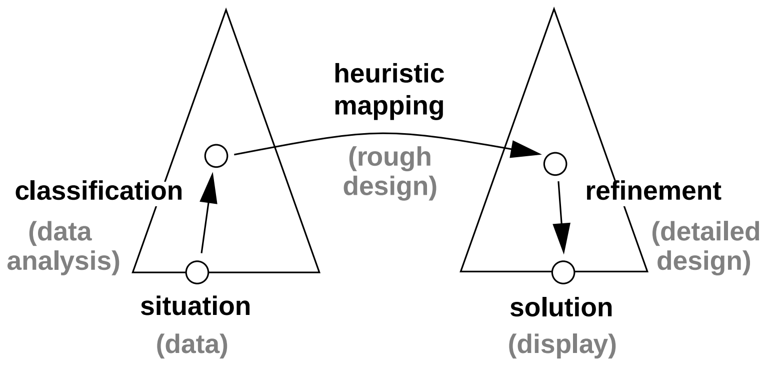


Figure 14: Rough and Detailed Design Seen as Classification

sign actions as they relate to heuristic classification.

Heuristic classification describes a process in which situations are first generalized to find abstract descriptions of the situations. From this, a heuristic mapping is made into an abstract solution. Details of the solution are determined by applying details of the situation. Heuristic classification is useful when a direct mapping from situation to solution is not apparent.

The data analysis stage is analogous to classification because it involves determining characteristics of the data. Abstraction, relations and characteristics help to classify a set of data. Once a set of data has been classified it is mapped to a partial solution through the use of rough design heuristics. The partial solution represents an abstract interface description which can be refined through the application of detailed design rules.

Many other UIMS have demonstrated that the use of rules is effective in automatically creating an interface [Arens et al 91], [Kim & Foley 90], [Myers et al 94], [Szekely 90]. The DON system described in [Kim & Foley 90] splits the interface design into rough and detailed design. The grouping of the requirements for COURT's interface design suggests this approach is appropriate. The rough design stage determines the grouping and general layout of data elements within windows. The detailed design stage determines the exact interface objects that will be used to represent the data elements.

The result of the mapping from data to interface must be a description that is sufficiently detailed that the interface can be produced in a windowing system. The output language of the mapping does not need to be a widget tree, but it has to be sufficiently detailed that it can be represented in a windowing system without requiring additional knowledge in the form of rules. The output of the interface generation in COURT is a set of interface objects.

Interface objects represent actual widgets or windows that can be created in the windowing system. Interface objects contain **interface attributes** which determine the exact appearance of the widgets. The use of interface objects simplifies the interface model and allows the implementation of the interface to be modified without affecting its design. The interface objects will be discussed in more detail in a later section.

The rough design and detailed design stages will now be examined more closely.

4.9 Rough Design

The rough design phase involves creating a layout for the data elements. This phase is largely done by examining the data abstractions and creating the proper windows to reflect the groupings of data elements. An analysis of the rough design phase is done by examining the kinds of interface specifications it can generate, the issues involved in generating the these specifications, and a structure for applying rules to the data to produce the specifications.

4.9.1 Constraining Rough Design

Since rough design involves determining the layout and content of higher level interface objects such as windows, a few simplifying assumptions about these interface objects greatly simplifies the job of the rough design stage. Simplifications made involve the kinds of windows that can be generated and the layouts of these windows.

4.9.1.1 Window Nesting Constraints

Placing one or more windows containing elements inside another window is known as nesting the windows. Nested windows effectively represent a grouping. Conceptually or logically related windows can be nested within the same window to show their similarity. Allowing the windows to nest provides flexibility in design and representation. However, allowing unlimited levels of nesting poses problems when laying out the windows and does not help to constrain the possible designs. Therefore it is sensible to limit the nesting level of windows.

In COURT, windows are limited to two nesting levels. This limitation means there may be any number of first level windows within the display, and within each of these there may be any number of second level windows. Within the second level windows are data elements. The data elements are laid out during the detailed design, but no further partitioning of the display can expressed within a second level window. The first level windows are known as **top level windows**. The second level windows are known as **abstraction windows**.

4.9.1.2 Window Layout Constraints

To further constrain the rough design stage, the abstraction window layouts are limited. An easy limitation on abstraction window layout involves the meaning of element positions along the window axes. By defining how the window axes may be used, the rough design stage more easily determines what kind of axis to use for a set of data. COURT allows three kinds of axis uses. These are position or order information, alignment of elements, or grouping of elements.

The first use of an axis is for position or ordering. Using an axis to display position or ordering information involves determining the location of an element using a single data value from the element. Position differs from ordering in that position specifies an absolute location of an element with respect to an axis, where ordering specifies that an element be located in a window relative to other elements.

The second use of an axis is for alignment. Using an axis to align elements involves locating all aligned elements at the same position relative to an axis. This allows the elements to be lined up.

The third use of an axis is simply to group elements. Elements placed along the axis indicate they are grouped, but their absolute or relative position is not meaningful beyond the fact that they are logically related.

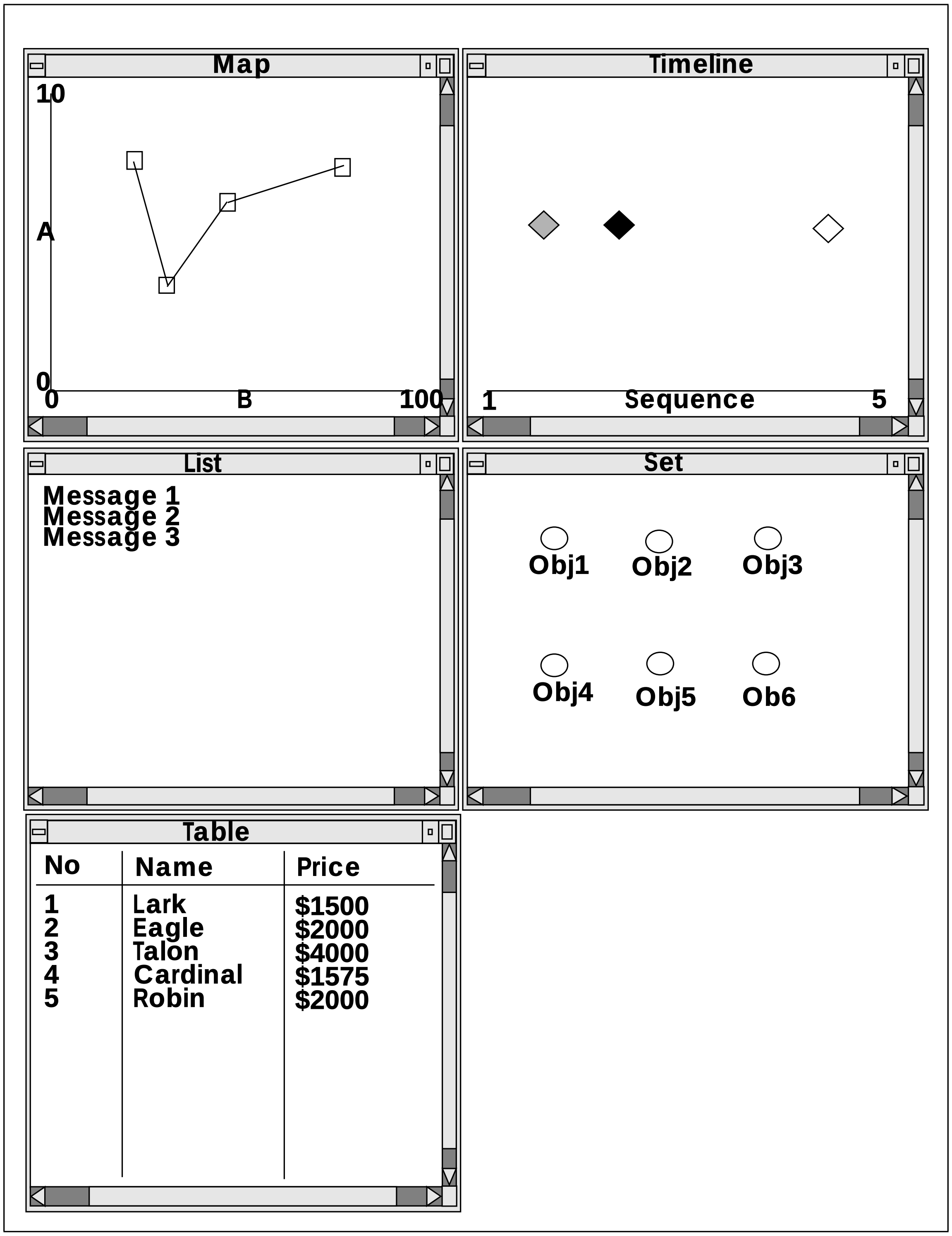
Some examples of the kinds of abstraction window layouts possible using these axis constraints are shown in Figure 15. These windows do not define distinct abstraction window classes, but rather they are types of windows that can be generated. The Map windows uses position on both axes and provides location information in two dimensions. The Timeline provides position information on one axis and grouping on another. The List window displays elements in order on one axis and uses alignment on another. The Set window uses both axes to group the elements together. The Table window uses an ordering in both axes.

The distinction between position and ordering can be seen in Figure 15. The Timeline and Map windows use position to indicate that the absolute location of elements is significant. The List window however uses ordering to show that the relative location is important. The exact values used to locate the elements are not important, only the order of the elements are.

4.9.2 Rough Design Issues

The rough design issues are simplified by the constraints described above. The issues involved in doing a rough layout are:

- *Determining what top level windows must exist.* Top level windows may contain multiple abstractions in order to save space, but they should contain abstractions that are



Message 1
Message 2
Message 3

No	Name	Price
1	Lark	$1500
2	Eagle	$2000
3	Talon	$4000
4	Cardinal	$1575
5	Robin	$2000

Figure 15: Types of Abstraction Windows

similar.

- *Determining what data abstractions can be combined into a single abstraction window.* A window cannot be generated for each abstraction, but combining two abstractions means they must present potentially different data in the same window using the same scales.
- *Determining the layout of abstraction windows.* This involves looking into the data to determine what similarities may exist in the data, and what is the most important information contained in it.
- *Providing a way for the relationships to be represented.* Space within the interface objects attributes must be reserved to display relations.

To resolve these issues, the rough design process uses the data descriptions. The abstractions, relations and characteristics all provide information that enable the rough design process to be completed. These descriptions are examined with regard to rough design.

4.9.2.1 Rough Design Based on Abstractions

Abstractions provide the most important layout information. Because an abstraction represents a strong grouping of data objects, it makes sense to place objects within an abstraction in the same window.

Abstractions at the top level of the augmented data object tree are call top level abstractions. Typically there are relatively few top level abstractions and they incorporate many other lower level abstractions. Top level abstractions represent a conceptually broad grouping of the lower level abstractions. As such, they are good candidates for conversion to top level windows. A top level window represents a broad conceptual grouping of the windows and elements within it.

Abstractions below the top level abstractions represent more specialized groupings. These abstractions are mapped to abstraction windows. An abstraction window represents a sub-grouping of the data elements within a top level window. The layout of the abstraction windows within the top level windows are determined by examining how the abstractions were grouped into the top level abstraction.

Abstractions at lower levels can either be merged to form a larger abstraction, or the elements within these abstractions can be distinguished from each other through the use of data attributes.

4.9.2.2 Rough Design Based on Relations

Relations represent secondary layout information. Because a relation is not as strong a grouping as an abstraction, the relations are not as influential in the rough design. Relations can be used to determine additional layout constraints on abstraction windows or top level windows, or can determine additional interface object attribute requirements.

Relations that exist between elements or abstractions within top level windows or abstraction windows may indicate these windows should be located close to each other. Though no similarity is expressed between the objects within these windows, the quantity of relations between them may dictate that they be located near each other to show their relations. This constraint on layout can be at the level of top level windows or abstraction windows.

Relations can also be expressed through the consistent use of interface object attributes. Though related data elements may not be located close to one another, if they both have the same color or shape, their relation is apparent.

4.9.2.3 Rough Design Based on Data Characteristics

The data characteristics can be as important as the abstractions when determining the content or layout of windows. Data characteristics describe the semantics and the types of values contained in a set of data elements. Characteristics can be in the form of properties which provide hints about the meaning of values in elements, or statistical information such as the range and number of unique values in a set. These characteristics affect the content, layout and types of windows created.

The characteristics affect the content since semantically-related elements should be placed together within a window. In addition, the elements should have the same data ranges. Elements that have different meaning or scales are better shown in separate windows. Characteristics affect the content of top level windows as well as abstraction windows.

Layout is affected through the characteristics as well. Abstraction windows or top level windows containing semantically related data should be located near each other. For example, three abstraction windows named A, B and C may be generated to contain data. There may be two pairs of windows (A and B, B and C) that are semantically similar, but the third pair may (A and C) be not be related. This suggests that window B, which is similar to both A and C, be laid out so both A and C are near it. There is no constraint that A and C be located in proximity to each other, since they are not related.

Window style is also affected by characteristics through modification of the style to reflect the semantics or types of data contained in a window. Windows containing more important elements should be located more prominently on the screen, or contain other visual cues to indicate their importance.

The type of abstraction windows used to contain data are also affected by the data characteristics. The number and type of data attributes determines what the most important data attributes are, and what data attributes should be displayed using location information.

4.9.3 Structuring of Rough Design

The input to the rough design stage is the augmented data object tree. The output of the rough design is the specification of top level and abstraction windows to be used to display data. This specification includes the contents of the windows and the constraints on the layout of both window types. In order to produce this specification, the following actions must be performed:

- *Statistical analysis of data.* A statistical analysis of the data provides additional characteristics that could not be specified through the script.
- *Partitioning of data tree into top level windows.* The top level windows must be determined, along with their content and layout.
- *Partitioning of top level windows into abstraction windows.* The content of the top level windows must be partitioned into abstraction windows. The layout of abstraction windows must also be determined.
- *Determining axis use in abstraction windows.* The meaning of location within abstraction windows, along with the attributes that map to location must be decided.
- *Creation of interface requirements for relations.* Interface requirements must be generated that provide for the relations to be expressed.

These steps are examined in more detail below.

4.9.3.1 Statistical Analysis of Data

The statistical analysis is used to determine additional characteristics of the data elements. This analysis includes characteristics pertaining to a set of data such as the minimum, maximum and average values, and the number of unique values in a set of attributes. This analysis must be performed at all levels with the augmented data tree, so that in further steps the contents of abstractions at any level can be determined.

The minimum and maximum values are useful in determining the scaling of values when performing a detailed design.

One value of particular use in further steps is the number of unique values in a data set. Knowing the number of values an attribute has may affect its display. For example, consider a large set of data elements representing computers. Each element contains three numeric attributes named X, Y and Status. The X and Y values may be different for each element, however, only three values may ever be specified for Status. Rather than using an interface attribute that can represent a continuous value for the Status, the system may chose an interface attribute such as shape, which can have a only a few possibilities. This approach provides a more effective display.

If a set of data elements has only one value for a particular attribute, the value should be represented only once for all the elements contained in the set, rather than displaying the value for each element in the set.

4.9.3.2 Partitioning Data to Top Level Windows

The data in the augmented data tree must be partitioned into top level windows. The system normally does this partitioning by creating a top level window for each top level abstraction. If there are too many top level abstractions to practically create one top level window for each, COURT should examine the top level abstractions to determine the abstractions that are most closely semantically related and combine them into a single abstraction. Abstractions that are not semantically related but contain values in a common range may also be combined. For example, a set of network components and a set of network connections may not have the same semantics, but they both may have coordinate data pertaining to their location in the network. If these coordinates are in the same range, the elements can be displayed together.

The layout of the top level windows must also be determined. This layout depends largely on the properties of the data in each top level window. Semantically related top level windows can be placed near each other. Windows that contain relations between data in them may be positioned together. Windows that are more important or represent more urgent data can be placed above other windows.

The creation and layout of the top level windows is performed using a set of rules. The rules match on patterns of top level abstractions and combine them as needed, or create top level windows for them. The rules also match on characteristics of the data to determine layout constraints on the top level windows.

4.9.3.3 Partitioning Top Level Windows to Abstraction Windows

Data within a top level window must be partitioned. This is done much as the data is partitioned into top level windows. If too many abstractions exist to create an abstraction window for each one, the abstractions can be combined.

Layout must also be determined, which follows the same general process as with the top level windows. Some differences exist with abstraction windows however. Because the abstraction windows are contained within the top level windows, they cannot be overlapping. Abstraction windows are also more constrained in their layout than top level windows. The most effective manner for laying out the abstraction windows is to place them next to each other aligned either horizontally or vertically. To make this layout more visually appealing, the adjacent edges should be made the same length.

As with top level windows, rules determine the abstractions that should be combined as needed, and to determine what abstraction windows should be created. Rules can also match on characteristics of data to provide constraints on the layout. These constraints should position together abstraction windows containing semantically related data, or windows with relations between objects contained in them.

4.9.3.4 Determining Use of Axes in Abstraction Windows

The data within an abstraction window must be laid out so as to show the most important aspects in the most effective manner. Each abstraction window has two axes which can be used to encode data attributes as location. The style of the abstraction window is most affected by the encoding method used in the axes.

A set of rules determines which data attributes will most effectively be mapped to the location, and what the location means. These rules work by examining the ranges of attribute values in the data and determining the values that are in the same range across all the data attributes. The rules determine the attributes that can be mapped to an axis using one of the encodings available.

4.9.3.5 Creation of Interface Requirements for Relations

Relations between elements in different windows should be displayed through the consistent use of an interface object. For example, message elements that all refer to a set of component elements may be grouped into a separate window. The relations between the two elements may be shown by coloring the components and the messages that refer to them the same color.

The interface attributes used to show data attributes are determined during the detailed design phase, but the rough design phase needs to alert the detailed design phase to the requirement that the relations be shown. This is done by attaching these requirements to the elements.

Rules are used to provide the requirements. The rules recognize elements grouped in a relation and attach additional interface requirements to them.

4.10 Detailed Design

The detailed design phase completes the creation of an interface that was started by the rough design. The rough design produces windows to contain data and layout constraints between the windows. The detailed design phase must complete the interface design by solving the layout constraints and mapping the data attributes in the augmented data object tree into the partial interface. More specifically, the functions of the detailed design phase are:

- *Solve layout constraints for windows.* Both top level windows and abstraction windows must be positioned and sized so as to satisfy the layout constraints determined in the rough design.
- *Create interface objects for windows.* Both top level windows and abstraction windows must have corresponding interface objects created for them.
- *Create interface objects for data elements.* Data elements within abstraction windows must have interface objects created for them.
- *Clean up design.* The design must be polished as much as possible. This includes neatening the display, and providing labels and legends as needed.

These functions are described in greater detail below.

4.10.1 Solve Layout Constraints

The rough design provides a set of top level windows and abstraction windows with layout constraints. These layout constraints describe the relative locations of windows in the display. The detailed design must determine the actual placement of the windows in the final interface. The layout of both top level windows and abstraction windows must be determined.

Top level windows have the ability to overlap one another, which aids in satisfying the constraints. The fewer the constraints there are, the more easily they can be satisfied. Abstraction windows have a limited number of ways in which they can be placed, which makes it slightly more difficult to satisfy all constraints.

The constraints specify that particular windows should be located close to each other. Because the windows each have four sides, there are a number of ways in which two windows can be positioned and be close to each other. This suggests that a backtracking algorithm would be effective for satisfying the constraints. A decision on a window's location made to satisfy one constraint will have to be changed if it does not satisfy all other constraints as well. The system must undo all decisions made after that decision, modify the decision to use another side of the window, and continue the process. The ability to undo decisions up to a point, change a decision, and continue on is known as backtracking.

4.10.2 Create Interface Objects for Windows

An interface object is created for each window specified in the rough design. The window should be labelled with a description of the data it contains. Interface objects that correspond to top level windows and abstraction windows exist, so the mapping does not require knowledge. Each window specification provided by the rough design simply has to be transformed into an interface object.

4.10.3 Create Interface Objects for Data Elements

Abstraction windows contain a set of data elements and specify an encoding mechanism for each axis. The detailed design phase determines which interface objects to create to represent the data elements and their attributes. A number of issues must be dealt with at this stage:

- Data elements and their attributes need to be mapped to interface objects.
- Certain data attributes may have already been mapped to axes in the window.
- Some data attributes may have the same value in all elements and can be displayed at a common area in the abstraction window, such as the title or legend.
- Additional interface attributes may be required by the rough design to display relations.
- Legends to explain the mapping of the attributes must be created.

In order to map the data attributes into interface attributes while keeping track of the above concerns, a two part process is used. The first part maps data attributes to interface attributes. This mapping is done using rules that examine the data characteristics and determine the best representation for a value. The second part picks interface objects that contain the interface attributes required for display. This mapping attempts to produce the most efficient display possible by picking interface objects that contain as many of the required interface attributes as possible.

4.10.3.1 Mapping of Data Attributes to Interface Attributes

By first determining the interface attributes required from the data attributes, the system also maintains the additional interface attribute requirements, as well as ignore the data attributes that have already been mapped. Any data item which is not being displayed in an interface attribute can be flagged as such and simply not mapped. Additional interface attribute requirements provided by the rough design are added to a list of attribute requirements generated from the data.

There are two ways in which data attributes are mapped to interface attributes. These are context free and context sensitive.

Context free attribute mapping is only concerned with the characteristics particular to a single data attribute. The data attribute type, properties and the value contained in the attribute are the important characteristics considered when mapping. Mapping rules match on particular combinations of the data type and properties and specify an interface attribute to use. Figure 16 shows how data properties are mapped to interface attributes in a context free manner.

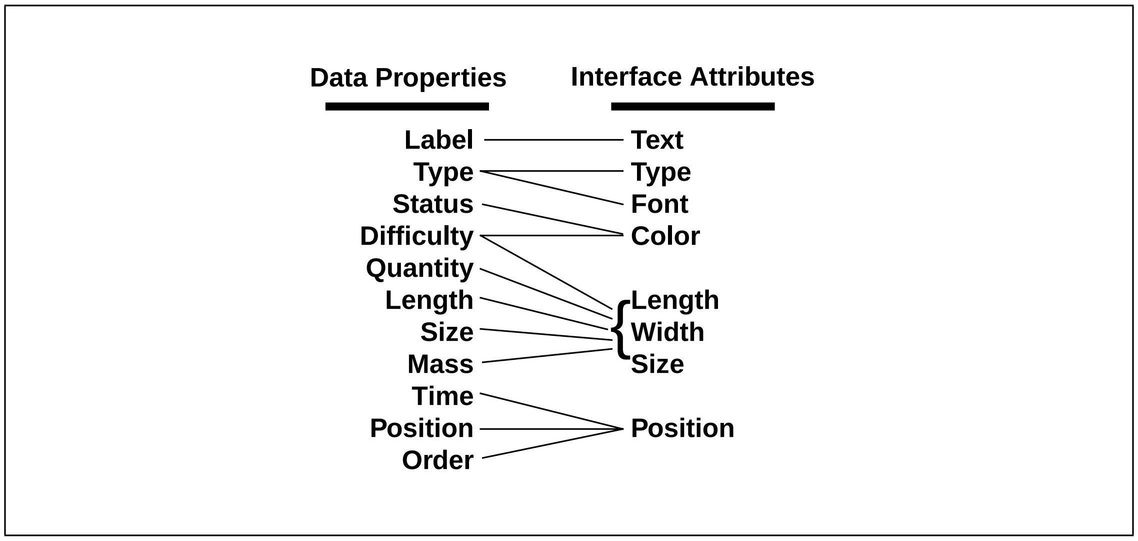


Figure 16: Context Free Data Property to Interface Attribute Mapping

Context sensitive attribute mapping is concerned with the other attributes that are being represented in the same set as a particular data attribute. When mapping, issues such as the minimum and maximum values of the attributes in the set, the number of attributes in the set, and the number of different values contained in these items are all important. For instance, of there are a large number of other values for an attribute in a set of elements, an interface element that can represent a continuous value, such as position, would be appropriate. If only a few values are present for a data attribute, an interface attribute that better shows an enumeration, such as shape, is a more effective choice.

The context free and context sensitive mappings are performed using a single set of rules. The rules match on data attributes with particular types, properties, and number of unique values, and determine an interface attribute to use. This approach is less complex than trying to create two sets of rules that must coordinate activities.

4.10.3.2 Mapping of Interface Attributes to Interface Objects

Once a set of interface attribute requirements has been created, interface objects are chosen to represent those attributes. The interface objects must contain attributes to match those desired, and contain as few additional attributes as possible. For example, if an element requires two position attributes to be displayed, using a line interface object would be unwise because a line contains four position attributes. An interface object with only two position attributes would be less ambiguous.

Rules determine which interface object to use. The rules match on a set of interface attribute requirements and produce a set of interface objects to represent them. The rules incorporate knowledge as to the effectiveness of the interface objects.

4.10.4 Clean up design

Many aspects of the design need to be touched up before it is complete. A number of these are described below.

If all the data elements in an abstraction window contain a single value for a particular data attribute, this attribute does not have to be displayed for each data element. The attribute should be moved to a higher level within the abstraction window, such as the title.

Legends for interface attribute mappings are created to provide information to the user on the meaning of the interface attribute values. This may include information on the ranges of values represented for a continuous value, or the actual values for data with few unique values.

Axes within abstraction windows are labelled. As with legends, this labelling may show the ranges of a continuous variable, or the values of attributes with few unique values. The axis labelling should also include the name of the attribute that is encoded using the axis.

Abstraction windows and top level windows must be titled to indicate their contents. Titling the windows can be done in the border of the window, or using a separate area of the window.

4.11 Interface Objects

Interface objects are the language of the interface design produced by the rough design and detailed design stages. Interface objects provide a simple, windowing system independent means of specifying an interface.

Based on the operation of the rough and detailed design phases, the interface objects must represent the following of interface concepts: Top level windows, abstraction windows, and interface elements to represent the data elements. Each of these is described below.

4.11.1 Top Level Windows

Top level windows represent the most complex interface structure represented in the design. All interface objects created by the mapping process are contained within top level windows. Top level windows can overlap, and must allow for a title bar. All the information contained within a top level window must be viewable by the user, but a top level window may be forced by the windowing system to be too small to show all the information at once. This means the top level window must provide some means of accessing this information, such as scrolling the window.

4.11.2 Abstraction Windows

Abstraction windows are windows within top level windows. Top level windows are partitioned into a set of abstraction windows. The abstraction windows must contain the interface elements. As with top level windows, abstraction windows must contain title bars and allow the user to access information hidden due to size restrictions.

4.11.3 Interface Elements

These are the basic building blocks of the system. Interface elements contain a set of interface attributes. These interface attributes represent the data attributes directly and as such must have the same data types. Specifically, these are numeric, string, and boolean. Types of interface elements are a node object, a line object, and a text object [Mackinlay 91], [Myers et al 94]. Figure 17 shows these objects and their attributes.

The node object is appropriate for displaying a set of attributes requiring up to two positions. The line object expands this to four positions at the expense of loosing the shape among other attributes. The text object is useful for presenting strings that cannot be encoded via other attributes.

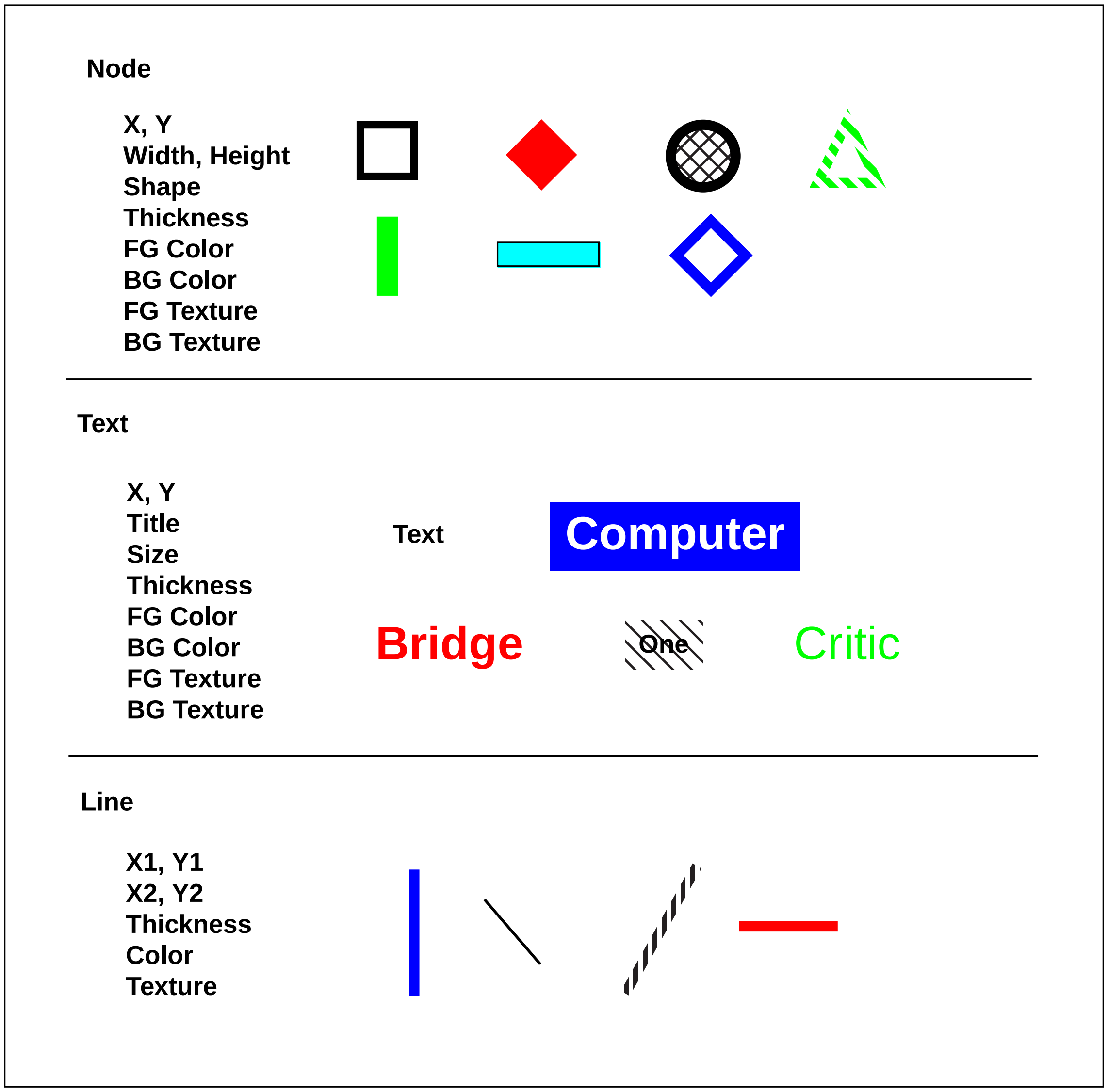


Figure 17: Interface Elements

4.12 Summary

In order to satisfy the goals of the system, COURT is designed as rule based automatic presentation system. Data provided from an application is stored as elements comprised of attributes. The definition of the elements and their properties is contained in the data description. The data description language, named VOLLEY, also allows for the expression of data groupings named abstractions and relations. These groupings represent structure within the data. The abstractions and relations provided in the data description are applied to the data in the augmentor component of COURT. The result is a tree structure representing a more highly structured set of data. This structured data is then processed by the interface generation component. The process of interface generation is split into rough and detailed design. The rough design determines the content and layout of the windows, while the detailed design produces a final interface from the partial description. The result is a complete presentation of the application data ready for viewing. The next chapter describes the implementation of the COURT design.

Chapter 5 System Implementation

In order to verify that the design of COURT was valid, all of the system components were implemented to some degree. Some portions of the system were only partially implemented, namely the relator and the rough design phase. Some changes had to be made when implementing the system, mainly in the detailed design component. However, for the most part, the original design was implemented as planned. The resulting system was able to produce a display of the data generated by TENNIS. Each of the system components will be described in detail.

5.1 Overall System Structure

The implementation of COURT was done using C++ on a Unix™ workstation. C++ was chosen because an object oriented language was required that could easily interface with existing applications and with a window system. Unix was chosen because of its availability and its power to handle applications requiring large resources. In addition, TENNIS was developed using C on a Unix platform. Using this same environment simplifies the communications with TENNIS.

Figure 18 shows the TENNIS system as it normally interacts with its own user interface and as it interacts with COURT. The TENNIS user interface displays all data produced by TENNIS and communicates with TENNIS via a protocol named User Interface Interface (UII). This protocol was designed to allow the TENNIS user interface to be modified without changes to the expert system. As such, it is ideal for allowing a system such as COURT to receive the TENNIS data.

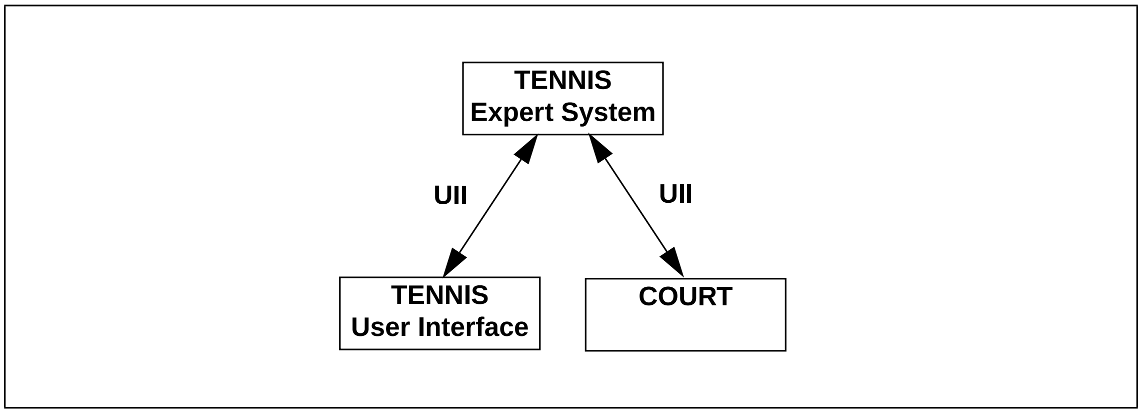


Figure 18: TENNIS Interface and COURT

5.2 Application Interface and Data

The application interface component converts application data from its native format into the data elements that COURT can understand. In the implementation of COURT, a number of different application interfaces were created. An application interface for use with TENNIS consists of a communications unit which speaks the UII protocol to TENNIS, and a conversion unit which converts UII data into data elements. The UII data produced consists of messages concerning a network. The messages contain a text string, and agent, a phase, a function, and sometimes a network component that the message refers to. These messages were easily adapted into data elements. Each component of the messages is converted into a data attribute. The application data is read and stored into a holding pool named `RawDataElts`.

5.3 Script and Parser

Before any processing of the data can occur, the data description must be read from the script. This is done by the parser. The grammar for the VOLLEY data description is listed in Appendix A. The lexical analyzer and parser used to read the script were generated using the flex and bison tools. These programs automatically generate a lexical analyzer and language parser based on a context free grammar provided. The parser generated can read the script and create internal representations of the declarations contained in the script.

The parser stores the data element classes and characteristics information together in a list named `ScriptDataEltClasses`. This list is used by the recognizer. The data abstraction classes are stored in `ScriptDataAbstractClasses`, and the data relation classes in `ScriptDataRelationClasses`. These are then read by the abstractor and relator respectively.

The language proved reasonably easy to parse and sufficiently powerful to express all the declarations required.

5.4 Recognizer

The duty of the recognizer is to determine to what element class the application data elements belong. The recognizer was implemented so as to take elements from the `RawDataElts` list, compare them against the classes in `ScriptDataEltClasses`, and add the element to the instances for the first matching class. A class matches an element when all attributes listed in the element class are contained in the element. In addition, characteristics such as properties are also stored in the data element classes. When an element is recognized, the characteristics are copied into the element.

The implementation of the recognizer finds the first matching class. A simple enhancement would be to match the most specific classes first. The implementation of the recognizer revealed that the data element classes could be enhanced to support more advanced matching techniques beyond checking against the existence of data attributes. Features such as specifying a value or a range of values would make the interface designer’s job easier.

Figure 19 shows how data element classes are stored after being read from the script. This figure also shows the recognized data element instances and their relation to the element class.

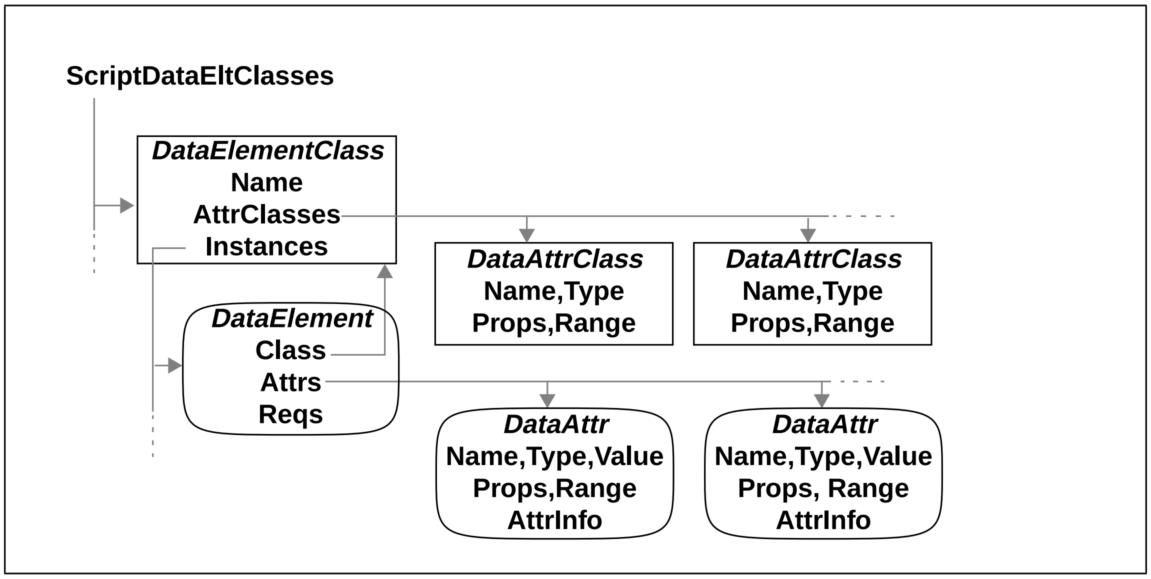


Figure 19: Data Element Storage Structure

5.5 Abstractor

After being recognized, data elements are sent to the abstractor. The abstractor groups elements according to the abstraction classes stored in `ScriptDataAbstractClasses`. The abstractor as implemented builds what is known as an abstraction candidate for each abstraction class. The abstraction class references are used to find all possible elements that may be combined to form an abstraction of the specified class. These elements are referenced by the candidate. The abstractor then generates all possible combinations of elements that the candidate supplies and tests them against the abstraction class constraints. If the constraints are satisfied, the abstractor instantiates an abstraction. The instantiated abstraction is attached to its class.

Figure 20 shows how the data abstraction classes are stored after being read from the script. This also shows the data abstraction instances and their relation to the abstraction class.

As with the recognizer, the abstractor and corresponding abstraction declarations could be improved to increase the expressive power of the declarations. Constructs such as existen-

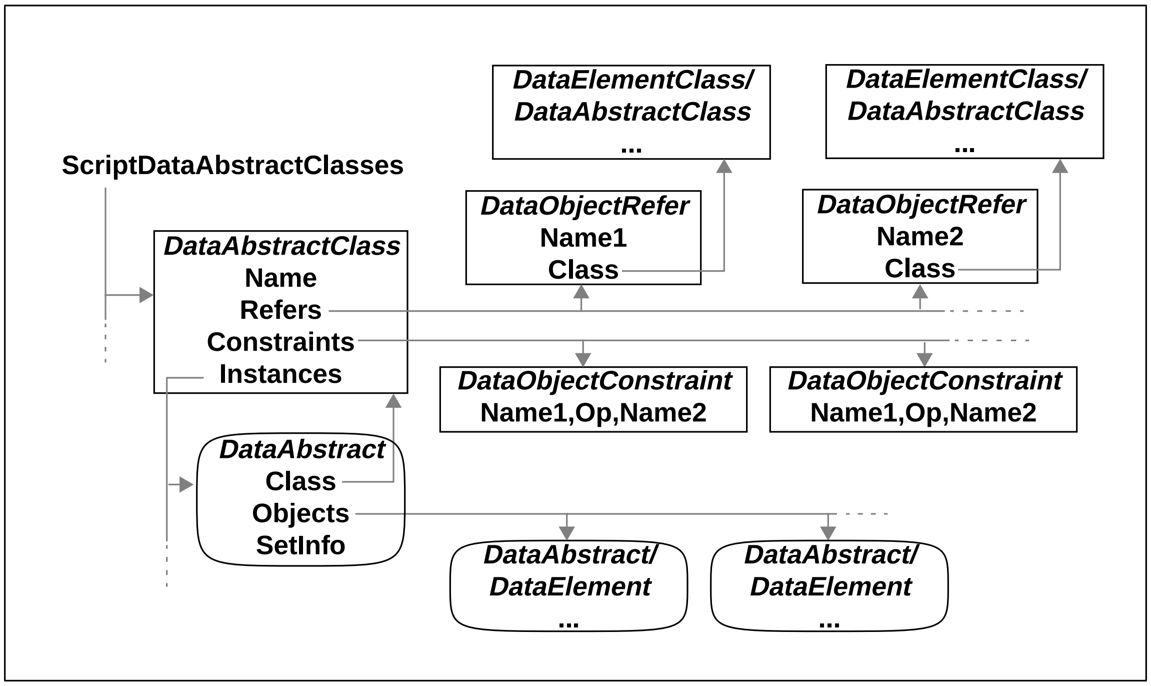


Figure 20: Data Abstraction Storage Structure

tial and universal quantifiers in the declaration would decrease the difficulty of specifying the members of an abstraction.

5.6 Relator

The relator was not implemented completely. The parser reads and stores the data relation classes into `ScriptDataRelationClasses`, but the relator was not written to produce the relation instances. However, implementation of the relator would be very close to the abstractor. A relation candidate would be created that referred to all elements that could be in the relation. The relator would then generate and test each relation that the candidate could produce and place valid relations in the instances for the relation class.

The same comment concerning expressive power of the references and constraints could be made about the relation classes, since they follow the same format as the abstraction classes.

5.7 Rough Design

After the data manipulation components come the interface generation components. These are the rough design and detailed design components. The implementation of the rough design phase differed from its original design in two ways. First, the rules were not implemented but rather a procedural approach was taken that captured what would have been encoded in rules and provided a basic rough interface design. Second, some functionality was moved from the rough design phase into the detailed design phase. Specifically, the abstraction window layout portion of rough design was relocated to the detailed design phase.

The general duties of the rough design phase are to analyze the data, generate top level and abstraction windows, and determine the abstraction window layouts.

The analysis of the data was implemented to determine useful statistical information on the data. The count of unique values for an attribute in a set of elements proved useful in performing the detailed design. The results of the data analysis is stored in a structure named a `DataSetInfo` which is attached to each data abstraction. This structure contains a summary, called a `DataAttrInfo`, for each of the attribute classes contained in a set of elements. When creating these summaries, the system distinguishes between attributes that have the same class but are in a different range. For example, a set of component elements may have coordinate attributes X and Y. Though these coordinates may have similar characteristics, the values for the attributes are not within the same range. There are two ranges for the two attribute classes.

Figure 21 shows the structures resulting from the data analysis. The figure shows the data abstraction with the elements it contains, and the attributes contained by the elements. Attached to the abstraction is a `DataSetInfo` containing the data summary. Also shown

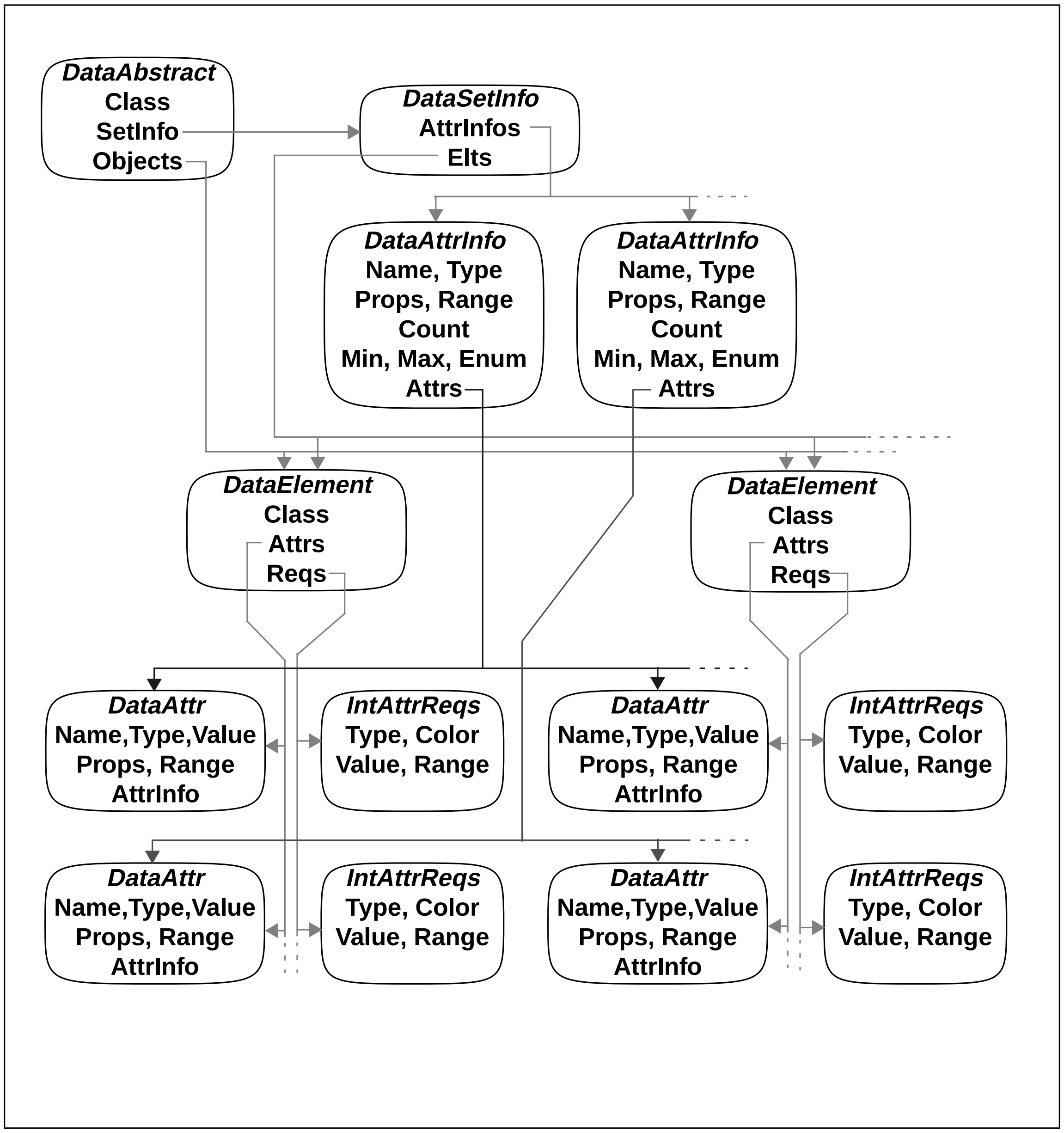


Figure 21: Data Analysis Information and Detailed Design Requirements Structures

are the interface attribute requirements which are described in the detailed design section.

The determination of top level and abstraction windows was designed to be done through rules, but was implemented to simply create a top level window containing a single abstraction window inside for each top level abstraction encountered. Though rules would have handled the window creation more subtly, the approach implemented produced a reasonable structure.

The rough design phase was to have determined the abstraction window layouts. This proved to be more straightforward to handle in the detailed design phase. This is discussed further under the detailed design implementation.

Because no relation information was being generated, the relations could not be mapped into interface attribute requirements.

5.8 Detailed Design

The detailed design phase was implemented to complete the interface design. This involves creating interface objects for the top level and abstraction windows, as well as determining interface attribute requirements and interface objects to satisfy those requirements.

The creation of top level interface objects was a simple process of creating one interface object per top level window created in the rough design. Each top level window contains a single abstraction window, so there are no layout constraints to be solved.

The detailed design implementation uses rules to map data attributes to interface attributes. The mapping takes into account the context free issues such as the data type and properties, as well as the context sensitive issues such as the number of unique values in the set (known as the enumeration count in the implementation) and the data range. For example, numeric attributes containing a property position can be mapped to a position attribute. String attributes with a low enumeration count may be mapped to a shape attribute. A complete listing of the detailed design rules that were implemented is contained in Appendix B.

The detailed design must take into account the number of dimensions available in each interface attribute. For instance, the shape attribute can only be used to encode a single range of values. Data attributes in two separate classes cannot be mapped to interface attributes in the same dimension. The position and color attributes both allow two dimensions, whereas all other attributes, except text, have only one dimension. Text attributes can have as many dimensions as are needed by prepending the dimension name to the value of the attribute.

The design of the rough design phase calls for it to determine the abstraction window layouts. This proved to be better done in the detailed design while determining the interface attribute requirements. Because the position of interface objects is simply an attribute of the objects, the mapping to these attributes could be done by the detailed design rules. The position attribute only allows for two dimensions, so up to two attributes could be encoded using position.

Once the interface attribute requirements have been determined, the requirements must be mapped to interface objects. This was implemented using a set of heuristics. The heuristics examined the attribute requirements and determined the best objects to use to represent the attributes. For example, if an attribute requires more than one position attribute in at two dimensions, a line object would be appropriate. Once the object is chosen, as many attributes as possible are mapped to it in order to create the most efficient display possible. Further objects may be required to satisfy all attribute requirements. Unmapped attributes in the subsequent objects will take on the same value as previous objects, so as to provide consistency between objects representing the same data element.

5.9 Interface Objects

Motif was chosen as the window system due to its popularity and its availability on the development platform. The interface objects were implemented by creating a set of C++ classes that encapsulated Motif widgets. Motif allows a high degree of flexibility in creating widget trees. In order to simplify the creation of Motif displays, the interface objects automatically create and manage a number of Motif widgets.

All interface objects were implemented. The implementation of interface objects showed that they have sufficient power to be combined to produce a complete display, and that they can be relatively easily reimplemented to create interfaces on other windowing systems. This is partly due to the inherent commonality between windowing systems, and also due to the simplicity of the interface object requirements. Because only a few basic interface concepts are represented in the interface objects, they can be easily ported to other systems.

5.10 Summary

The implementation of the COURT system produced a working prototype that was able to validate the design, show where it had to be modified, and reveal areas for future improvement. The major modification was the relocation of abstraction window layout to the detailed design stage. The next chapter shows examples of the system operation.

Chapter 6 Usage

This chapter shows examples using the COURT system. Four examples of use are shown, one involving the TENNIS system, one involving data from Napoleon's campaign in Russia, and two displaying process information from the UNIX ps program. These examples show the general operation of COURT, the types of displays it can generate[1], the use of different scripts to produce alternate displays, and the use of different applications.

6.1 Usage with TENNIS

This example shows the use of COURT with TENNIS. TENNIS is a multi-agent expert system that produces messages concerning the ease of servicing networks and network components. TENNIS can produce many messages concerning a network. Some of these messages, known as indices, provide summary information for the evaluation of the network. The COURT interface for TENNIS was used to display these summary messages. A sample index message is shown below:

```
String Category Index1
String Item Components per Series
Numeric Min 0
Numeric Max 232
Numeric Value 7.73
Numeric Sequence 75
```

The `Category` attribute is the general type of index being provided. TENNIS provides two levels of indices which correspond to primary and secondary evaluations of the network. The `Item` attribute is the name of the index being provided. The `Min`, `Max` and

1. Please note that all figures in this chapter are located at the end of the chapter.

`Value` attributes provide the value itself, along with scaling information. The `Sequence` attribute is the sequence number of the message provided from TENNIS. All messages provided are numbered sequentially using this attribute. The script used to describe the indices is as follows:

```
element Index has
     numeric Min, Max
     string Category, Item
     numeric Sequence, Value
end

characteristics of Index
     Sequence properties are order
     Value properties are size
     Category, Item properties are label
     Min, Max properties are label
end

abstraction Indexes has
     all Index
     where
     Category of Index is same
end
```

This script specifies that `Inde`x elements contain `Min`, `Max`, `Sequence`, and `Value` numeric values, along with `Category` and `Item` strings. The characteristics declaration specifies that the `Min`, `Max`, `Category` and `Item` attributes are `label`s. The `Sequence` specifies an `order` and the `Value` is a `size`. An abstraction named `Indexes` is built which contains all the `Index` data elements. A constraint in the abstraction class specifies that all the `Index` elements in an `Indexes` abstraction must have the same value for the `Category` attribute.

Figure 22 shows the resulting interface generated by COURT. When building the abstractions, COURT created two top level abstractions, each containing all the elements that belong in the same category. To show these abstractions, the indices have been split into two top level windows, corresponding to the two abstractions. In both windows, the `Sequence` of the message is shown on the X axis and the `Value` of the index on the Y axis. The design rules in COURT determined the combination of a numeric value and an order or size property should be mapped to a position interface attribute. Remaining data attributes in each element are mapped to text strings.

One of the major limitations of the COURT implementation is shown in this example. The text strings are positioned so that conflicts with each other are possible. The text strings overlap when many are placed in the same area on the screen. This problem occurs because the rough design phase does not examine the data in an abstraction to determine the abstraction window layout. The layout is performed in the detailed design phase by the rules. The detailed design rules are ordered to emphasize the importance of the position dimensions available in a window. The rules attempt to map data to position interface attributes first, and use other interface attributes after the position attributes are mapped. By always mapping data to both dimensions of position, a list abstraction window does not get created. The list window would map a data attribute on one axis, but not use the other axis for any attribute, allowing a list of text strings to be cleanly displayed. This problem caused COURT to produce unusable displays when attempting to show the entire set of messages produced by TENNIS.

6.2 Usage with Napoleon Data

This example uses COURT to create a display of data from Napoleon’s campaign in Russia in 1812. It is derived from a graphic originally produced by Charles Minard [Tufte 83]. The data includes information about the eastward advance and westward retreat of Napoleon’s troops, the battles they fought along the way, and the troop size and temperature at points in the campaign. The data are broken into two parts, the battles fought and the segments of the march. Each of the battles and march segments can be represented with a data element. Battles have a city name, a date, and a location. A sample battle data element is shown below.

```
String City Borisov
String Date Nov 16
Numeric Latitude  54.25
Numeric Longitude 28.5
```

March segments have starting and ending locations, a troop size, and a temperature. A sample march segment is shown below.

```
Numeric StartLat  54.25
Numeric StartLon  28.5
Numeric EndLat    54.35
Numeric EndLon    26.8
Numeric Troops    28000
Numeric Temp      -24
```

The VOLLEY description of this data is provided below.

```
element Battle has
     string City, Date
     numeric Latitude, Longitude
end

characteristics of Battle
     City properties are label
     Date properties are time
     Latitude, Longitude properties are position
     Latitude range is LatRange
     Longitude range is LonRange
end
```

```
element March has
     numeric StartLat, StartLon
     numeric EndLat, EndLon
     numeric Troops
     numeric Temp
end

characteristics of March
     StartLat, StartLon properties are position
     EndLat, EndLon properties are position
     StartLat, EndLat range is LatRange
     StartLon, EndLon range is LonRange
     Troops properties is size
     Temp properties is temperature
end

abstraction Campaign has
     all Battle
     all March
end
```

The `Battle` data element class lists the attributes in the elements. These are `City`, `Date`, `Longitude` and `Latitude`. The characteristics declaration immediately following provides the properties and ranges of the data. `City` is a `label` and `Date` is a `time`. The `Latitude` and `Longitude`s are `position`s. Ranges are for the `Latitude` and `Longitude` are also specified. These ranges will be used later to correlate with the battles. The next declaration is for `March` elements. These elements have start and end coordinates, as well as `Troops` and a `Temp`. The characteristics declaration for `March` specify that the coordinates are `position`s. The coordinates have the same range as the `Battle` coordinates. This indicates to COURT that these coordinates can be presented in the same attribute dimension, such as an axis. Appropriate properties are provided for `Temp` and `Troops`. The last declaration is an abstraction of the data called a `Campaign` which includes all `Battle` and `March` elements. This abstraction specifies that the two types of data elements should be displayed as a single structure.

Figure 23 shows the interface that COURT generated to display this data. The march segments are shown as lines, and the battles are shown as text strings. COURT creates interface attribute requirements for each of the data attributes in the data set. The position data attributes are mapped to position interface attributes first. The remaining data attributes are mapped into appropriate interface requirements, such as temperature to color, and size to thickness. Because the `March` elements contain multiple position attributes, COURT chose to use lines to represent the data. The color and thickness interface requirements are also satisfied by the lines. Battle elements contain position information and labels. The detailed design rules determined that these attributes are best mapped to text strings, since they can encode a position and a string. The display COURT produced corresponds closely to the graphic originally drawn by Minard. It represents an efficient use of display attributes. Because the data contains a wide variety of data types and properties, COURT can efficiently encode the data attributes into the interface attributes available in the system. The evaluation of the display is discussed in more detail in Chapter 7.

6.3 Usage with PS

PS is a UNIX program that displays the status of processes (programs) running on a computer. It provides a set of values for each running process, such as the owner, the amount of time it has been running, and the amount of memory it uses. The standard output of the program consists of text aligned to make a table. Each process is displayed on one line. An excerpt from the standard output of ps is shown below:

```
USER        PID %CPU %MEM   SZ  RSS TT STAT   TIME COMMAND
blynch     4011  0.0  1.5 3852 2532 q5 I      0:01 court
blynch     4027  0.0  0.3  736  544 q5 R      0:00 ps
blynch    18021  0.0  0.2  628  276 p1 I      0:00 tcsh
blynch     4028  0.0  0.1   88   80 q5 S      0:00 more
```

This application lends itself well to COURT due to its use of multiple attributes that can be combined into elements. In order to provide COURT with data from ps, a filter was written to convert ps output into COURT input. An example of the output from the filter is shown below:

```
String User blynch
Numeric PCPU  0.0
Numeric PMEM  1.5
Numeric VSize 3852
Numeric RSize 2532
String State I
Numeric Time 1
String Command court
```

The output above shows the attributes of a single element. The `User` attribute is the owner of the process. The `PCPU` and `PMEM` attributes correspond to the percentage of CPU time and memory used by the process, respectively. These are supplied in the `%CPU` and `%MEM` columns in ps. The `VSize` and `RSize` attributes show the virtual size and real size of the process. These come from the `SZ` and `RSS` columns of ps. The `State` attributes comes from the STAT column in ps and represents the status of the process, such as running or waiting. The `Time` attribute show the elapsed time the process has been running, and `Command` shows the process name. The script used to describe the ps data follows:

```
element Process has
     string User, State, Command
     numeric VSize, RSize, PCPU, PMEM, Time
end

characteristics of Process
     User, Command properties are label
     VSize, RSize properties are size
     State properties are status
     Time properties are age
end
```

```
abstraction Processes has
     all Process
end
```

This script specifies an element named `Process`. A `Process` has a `User` (process owner), a `State` (running or stopped), and a `Command` (program name) which are strings, and a `VSize`, an `RSize`, a `PCPU`, a `PMEM`, and a `Time` that are numeric. The characteristics of the `Process` specify properties for each of the attributes. Because only one set of data is being used, the system does not need to correlate any data, and thus no ranges are needed. An abstraction of the `Process` elements named `Processes` is specified. This abstraction sets contains all the `Process` elements. Figure 24 shows the interface for ps generated by COURT. The detailed design rules in COURT first matched the `VSize` and `RSize` data attributes to position interface attributes. Next, the `Time` data attribute was mapped to color to indicate the age of the process. A solid green is a new process where a brown one is old. The remaining data attributes were mapped to additional text interface attributes.

Different users of the system will have different requirements for its output. A script specifying different data element groupings is required to change the data element groupings in the display. In this example, a different user wishes to see the process information split into larger and smaller processes. The alternate script is shown below:

```
element Process has
     string User, State, Command
     numeric VSize, RSize, PCPU, PMEM, Time
end
```

```
characteristics of Process
     User, Command properties are label
     VSize, RSize properties are size
     State properties are status
     Time properties are age
end

abstraction SmallUserProcesses has
     all Process
     where
     User of Process <> “root”
     VSize of Process < 2000
end

abstraction LargeProcesses has
     all Process
     where
     VSize of Process > 2000
end
```

Different abstractions are specified in this case. The `SmallUserProcesses` abstraction contains all `Process` elements which are not owned by “root” and which are smaller than 2000. The `LargeProcesses` abstraction contains all `Process` elements whose size is greater than 2000. These two abstractions result in two windows, as shown in Figure 25. The top window displays the `SmallUserProcesses` abstraction. The elements are shown using text strings and, as before, plotted against the `VSize` and `RSize` attributes. The lower window shows the `LargeProcesses` abstraction. The elements here are displayed using node interface objects instead of just text strings. The node shape represents the process status. Though it cannot be seen directly from the display, the `State` attribute of the `Process` elements has an enumeration of four or fewer. This difference allows COURT to adapt by using a node shape to encode this value instead of simply using a text string.

There are two limitations apparent in the display of the data from ps. First, there is no key to specify how the interface attributes are being used. COURT as implemented does not include support for generating legends. Second, the text strings overlap and many are unreadable. This is another example of the problems created by not determining window layout in the rough design phase. The detailed design phase does not have information about the types of data being displayed and cannot ensure that text strings will not overlap.

6.4 Summary

COURT can be used to create displays for a variety of applications, as well as allow easy modification of the displays produced to suit the requirements of different users. A number of aspects of the displays can be improved, including the layout of elements, the overlapping of text strings, and the absence of legends. These are largely implementation concerns. The next chapter provides an evaluation of the COURT system.

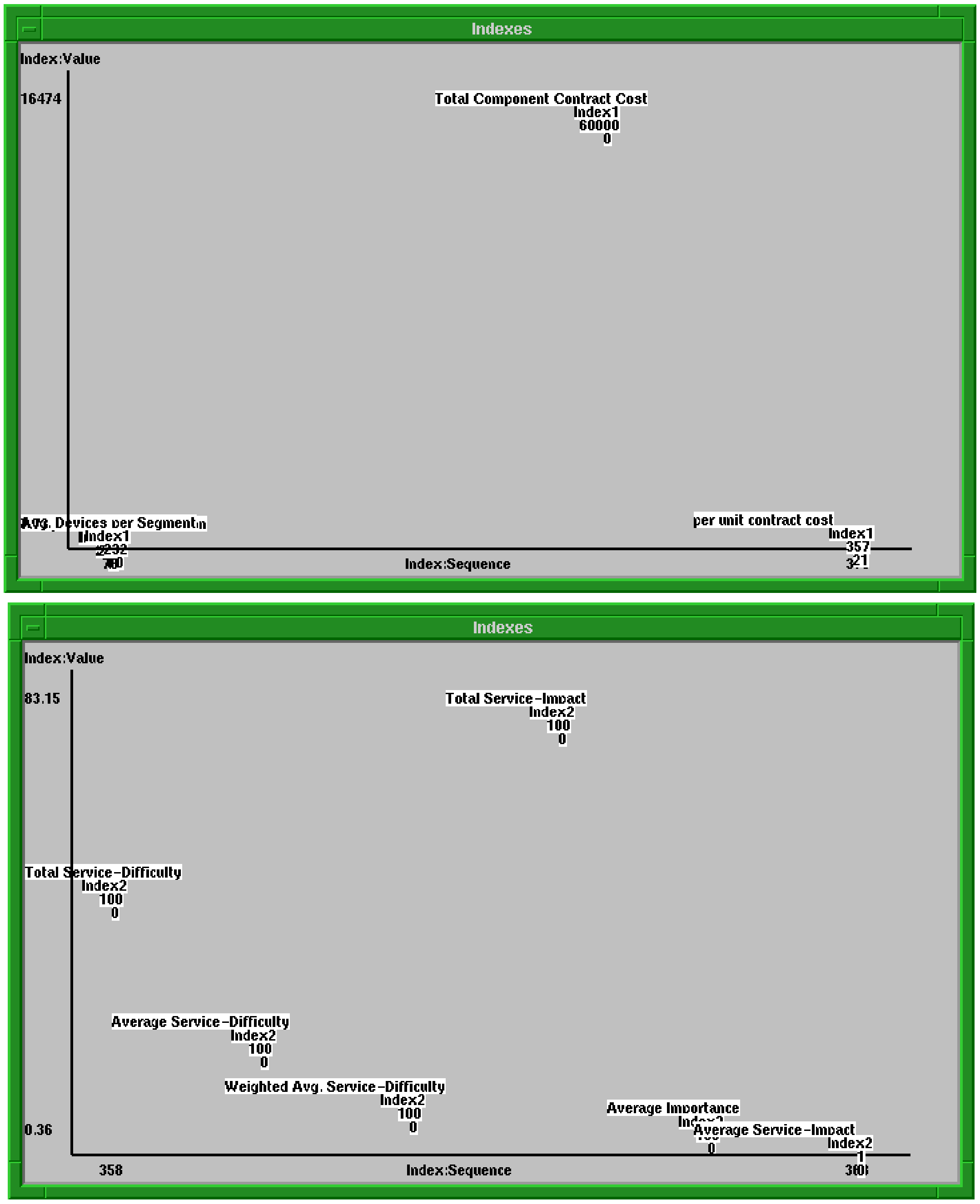


Figure 22: COURT Generated Displays for TENNIS

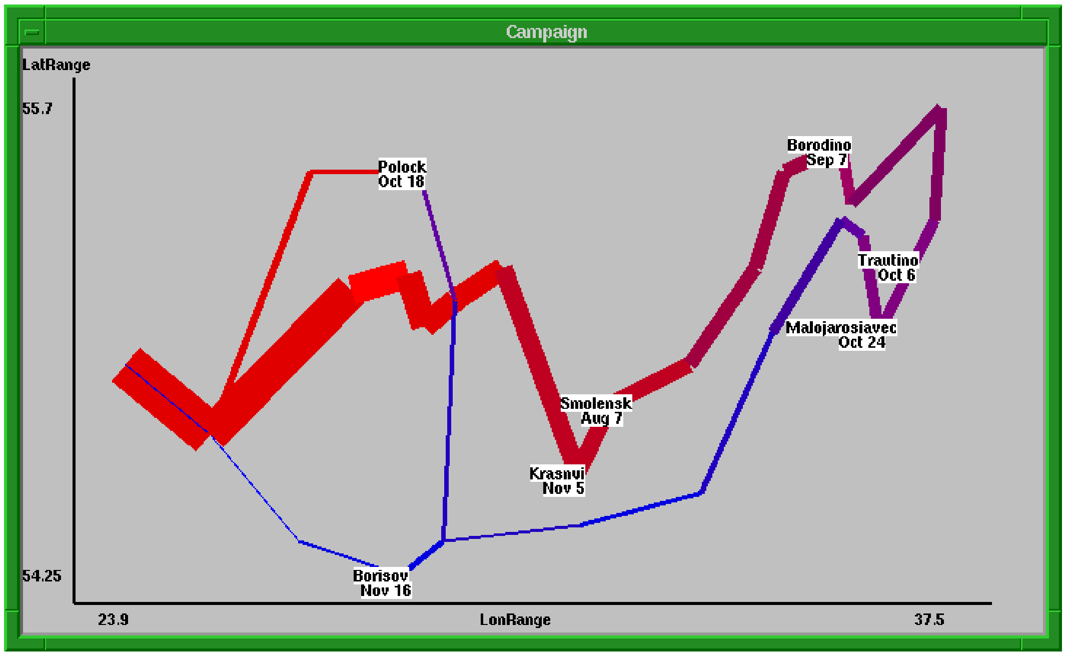


Figure 23: COURT Generated Display of Napoleon’s March

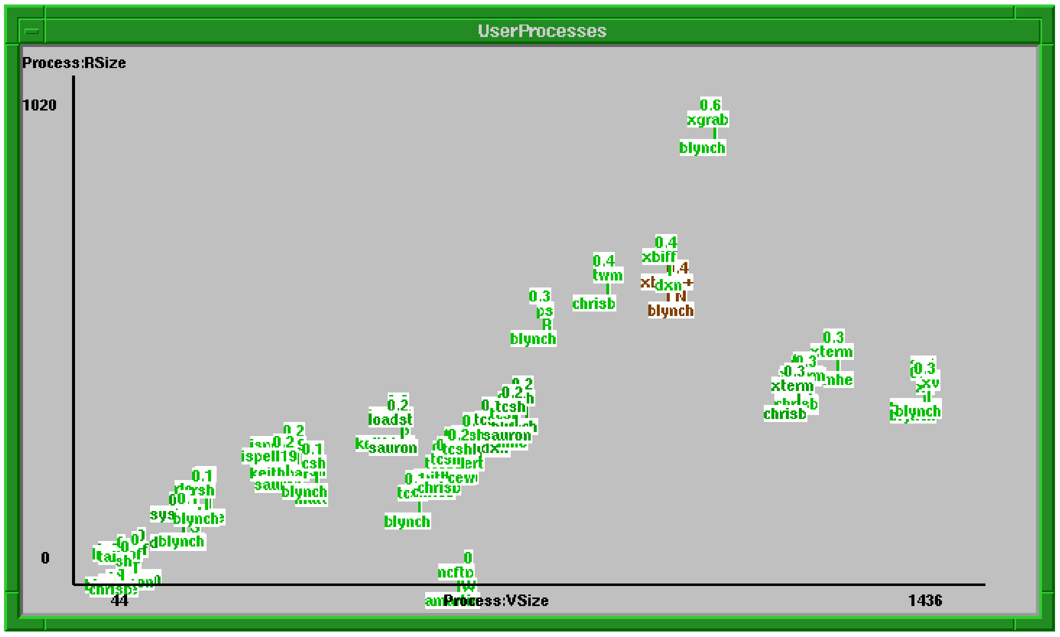


Figure 24: COURT Generated Display of ps Data

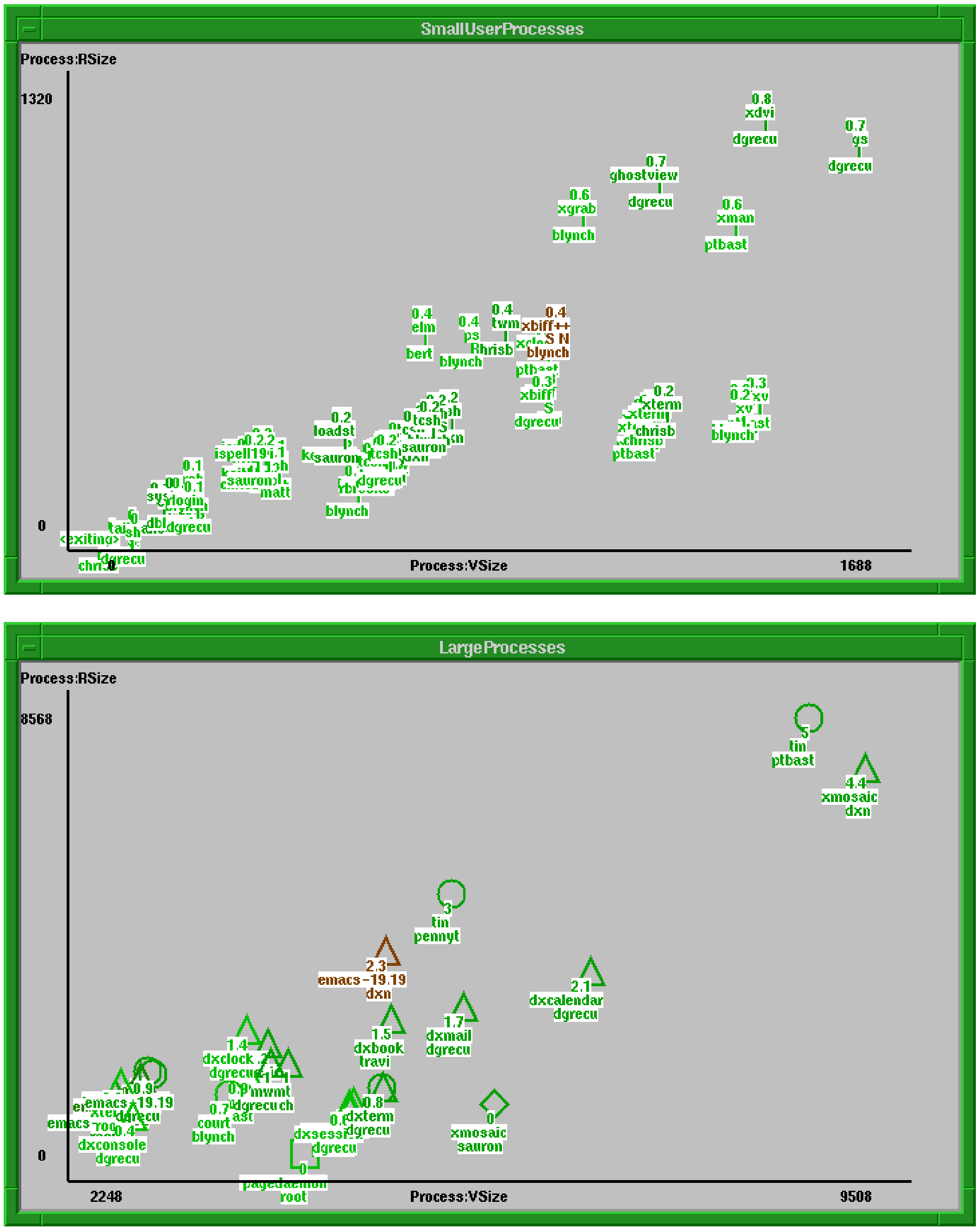


Figure 25: Alternate COURT Generated Displays of ps Data

Chapter 7 Verification and Evaluation

The evaluation of the system involves verifying its operation as well as evaluating the interfaces generated. Verification involves testing the implementation to ensure it is working properly. Evaluation is a subjective assessment of how well the system reached its goals.

7.1 Verification

Verification of the system consists of ensuring that the implementation of the various components worked as they were designed. This includes verifying that the script was read properly, the data elements were recognized properly, characteristics were added correctly, abstractions and relations were created properly, the correct rules were applied, the all aspects of the interface were correct. All these aspects of the system were tested during the development of the system and found to be working correctly.

7.2 Evaluation

The evaluation of COURT can be done using a number of criteria.

- *Design goals.* The original design goals specify criteria for the characteristics of the system. This includes properties of the language, interfaces generated, and operation of the system. The goals provide the most important evaluation criteria.
- *Effectiveness of generated interfaces.* The interfaces generated by COURT can be evaluated for their effectiveness in presentation. This can be done by comparing the generated interfaces to existing interfaces, and evaluating the generated interfaces for their effectiveness.
- *Performance of system.* The speed and ease of using COURT can be compared to tra-

ditional interface generation methods.

- *Comparison to other systems.* COURT can be compared to existing work in the area of automatic presentation systems, such SAGE. This provides a comparison for the flexibility and power of the system.

Each of these categories of evaluation criteria are examined more closely.

7.3 Evaluation With Respect to Goals

The evaluation of COURT with respect to the design goals compares how well each of the goals was accomplished. Each of the design goals from Chapter 3 is examined below with respect to how closely it was met.

7.3.1 Describe Interface at a High Abstraction Level

This goal states that the interface designer should not be specifying concrete interface components, but rather he should describe the interface in an abstract manner. Abstract interface descriptions can include the ways in which data should be shown, groupings of data within windows, and relations between the windows. Though the interface designer cannot directly specify these interface attributes in an abstract manner, the data description provides a way to describe what data should be shown, the groupings of data that should be displayed, and the relations between the data, all in an abstract manner. These descriptions of data are eventually used to determine the corresponding description of the interface. Though the interface is not described directly, the abstract data description provided by the designer is an abstract interface description.

7.3.2 Describe and Filter the Data

The goal of describing and filtering the data specifies that the interface designer has freedom in specifying the characteristics and structuring of the data, and determining what data is to be displayed. COURT allows a data description to be provided in the form of a script. The script specifies characteristics of the application data as well as abstractions of and relations in the data. The declaration of these abstractions and relations allows for constraints to be placed on the data. These constraints limit the data included in a grouping and determine what data is shown in the resulting display.

7.3.3 Allow for Multiple Users

This goal states that different users be able to visualize the same data differently, depending on their requirements. The user requirements can be based on what data element a user is interested in seeing, or on how the data elements are displayed. This goal is met through the use of a data description provided along with the data. For each type of user, a different set of data abstractions and relations can be provided to COURT in the form of a script that specifies the data to be displayed. The script is provided along with the data when the system is run, allowing the description to change at any time.

7.3.4 Incorporate Design Guidelines

Design guidelines should be incorporated into the system to ensure that an effective display is produced. These guidelines specify the appropriate use of interface attributes, such as color and size, for encoding data. In COURT, these guidelines are stored as rules. The various kinds of rules in COURT represent knowledge of screen layout, the use of interface attributes to represent data, and the efficient encoding of interface attributes using interface objects. These rules work together to produce an effective display.

7.3.5 Allow Modification of Interface Style

This goal specifies that the interface style be modifiable. Because the style of the interface is determined by the rules that generated the interface, changing the rules allows the style to be changed. COURT was designed to allow the rules to be loaded externally. The implementation of the system encoded the rules internally, but their internal representation is in a form that could easily be read from an external source. During the development of the system, the rules were modified to produce different results. This suggests that changes to an external representation of the rules would be even easier to perform.

7.3.6 Decouple Interface from Application

COURT should detach itself from the application as much as possible. This goal states that the only connection between COURT and the application be for the transfer of data. This separation provides application independence in COURT and minimizes dependencies on data transfer between COURT and the application. Decoupling the application was achieved in COURT by designing it as a stand-alone system which reads application data in a simple standard format.

7.3.7 Automatically Generate Interface

COURT should automatically generate an interface to display the data for an application. Neither the interface designer nor the application should have to specify any aspects of the interface. As shown in Chapter 6, this goal was met with a certain degree of success. While COURT was able to effectively display some data sets, others were not as effective in presentation.

7.3.8 Summary of Evaluation with Respect to Goals

Overall, COURT met the design goals. The system as implemented allows application data and an abstract description of the data to be provided, and, using incorporated interface guidelines, automatically produces an effective interface.

7.4 Evaluation of Interfaces Generated

The COURT generated interfaces for various applications can be evaluated for their effectiveness in displaying the data according to user requirements. The evaluation can be done be examining the displays on their own, and by comparing the COURT generated displays to those of programs created specifically to display the data, such as the standard TENNIS interface. The COURT generated interface for TENNIS, the ps data and the Napoleon data are evaluated below.

7.4.1 Evaluation of Interface Generated for TENNIS

The TENNIS interface generated by COURT displays the service ease indices produced by TENNIS. These indices provide a summary of the evaluation of a network. The standard TENNIS interface used to display the service ease indices is shown in Figure 26. There are two windows generated to contain the two categories of indices. Vertical bars are used to display the values of the indices. Figure 22 shows the TENNIS interface generated by COURT. Different aspects of the two interfaces are compared below.

7.4.1.1 Layout

The standard TENNIS interface separates the indices into two windows, as does the interface generated by COURT. In both cases, the two windows are used to separate the indices into the two categories to which they can belong. The separation into windows in the COURT generated interface is due to the abstraction class definition in the script. This shows that COURT is capable of making the same rough layout decisions as the TENNIS interface designer made, given the same data and knowledge of the data.

The standard interface uses only the X axis to show information about the indices, whereas COURT places the indices in a two dimensional graph. In the standard interface, the X axis shows the ordering of the indices, but not the time they were received. In COURT, the indices are placed along the axis so as to indicate the relative times they were received. The Y axis is used by COURT to encode the value of the index. The use of the axes in both interfaces is similar, though the exact details of the representation differ. COURT has suf-

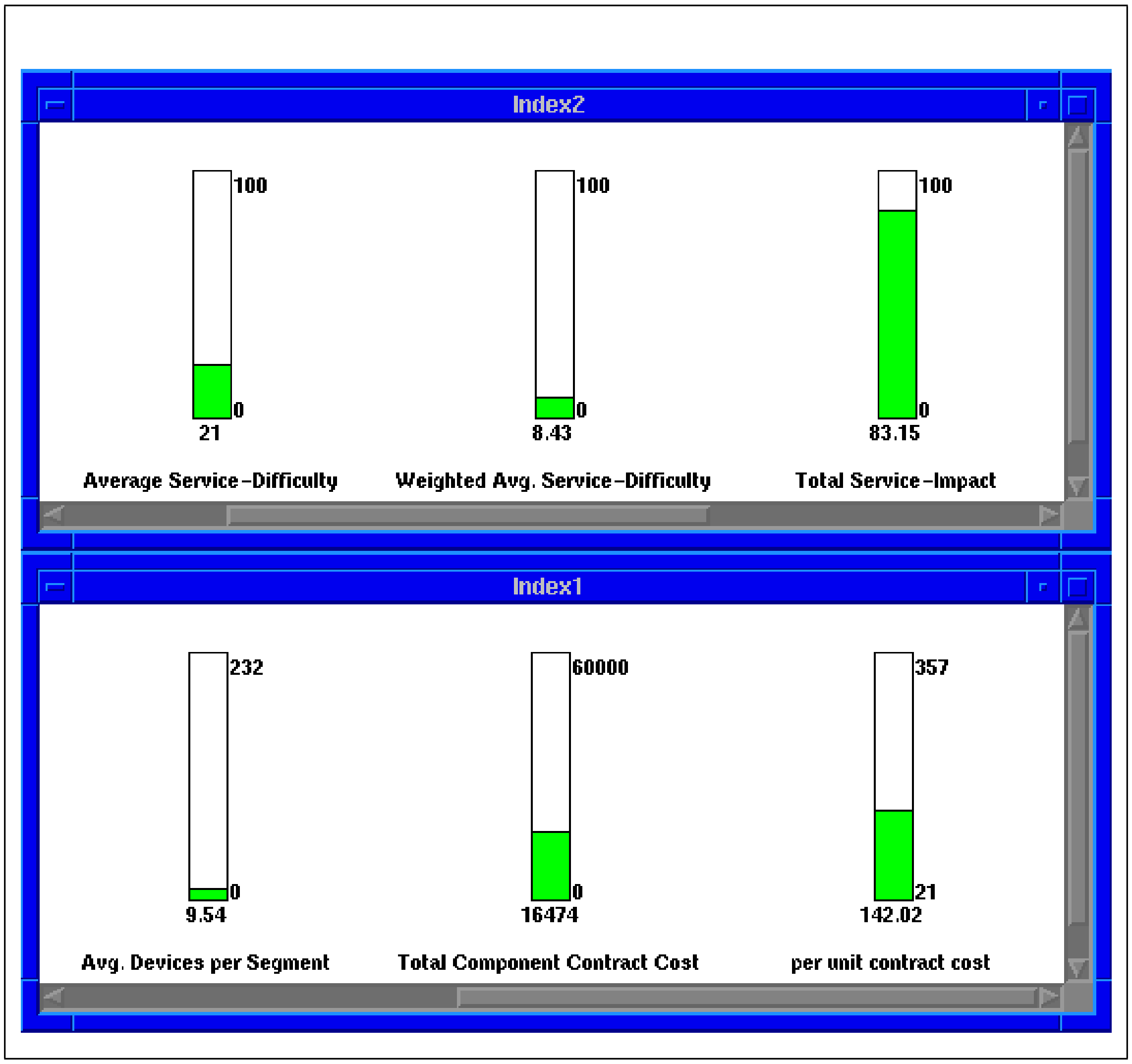


Figure 26: Standard TENNIS Display of Indices

ficient knowledge to determine the general use of the axes, but fails to refine their use to be as effective as the standard interface.

7.4.1.2 Choice of Interface Attributes

The standard interface uses bars and text strings to represent the indices, whereas COURT uses text strings only. The name of the index is shown in a text string in the COURT interface, and below it is the category name and the scaling values. In the standard interface, the bars are used to indicate the value of the index relative to the scaling values provided. This makes the standard display more readable.

7.4.1.3 Usability

The major deficiencies in the COURT generated interface are the use of the X axis for time instead of ordering, and the inability of COURT to use the scale information provided with each index.

The use of the X axis for time in COURT causes some messages to overlap. Though COURT provides an order property that distinguishes a value from a time property, the implementation does not use this property as it should. The system uses the value as though it were a time value instead.

Within a category, the indices are not all normalized to a single scale. Each index sent specifies the scale within which the value lies. The scale information is used by the standard interface to determine the height of a bar. However, in the COURT interface, the values are all plotted against a common scale, making the display less effective.

Overall, the COURT generated display shows the same rough design as the standard interface uses, but the details of the design still need to be refined.

7.4.2 Evaluation of Interface Generated for ps

The interface generated for ps was evaluated by Amy Rich, a system administrator in the Computer Science department at WPI. The data, script and COURT generated interface for ps are shown in Chapter 6.

Initial comments were that the interface was relatively easy to generate and provided a good overall display of the process information. A display such as this is good for a novice user who is often more comfortable with a graphical interface than the standard text based tabular output. The correlation of process information in the two axes can be more useful than the standard output of ps.

Some comments concerning the script included making the specification of elements more flexible. A useful addition would be specifying optional attributes that may or may not be present in the elements. Optional attributes would allow the user to create different output from ps, or any other application, and still allow COURT to recognize the data elements. It was also noted that the script was relatively simple to understand, and that this simplicity is important for the casual user.

Concerning the display created, it was noted that the text strings are sometimes hard to read. This is because interface objects are sometimes placed too close to each other causing them to overlap. Also, the usage of interface attributes other than position is not explained clearly. A key is needed in each window to explain the usage of shapes, colors, line thickness, etc. The labelling of axes could also use improvement, especially when the dimension of data on the axis is a text string. Currently, only the endpoints of the axis are labelled.

Some interesting observations about the data were made. Some data element sets contain attributes that are concentrated around a particular value, but contain a few spurious attributes with a high deviation. For example, the amount of memory consumed by most processes is relatively low, but a few processes consume a significantly higher amount of memory. These patterns in the data cause the display to be less effective. Because the data is scaled directly to the display, data attributes mapping to position for example will cause many values to be placed at one end of a scale and few at the other. A suggestion to fix this problem was to perform a more thorough analysis of the data to find these patterns, and to create the display accordingly. A more thorough analysis can prune the values that are out of range, or the interface can allow the user to zoom in to parts of the display.

7.4.3 Evaluation of Interface Generated for Napoleon Data

The Napoleon graphic created by COURT is almost identical to the original graphic produced by Charles Minard. The graphic efficiently encodes a large amount of data and “may well be the best statistical graphic ever drawn” [Tufte 83].

The resemblance of COURT’s display to the original graphic shows that COURT allows sufficient expressive power in the data description, and contains sufficient interface design knowledge to make appropriate mappings of data attributes to interface attributes. Mapping of longitude and latitude to position, for example, shows that COURT can allow the data to be described as having position data, and that COURT can make the decision to map this data to a position interface attribute.

The detailed design rules used to produce the display were developed before this example was created. The fact the same graphic was produced by both Minard and COURT given the same input, shows that both made equivalent design decisions. The quality of Minard’s work gives merit to his design decisions, and in turn to the design rules in COURT.

COURT can produce this graphic because the data lends itself well to being displayed using a variety of interface attributes. The data contains attributes that can be mapped to positions, sizes, colors and text. The distribution of the data into these attributes is sufficiently even that COURT does not run into problems such as trying to display many dimensions of a single interface attribute.

7.4.4 Overall Evaluation of COURT Generated Interfaces

In general, COURT did a reasonable job of creating displays for applications. One major problem with COURT is its preference for encoding continuous data attributes on both axes. By movng the layout decisions into the detailed design, COURT cannot create layouts such as lists, which would be better for displays such as TENNIS. The implementation determined window layout by mapping a data attribute to each axis during the detailed design stage, rather than by examining the data more thoroughly in the rough design stage. Implementing the system as it was originally designed would have avoided this problem.

7.5 Performance of COURT

The performance of COURT can be measured in the time it takes to learn the VOLLEY language, the time required to generate interfaces, and the time saved by using the system as opposed to traditional interface generation methods.

The time required to learn the VOLLEY language should not be long for a somewhat experienced programmer. With only four declarations available, the language is relatively simple compared to general programming languages.

The time required for COURT to create interfaces varies depending on the size of the data size and the complexity of the abstractions. For most data sets, the workstation used to implement the system was able to generate an interface within 15 seconds.

The developer time saved by using COURT compares the time taken to create an interface through COURT with the time taken to create an interface using alternate means. Even taking into account the time required to learn the script, using COURT should be approximately an order of magnitude faster than writing a program to create an interface manually.

7.6 Comparison to Other Systems

Comparing the COURT system to other systems can show the advantages and disadvantages COURT has over them. The SAGE system [Roth et al 94] comes the closest to COURT in terms of the data supplied and the interface generated. COURT is briefly compared to SAGE in the areas of data input, effectiveness of display created, and flexibility in changing the design.

7.6.1 Comparison of Data Input

The SAGE system uses SAGEBRUSH to allow an interface designer to specify the data attribute to interface attribute mappings. SAGEBRUSH is a graphical, interactive tool that provides the user with the data to be mapped and allows the user to construct a partial interface description by directly mapping pieces of the data to pieces of the display. SAGE then takes this partial interface description, called design directives, and completes the design by applying design knowledge.

The main difference between SAGE and COURT lies in the input description. SAGE accepts a high level description of data including partial mappings to interface elements, where COURT only allows a data description. In SAGE, the interface designer can specify what interface objects are used to encode information. Because COURT does not require interface specifications, the interface designer is free from interface concerns. All interface mapping knowledge required is contained in COURT.

7.6.2 Comparison of Displays Created

The displays generated by COURT and SAGE are comparable. SAGE was also able to generate the graphic showing Napoleon’s march into Russia, demonstrating that SAGE has the ability to utilize the same set of interface attributes as COURT. SAGE was able to automatically create a legend for the graph, and also provided more appropriately labelled axes. These are areas in which COURT can be improved.

7.6.3 Flexibility in changing design

The specification of interface elements in SAGE allows the interface designer to easily modify the resulting interface. In COURT, modifications to the interface are made by describing the data in different ways. The description contains data characteristics and data groupings. Modifying the data characteristics in the description is not always sensible, because they do not change in the data. The semantics of the data remains constant; so should the characteristics provided in the description. Modifying the groupings in the data description changes the display generated. Changing the groupings makes sense because no groupings exist in the data to start with. The groupings specify what portions of the data the user is interested in seeing, and how the data is structured. All groupings are provided through the data description.

7.7 Summary

The COURT system was able to meet its design goals and compares favorably to other systems. Though many aspects of the displays generated call for future enhancements, the overall system showed that a display could be automatically generated based solely on data and a description of the data. A summary of the work and the potential enhancements to the system are discussed in the next chapter.

Chapter 8 Conclusions and Future Work

The implementation of the COURT system shows it was able to obtain the major design goals. The accomplishments of this work include the successful design and implementation of an automatic presentation system. Implementation and evaluation of the system revealed a number of areas for future work. A summary of the work accomplished and ideas for future work are described in this chapter.

8.1 Conclusions

This thesis examined an automatic presentation system designed to produce a display based on data and descriptions of the data. A hypothesis was formed that a system could be designed to create a display of data given only the data and a description of the data, and that no interface description is required for an effective display to be generated. The accomplishments and impact of the investigation of this hypothesis are described below.

8.1.1 Accomplishments

A survey of existing work in the area of UIMS was performed. A number of systems were described with regard to their inputs, operation and outputs. Different approaches to the problem of automatic interface generation were discussed. The systems were compared as to the design knowledge they incorporated, and the input data they required. This comparison revealed that current research in UIMS is not strongly focussed in the area of automatic presentation systems driven by abstract data descriptions.

The goals of a data driven automatic presentation system were outlined. These goals specified the key aspects of such a system, and provided a basis for the design of such a system.

A design of an automatic presentation system based on data descriptions was created. The design includes descriptions of all systems components and specifies their method of operation, inputs and outputs. The design incorporates useful features of existing systems with ideas about abstract data descriptions and their role in interface design.

The design was implemented to create a reusable interface generation system for various applications. The implementation showed that the design was feasible. Though the major components of the design were implemented, some components, most notably the rough design stage, were not implemented fully.

The system was evaluated with regard to how well it met the design goals, the effectiveness of the displays it created, and its performance both individually and as compared to other systems. The implementation of COURT met the design goals. The system was shown to successful in creating displays, though many improvements can be made to the implementation. COURT interfaces were compared to other displays of the same data and found to be comparable in a number of aspects. The evaluation provided a number of future enhancements to the design and implementation.

8.1.2 Impact on Other Work

Automatic presentation systems have been developed, but none are driven solely by data and descriptions of data. COURT was designed and implemented to use only data descriptions, and not need interface descriptions. The initial success of COURT serves as a base for future work in building systems which encompass the entire interface design process. Currently, the interface designer is required by most systems to supply a partial interface design. Future work in data driven automatic presentation systems can focus on incorporating as much interface design knowledge as possible, and reducing the amount of interface specification required as input.

8.2 Future Work

The evaluation of the COURT system provided a number of possible enhancements. These center on improving the power and ease of use of the system. Each of these enhancements is described below.

8.2.1 Explore Interface Mapping Rules

A more thorough investigation of the rough and detailed design rules is appropriate. The design rules as implemented could be expanded and made more flexible. The implementation of the rough design phase and other components could be completed. Other rules, such as the rough design rules, deserve much attention to determine their exact nature. In addition, more work could be done to determine the types of data to interface mappings and their implications.

8.2.2 Enhancements to Data Description Language

The expressive power of the VOLLEY language could be improved by adding more powerful constructs. In particular, the data element class descriptions could include specific ranges of values that are contained in elements, useful when recognizing data and when determining the scaling of data. Abstraction and relation class descriptions could benefit from existential and universal quantifiers. Additional properties for data elements and accompanying knowledge to use them would increase the range of applications that could be used with COURT. In addition, attribute importance should be expressible.

8.2.3 Further Investigation of Relationships

Relations is COURT were designed but not implemented. Further work can be done on determining the meanings and uses for relationships between data. As with quantities of data, there may be more relations between data than can be displayed with the available interface attributes. Future enhancements could allow data elements to be selectable and the relations between them and other data elements shown when the elements are selected.

8.2.4 Changes in Application Data

As it stands, the system cannot gracefully handle additional data provided by the application, or data removed by the application. To present new data, the entire interface must be destroyed and then regenerated to incorporate the new data. COURT could allow incremental updates of the interface as new data becomes available from the application. By independently varying the time when the interface is updated to incorporate new data and the time when the application data is cleared, the system can provide different types of animated displays. These displays may range from a “movie” showing only one data element at a time, to a “jittery movie” showing groups of data one snapshot at a time, to an interface that responds quickly to slight changes in the application data.

8.2.5 Ability to Summarize Data

For large data sets, displaying all the data may be impractical. By combining data and summarizing it, the user can visualize a representation of the data in more manageable quantities. There are many issues involved with producing data summaries, such as determining what data can be combined for a summary, how the data can be combined, and how the result should be displayed. The data description would have to be modified to include descriptions of what data can be combined and how it can be combined.

8.2.6 Generate Interactive Interface

COURT creates a static presentation of data. Enhancements to the system could allow actions on data to be specified with COURT creating an interactive interface. These enhancements would involve automatically creating dialogues with the user, collecting data and requests from the user, and passing these data on to the application. Of course, to follow in the lead of the input to COURT, the description of the actions must contain no hints of the interface used to make them available. Rather, the actions should be described to

COURT in terms of the data they operate on and the semantics of the functions they provide.

8.3 Summary

This work was founded on a hypothesis that a system could be designed to create a display of data given only the data and a description of the data. The COURT system developed provides an automatic presentation system based on data descriptions only. The development of the system shows that this approach is feasible and provides a new direction in UIMS research. Because no external description of the interface is required, COURT encapsulates all interface design issues. New systems that encompass the entire interface design process can be modelled after the approach described in this work.

Appendix A VOLLEY Language Grammar

The grammar for the VOLLEY data description language is shown here in Backus-Naur Form. The bold words are terminals. The initial non-terminal is Script.

Script:	DeclarationList
DeclarationList:	Declaration |
	Declaration DeclarationList
Declaration:	EltClass |
	Characteristics |
	Abstraction |
	Relation
EltClass:	**ELEMENT** Identifier **HAS** AttrClassList **END**
AttrClassList:	AttrClass |
	AttrClass AttrClassList
AttrClass:	Type IdentifierList
Characteristics:	**CHARACTERISTICS OF** Identifier AttrCharList **END**
AttrCharList:	AttrChar |
	AttrChar AttrCharList
AttrChar:	IdentifierList **PROPERTIES** PropertyList |
	IdentifierList **RANGE** Identifier
Abstraction:	**ABSTRACTION** Identifier **HAS** AbstractReferList AbstractConstraints **END**
AbstractReferList:	AbstractRefer |
	AbstractRefer AbstractReferList
AbstractRefer:	Quantity Identifier OptionalIdentifierList
AbstractConstraints:	*null* |
	WHERE AbstractConstraintList
AbstractConstraintList:	AbstractConstraint |
	AbstractConstraint AbstractConstraintList
AbstractConstraint:	SameConstraint |
	Constraint
Relation:	**RELATION** RelationRefer RelationType RelationRefer RelationConstraints **END**
RelationRefer:	Identifier OptionalIdentifier

```
RelationConstraints:      null |
                          WHERE RelationConstraintList
RelationConstraintList:   RelationConstraint |
                          RelationConstraint RelationConstraintList
RelationConstraint:       Constraint

SameConstraint:           Identifier OF Identifier Equal SAME
Constraint:               Identifier OF Identifier Equal Boolean |
                          Identifier OF Identifier Conditional Numeric |
                          Identifier OF Identifier Conditional String |
                          Identifier OF IDentifier Conditional Identifier OF Identifier

PropertyList:             Property |
                          Property , PropertyList

OptionalIdentifier:       null |
                          Identifier

OptionalIdentifierList:   null |
                          IdentifierList
IdentifierList:           Identifier |
                          Identifier , IdentifierList

Property:                 POSITION | DIFFICULTY | STATUS | TIME | MASS | TYPE | SIZE |
                          LABEL | ORDER | TEMPERATURE
Type:                     STRING | NUMERIC | BOOLEAN
Quantity:                 ONE | ALL
RelationType:             REFERSTO | SIMILARTO | DEPENDSON
Conditional:              Equal | <> | < | > | <= | >=
Equal:                    IS | ARE | =
Boolean:                  TRUE | FALSE
Numeric:                  [0-9]+
String:                   [A-Z][A-Z0-9]*
Identifier:               [A-Z][A-Z0-9]*
```

Appendix B Detailed Design Rules

The rules used in the detailed design process are listed here. These rules map single data attributes to single interface attributes. For a complete discussion of data and interface attributes and the mapping from data to interface attributes, see Chapter 4.

Table 3 lists the rules. The first three columns specify a data attribute and the last two columns specify an interface attribute requirement. The data attribute is specified using the attribute type, such as Numeric, String or Boolean, the attribute properties, such Size, Position or Age, and the maximum value for the enumeration. The enumeration is the number of unique values present in a set of data attributes. The interface attribute is specified as an attribute type, such as Position, Color, or Size, and in the case of Color, the starting and ending color values are specified. Data values mapped to a color attribute are scaled into the color range.

For example, the first rule specifies that numeric data attributes having the position property should be mapped to a position interface attribute. Since the position interface attribute can encode a continuous value, there is no need for a limit on the enumeration of the data attribute.

Table 3: Detailed Design Rules in COURT

Data Attribute Type	Data Attribute Properties	Data Enumeration		Interface Attribute	Color Range
Numeric	POSITION		⇒	Position	
Numeric	TIME		⇒	Position	
Numeric	ORDER		⇒	Position	
Numeric	SIZE		⇒	Position	
Numeric	SIZE		⇒	Thick	
Numeric	SIZE		⇒	Size	
Numeric	MASS		⇒	Size	
Numeric	TEMPERATURE		⇒	Color	Blue->Red
Numeric	STATUS		⇒	Color	Red->Green
Numeric	AGE		⇒	Color	Green->Brown
Numeric	DIFFICULTY		⇒	Color	Green->Red
Numeric	TYPE	4	⇒	Shape	
Numeric	LABEL		⇒	Text	
Numeric	SIZE	4	⇒	Thick	
Numeric		4	⇒	Texture	
Numeric			⇒	Text	
String	TYPE	4	⇒	Shape	
String	STATUS	4	⇒	Shape	
String	TYPE	16	⇒	Color	Enum
String	TIME	10	⇒	Position	
String	TYPE	10	⇒	Position	
String	ORDER	10	⇒	Position	
String	TYPE	4	⇒	Texture	
String		10	⇒	Position	
String			⇒	Text	

Table 3: Detailed Design Rules in COURT

Data Attribute Type	Data Attribute Properties	Data Enumeration		Interface Attribute	Color Range
Boolean	TYPE		⇒	Shape	
Boolean	SIZE		⇒	Thick	
Boolean	POSITION		⇒	Position	
Boolean	SIZE		⇒	Size	
Boolean	SIZE		⇒	Position	
Boolean			⇒	Color	Enum
Boolean			⇒	Texture	

References

[Arens et al 88] Yigal Arens, Lawrence Miller, Stuart Shapiro, Norman Sondheimer, Automatic Generation of User-Interface Displays, *Proceedings AAAI-88*, pp. 808-813, 1988.

[Arens et al 91] Yigal Arens, Lawrence Miller & Norman Sondheimer, Presentation Design Using an Integrated Knowledge Base, in *Intelligent User Interfaces*, Addison-Wesley Pub. Co., Reading, MA, pp. 241-258, 1991.

[Bailey 82] Robert Bailey, *Human Performance Engineering: A Guide for System Designers*, Prentice Hall, NJ, 1982.

[Berlage 92] Thomas Berlage, Using Taps to Separate the User Interface from the Application Code, *Proceedings of the ACM Symposium on User Interface Software and Technology '92*, pp. 191-198, 1992.

[Brown et al 94] David Brown, Craig Wills, Bertram Dunskus & Jonathan Kemble, *Computer Network Ease of Service Evaluation System*, Research Report, Worcester Polytechnic Institute, Worcester, MA, 1994.

[Cordy 92]	James Cordy, Hints on the Design of User Interface Features-Lessons from the Design of Turing, in *Intelligent User Interfaces*, Addison-Wesley Pub. Co., Reading, MA, pp. 329-340, 1991.
[Feiner 91]	Steven Feiner, An Architecture for Knowledge-Based Graphical Interfaces, in *Intelligent User Interfaces*, Addison-Wesley Pub. Co., Reading, MA, pp. 259-278, 1991.
[Fischer & Girgensohn 90]	Gerhard Fischer & Andreas Girgensohn, End User Modifiability in Design Environments, *CHI '90 Proceedings*, Seattle, WA, pp. 183-191, 1990.
[Foley 87]	James Foley, Transformations on a Formal Specification of User-Computer Interfaces, *Computer Graphics*, Vol. 21, No. 2, pp. 109-113, 1987.
[Foley et al 88]	James Foley, Christina Gibbs, Won Chul Kim & Srdjan Kovacevic, A Knowledge-Based User Interface Management System, *CHI '88 Proceedings*, Washington, DC, pp. 67-72, 1988.
[Foley et al 89]	James Foley, Won Chul Kim, Srdjan Kovacevic & Kevin Murray, Defining Interfaces at a High Level of Abstraction, *IEEE Software*, pp. 25-32, 1989.
[Gray et al 93]	Wayne Gray, James Spohrer & Thomas Green, End-User Programming Languages: The CHI '92 Workshop report, *SIGCHI Bulletin*, Vol. 25, No. 2, pp. 46-50, 1993.

[Hayes-Roth 92]	F. Hayes-Roth, Rule-Based Systems, in *Encyclopedia of Artificial Intelligence*, Wiley and Sons, New York, 1992.
[Herczeg et al 92]	Jurgen Herczeg, Hubertus Hohl & Matthias Ressel, Progress in Building User Interface Toolkits: The World According to XIT, P*roceedings of the ACM Symposium on User Interface Software and Technology '92*, pp. 181-190, 1992.
[Jackson 90]	Peter Jackson, *Introduction to Expert Systems*, 2nd edition, Addison-Wesley Pub. Co., Reading, MA, pp. 238-240, 1990.
[Jeffries et al 91]	Robin Jeffries, James Miller, Cathleen Wharton & Kathy Uyeda, User Interface Evaluation in the Real World: A Comparison of Four Techniques, *CHI '91 Proceedings*, New Orleans, LA, pp. 119-124, 1991.
[Kim & Foley 90]	Won Chul Kim & James Foley, DON: User Interface Presentation Design Assistant, *Proceedings of the ACM Symposium on User Interface Software and Technology '90*, pp. 10-20, 1990.
[Lowgren 88]	Jonas Lowgren, History, State and Future of User Interface Management Systems, *SIGCHI Bulletin*, Vol. 20, No. 1, pp. 32-44, 1988.
[Mackinlay 91]	Jock Mackinlay, Search Architectures for the Automatic Design of Graphical Presentations, in *Intelligent User Interfaces*, Addison-Wesley Pub. Co., Reading, MA, pp. 281-292, 1991.

[MacLean et al 90] Allan MacLean, Kathleen Carter, Lennart Lovstrand & Thomas Moran, User-Tailorable Systems: Pressing the Issues with Buttons, *CHI ‘90 Proceedings*, Seattle, WA, pp. 175-182, 1990.

[Myers et al 90] Brad Myers, Dario Giuse, Roger Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish & Philippe Marchal, Comprehensive Support for Graphical, Highly Interactive User Interfaces: The Garnet User Interface Development Environment, *IEEE Computer*, Vol. 23, No. 11, pp. 71-85, 1990.

[Myers 91] Brad Myers, Graphical Techniques in a Spreadsheet for Specifying User Interfaces, *CHI ‘91 Proceedings*, New Orleans, LA, pp. 243-249, 1991.

[Myers 92] Brad Myers, Ideas from Garnet for Future User Interface Programming Languages, in *Languages for Developing User Interfaces*, Jones and Bartlett Publishers, Inc., Boston, MA, pp. 147-157, 1992.

[Myers et al 94] Brad Myers, Jade Goldstein & Matthew Goldberg, Creating Charts by Demonstration, *CHI ‘94 Proceedings*, Boston, MA, pp. 106-111, 1994.

[Olsen 89] Dan Olsen, A Programming Language Basis for User Interface Management, *CHI ‘89 Proceedings*, Austin, TX, pp. 171-176, 1989.

-

[Olsen 92]	Dan Olsen, *User Interface Management Systems: Models and Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, 1992.
[Pfaff 85]	G. Pfaff, ed., *User Interface Management Systems*, Springer-Verlag, Berlin, 1985.
[Pittman & Kitrick 90]	Jon Pittman & Christopher Kitrick, VUIMS: A Visual User Interface Management System, *Proceedings of the ACM Symposium on User Interface Software and Technology ‘90*, pp. 36-46, 1990.
[Roth et al 94]	Steven Roth, John Kolojejchick, Joe Mattis & Jade Goldstein, Interactive Graphic Design Using Automatic Presentation Knowledge, *CHI ‘94 Proceedings*, Boston, MA, pp. 112-117, 1994.
[Roth & Mattis 94]	Steven Roth & Joe Mattis, Data Characterization for Intelligent Graphics Presentation, *CHI ‘90 Proceedings*, Seattle, WA, pp. 193-200, 1990.
[Scott & Yap 88]	Michael Scott & Sue-Ken Yap, A Grammar-Based Approach to the Automatic Generation of User Interface Dialogues, *CHI ‘88 Proceedings*, Washington, DC, pp. 73-78, 1988.
[Seminar & Robson 90]	Kudang Seminar & Robert Robson, An Iconic Description Language: Programming Support for Data Structure Visualization, *SIGCHI Bulletin*, Vol. 22, No. 1, pp. 70-72, 1990.

[Singh 92] Gurminder Singh, Requirements for User Interface Programming Languages, in *Languages for Developing User Interfaces*, Jones and Bartlett Publishers, Inc., Boston, MA, pp. 115-123, 1992.

[Singh & Green 89] Gurminder Singh & Mark Green, A High-Level User Interface Management System, *CHI ‘89 Proceedings*, Austin, TX, pp. 133-138, 1989.

[Singh et al 90] Gurminder Singh, Chun Hong Kok & Teng Ye Ngan, Druid: A System for Demonstrational Rapid User Interface Development, *Proceedings of the ACM Symposium on User Interface Software and Technology ‘90*, p. 167, 1990.

[Sullivan & Tyler 91] Joseph Sullivan & Sherman Tyler, ed., *Intelligent User Interfaces*, Addison-Wesley Pub. Co., Reading, MA, 1991.

[Szekely 90] Pedro Szekely, Template-Based Mapping of Application Data to Interactive Displays, *Proceedings of the ACM Symposium on User Interface Software and Technology ‘90*, pp. 1-9, 1990.

[Tufte 83] Edward Tufte, *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, CT, 1983.

[Vander Zanden & Myers 90] Brad Vander Zanden & Brad Myers, Automatic Look-and-Feel Independant Dialog Creation for Graphical User Interfaces, *CHI ‘90 Proceedings*, Seattle, WA, pp. 27-34, 1990.

[Wiecha & Boies 90]	Charles Wiecha & Stephen Boies, Generating User Interfaces: Principles and Use of ITS Style Rules, *Proceedings of the ACM Symposium on User Interface Software and Technology '90*, pp. 21-30, 1990.
[Zarmer & Chew 92]	Craig Zarmer & Chee Chew, Frameworks for Interactive, Extensible, Information-Intensive Applications, *Proceedings of the ACM Symposium on User Interface Software and Technology '92*, pp. 33-41, 1992.