Representation Issues for Reengineering User Interfaces

Research Report

Melody M. Moore

Open Systems Laboratory and Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology

Feb 19, 1996

Prepared for Depth Exam Committee:

Dr. James D. Foley (chair)
Dr. Ann Chervenak
Dr. Richard LeBlanc
Dr. Spencer Rugaber

Introduction

In [RUG93], Rugaber and Clayton assert that a foundational issue in reengineering is the abstract representation of the system. This is especially true of user interface reengineering - we need to be able to describe the behavior and functionality of the user interface in an abstract manner that is not dependent on any specific display technology or user interface techniques. Yet, the representation must be complete and robust enough to adequately represent all of the functional requirements of the user interface through out the evolution process. Solving this representation problem and building a model of the user interface is key to understanding the process of reengineering.

Several good taxonomies of representations for designing user interfaces exist [ABO89],[DIX93], [FOL90], [HAR89], [GRE87]. While the representations surveyed give the designer a framework to generate a *new* interface, they were not intended to support the recovery of an *existing* interface from a legacy system. The goal of this work was to investigate categories of user interface design and analysis representations for suitability to the reengineering process.

Information Systems

The domain of this study is text-based interactive information systems for several reasons. First, most information systems are very large, some on the order of millions of lines of code. Many old systems are text-based or use simple terminal controls (such as curses). In order to fully utilize new graphical workstation technology, these systems need to be reengineered and their user interfaces retrofitted to adapt to graphical user interface (GUI) environments. The sheer size of many information systems makes this a daunting task without the aid of automation.

A Process for Reengineering

There are three steps in the reengineering process:

- *Detection* - (also called Reverse Engineering) Analyzing source code to identify and extract user interface components from the legacy system.
- *Representation* - Building a model of the existing user interface from the detection step.
- *Transformation* - Manipulating, augmenting, or restructuring the resulting model to allow forward engineering to a graphical environment.

This paper addresses the detection and representation steps, specifically, evaluating representations as the target of the detection process. We also consider some of the transformation issues inherent in adding or modifying user interface functionality once the model is built. Figure 1 describes options in the reengineering process:

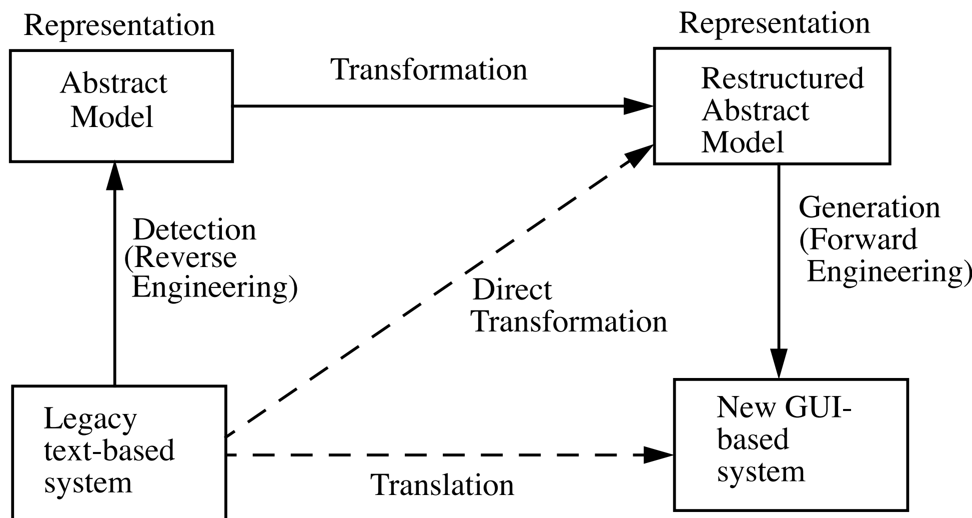


Figure 1 : The Reengineering Process

There are several options for reengineering user interfaces in this process model. The user interface calls in the text-based legacy system could be simply replaced by GUI counterparts, represented by the "translation" edge in Figure 1. There are tools currently available ("screen scrapers") that accomplish this. However, this strategy does not address the necessary shift in control from application-driven to interface-driven (discussed in more detail in the "Modeling Strategy" section below). A better strategy is to extract the user interface from the computational legacy code, using program understanding techniques to build an abstraction, or model, of the existing interface. This path is depicted by the "detection" edge and the "abstract model" box in Figure 1. The application-driven model can then be transformed to an interface-driven model, shown by the "transformation" edge. Once the model has been restructured, a forward-engineering tool, such as a UIMS, could be used to automatically generate a new graphical interface for the system. Another possibility is that the interface-driven model could be directly detected from the legacy code, without modeling the existing structure, shown by the "direct transformation" edge.

A Modeling Strategy

Text-based legacy applications are almost always *Computation-Dominant* as opposed to *Dialog-Dominant*. [HAR89]. This means that the user interface is intertwined in and controlled by the computational algorithms of the application. Most modern GUI interfaces are just the opposite; the user interface dialog structure controls the application (Dialog-Dominant). In order to transition the Computation-Dominant legacy system to the new Dialog-Dominant paradigm, the legacy system must be restructured.

Green's Seeheim model [GRE85] defines an architecture for an interactive system that separates

the computational code from the details of the user interface. The Seeheim model is the basis for many User Interface Management Systems (UIMS), which allow designers to specify the presentation, dialog control, and application interface separately from the computational code. A significant advantage could be gained by reverse engineering an existing text-based system into a representation that could be then transformed into a UIMS representation. Since the legacy application could be maintained separately from its user interface, the user interface could be tailored and updated as needed without having to modify the computational code. Another advantage of separating the user interface from the computational code is in migration to client-server architectures. Reverse engineering to a Seeheim model requires extracting several different models from the original legacy code:

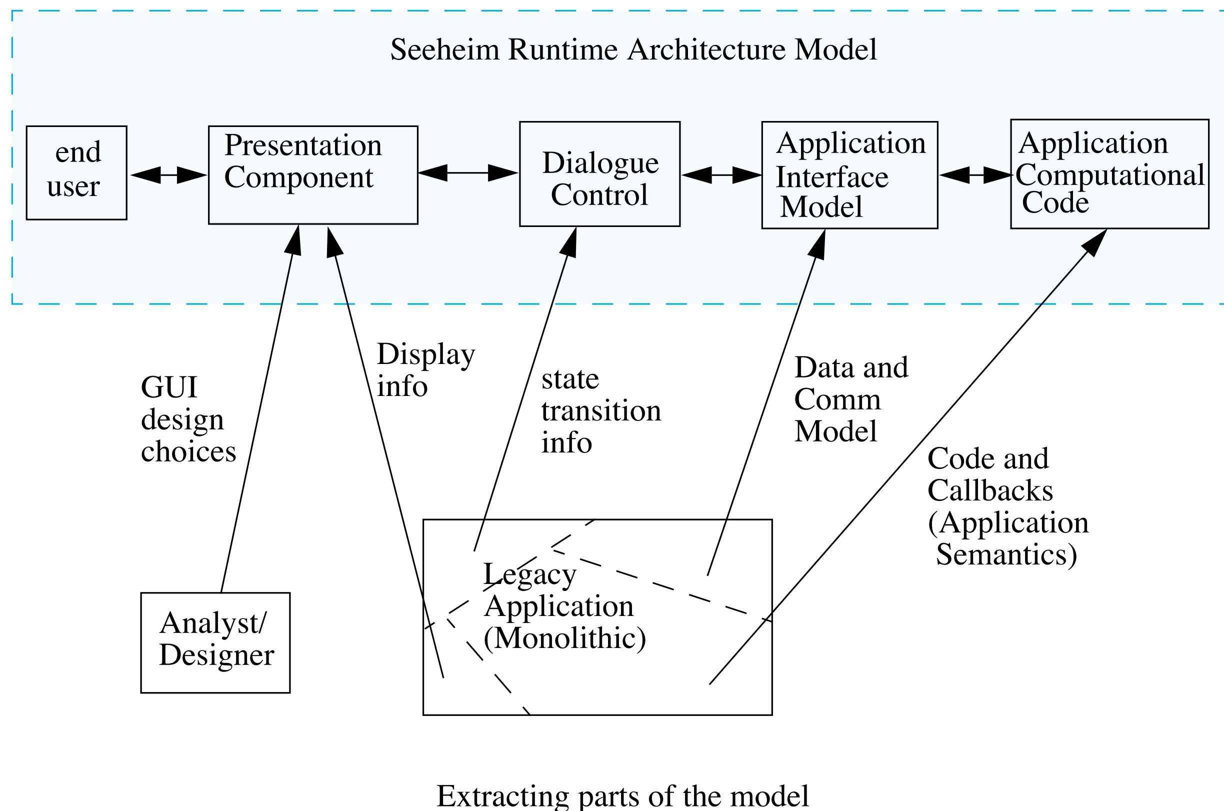


Figure 2 : Reverse Engineering to the Seeheim Model

- The *Presentation Component* of the legacy system includes information such as output groupings (such as in an error message), output-input pairs (such as in a dialog box), contents of screens, and any visible state information. Since the presentation changes drastically between a text-based system and a GUI-based system, the analyst/designer may need to supply further presentation information, such as choosing a selection mechanism (pushbuttons vs. a cascade menu, for example).
- *Dialogue Control* is essentially a “map” of the system, its states and transitions, and also storage for information that allows the interface to be traversed. Sequencing of menus and dialogs are kept in this model, and as a result, control flow analysis can be used to build it.

Preconditions and Postconditions are also maintained here to ensure control to flow only in allowed circumstances.

- The *Application Interface* is the connection between the application and the user interface, essentially a communication model. Values and messages to be displayed are computed in the algorithmic code, and relationships between these “user interface variables” and corresponding user interface objects must be maintained. Slicing and dataflow analysis can be used to extract this model from the legacy code.
- The *Computational Code* is everything that is left when the user interface models have been extracted. Although the computational aspects of the code should not need to be altered, the control structure of the program is likely to be reorganized to fit the new Dialog-Dominant paradigm. The result is that the computational code may be restructured in the form of callbacks that respond to user input events.

The user interface components that can be extracted directly from the legacy code consist of parts of the Presentation component and the Dialogue control component. The Application Interface is built as a result of the reverse engineering step (although information needed to build this model is derived from reverse engineering).

Evaluation Criteria

In light of the discussion above, it is obvious that we will have several different interoperating models as the result of reverse engineering rather than one monolithic one. We must evaluate candidate representations against the requirements of each modeling task to determine which task, if any, the representation is best suited for. This section outlines the evaluation criteria for the Presentation component, the Dialogue Control component, and the Application Interface models.

Presentation Component

The Presentation Component is the “lexical” level of the user interface [GRE87], responsible for direct contact with the end user via input and output devices. Details of the “look and feel” of the interface are preserved here, as well as information about what is displayed at a given point in the interface. A text-based legacy system typically contains little presentation information - except for systems that use simple terminal control semantics (such as the Unix library “curses”). In the transition from text-based to GUI-based structure, the majority of the presentation information will be added by the analyst/designer. However, some presentation information, such as groupings of output statements that belong together in one dialogue box, can be recovered from the text-based legacy system. Following are the requirements of a representation for the Presentation Component:

- The capability to represent and characterize user interface entities or “presentation objects” in abstract terms - for example, to be able to specify that a particular window contains a “selection” without having to specify “group of pushbuttons” or “cascade menu”.
- The ability to define “containers” for presentation objects in order to group them.
- A mapping capability that allows the analyst/designer to choose specific user interface techniques to be associated with the abstract model. For example, mapping “selection” to a Scrolling Textbox widget.
- The ability to express screen layout and organization.
- Interaction with the Application Interface in order to communicate values and selections from

the user to the application program.

Dialogue Control

Dialogue Control is the engine of the user interface, implementing the "syntax" of the interaction [GRE87]. This component contains the "map" of the system, and maintains state information as the user traverses the interface. Dialogue control interprets and handles events, both from the user and from the algorithmic code. Most of the research in user interface representation has addressed the Dialogue Control component of the Seeheim model. Following are the reverse engineering requirements for Dialogue Control:

- Accurately representing the states of the user interface and the transitions that occur as a result of user interface or system action.
- Defining conditions or "guards" and associating them with transitions in order to represent preconditions.
- Focus on describing control paths through the interface.
- Support for asynchronous events, such as an interrupt handler invoked from an escape sequence.
- Understandability - since the analyst/designer will be refining and manipulating the model, it must be able to be expressed in a human-understandable form as well as in a form that can be easily used for analysis. Graphical representations will be preferred for this reason.
- Scalability - Since this method will be used to reengineer very large applications, the resulting model must be hierarchical to allow understanding of the "big picture" as well as the details of a particular state.
- Ability to represent *modal* semantics - Certain choices are available in certain modes and not available in others - represents a restriction of the control paths. This includes maintenance of precondition and postcondition information.

Application Interface

The Application Interface is a communication model between the user interface and the computational code. Relationships between user interface entities and variables in the code must be maintained here, as well as information about entry points into the legacy code (converted to "callbacks"). Following are the requirements of the Application Interface:

- A fairly powerful data model is needed to describe the user interface variables and how they map to entities in the user interface. Information about data organization, types, and values needs to be maintained here.
- Relationships between the data in the application and user interface must be maintained, although these will typically be straightforward mappings rather than complex functions.
- A method for identifying and invoking appropriate computational procedures must be supported, including handling input and output parameters and potentially even side effects. (For example, if a user action invokes a procedure that changes a user interface entity in another window, this needs to be reflected in the user interface.)

Recoverability

Previous experiments with recoverability of user interface components from legacy applications [MOO94a] have shown that a set of recognition rules can be devised to locate user interface pat-

terns in the code. This section lists the user interface constructs that can be detected using reverse engineering techniques, to serve as a framework for evaluating the representation models.

Selection

Selection mechanisms in a text-based interface are much more limited than in a GUI interface. Following are methods that have been detected:

- **Menus** are used to present lists of commands in a hierarchical fashion - submenus are brought up by textual commands, and another command is used to traverse back up the menu hierarchy. Menus have an associated set of choices.
- **Text letters** or words are used to select commands. The user types the command which causes the application to perform the associated action. Typically the user must press an "enter" key to cause the input event, but some systems are implemented in "immediate" mode and the characters are directly sent to the application as they are typed.
- **Numeric values** can be entered to specify quantifiable inputs. These are distinct from character inputs because they may transform to different kinds of presentation in the resulting GUI (for example, a numeric value might be represented by a slider in a GUI, whereas text could not).
- **Cursor position** can be used in a system that contains terminal-control semantics to determine a selection from a list. The selection is actually accomplished by the user pressing the "enter" key, and the choice is determined from the location of the cursor.
- **Interrupts** can be used to select a state that is available from any other state, for example, a "quit" or "return to main menu" escape key that causes the application to abandon the current state and vector to the new state.

State

The notion of continuously displayed state or status is also much more limited in a textual interface than in a GUI. Many text-based applications have states or modes that can be changed by the user (for example, the Unix mailer "elm" is a modal textual interface, having a display mode, a command mode, and a text entry mode). The current mode is indicated by the context (menus or command descriptions present). Text commands and escape sequences are used to change modes. In contrast, GUI interfaces can easily show state, such as the "bold" toggle button in a word processor that changes color when the mode is "on".

Textual Inputs and Outputs

Inputs and outputs in a text-based interface are typically simple character strings. There may be semantics associated with the location of the text on the screen, for example, an error message in elm usually appears on the bottom line of the screen.

- **Prompt - response dialogs** are used to get information from the user. These consist of a series of output statements followed by an associated input statement.
- **Messages** to the user are simply textual outputs and generally appear underneath the command prompt that caused the message. Terminal control semantics may place the message at a specific place on the screen.

Preconditions and Postconditions

Preconditions are simply conditions that must be true in order for the system to change its state. For example, if the system is directed to delete a record in a database, then it must be true that there is at least one record in the database. Postconditions are the results of system transitions,

such as in creating a new record for the database, causing the postcondition of incrementing the number of records. In text-based systems, preconditions are often implemented as conditional expressions that share data objects with associated I/O statements. [RUG95] contains an excellent discussion of a method for extracting preconditions from legacy code. When data objects associated with preconditions are identified, then the postconditions for those data objects can be also detected by dataflow analysis techniques by checking to see how those data values are modified.

Experimental Method

Almost all user interface representations were developed for the task of designing - not reverse engineering (the only exception being AUIDL [MER93]). They are organizational tools designed to focus and formalize a generative effort. The ultimate goal for representation in reverse engineering is to produce a high-level abstraction that adequately models the existing legacy user interface, allows the model to be analyzed and reasoned about, and then allows forward engineering of the new user interface. The major difference between choosing a representation for designing new interfaces and recovering old user interface information is that the existing control structure and the data model in the code both need to be preserved in reverse engineering, otherwise some of the semantics of the user interface may be lost.
In order to fairly evaluate different user interface representations, a small legacy system was chosen to be the subject of experiments in which the user interface model in the legacy system was described in each of several different representations. The resulting models were then evaluated for recoverability - in other words, could the model be obtained from analysis of the source code for the legacy application. The remainder of this paper describes a series of thought experiments in which the legacy system was manually reverse engineered into each of the different representations. The resulting models were then analyzed for recoverability and other evaluative criteria. The generated models and the results of the analysis follow.

A Sample Legacy System

The code used in these experiments is contained in a simple, genuine legacy information system called the “Date-A-Base”. It was developed to assist in a computerized dating service. The system collects data from individuals from a set of questions, maintained in an external file. The user can browse the member database, and can also ask the system to do automatic matching based on a user-supplied threshold of matched answers. The “Date-A-Base” is text-based, menu-driven, and is written in Pascal.

Although the original system does not contain any asynchronous interactions, for the purpose of these experiments an interrupt handler for the SIGINT signal (usually mapped to <cntl>-C on Unix keyboards) was added to the system. (This is representative of a change that might occur in ordinary software maintenance.) Also as part of the thought experiment, we assert that two of the commands, “browse”, and “answer”, can be done in parallel. For example, a user might want to browse an old record while filling out a new questionnaire, thereby revising and updating answers. This possibility of this parallelism should be representable in the notation.

The user starts the system with a command-line command “date”. For a new user (a system state determined by the application, not the user), the menu structure begins with an initial screen:

```
                The
             Date-A-Base

    The Computerized Dating Service

 To use the Date-A-Base you will have to answer a
 personal questionnaire. Your answers to all the
 questions will be available for anyone registered
 in the Date-A-Base to look at.

 Do you want to continue?__
```

This is the first prompt-response dialog with the user. If the user types "Y", then the program continues, reading a set of questions from a file and displaying them on the screen, saving the user's responses. At this point, the main menu is displayed:

```
                    Menu

      [a]   Answer Questionnaire
      [b]   Browse Questionnaire
      [c]   Make a match
      [d]   Delete your questionnaire
      [e]   Quit

      Your choice: __
```

The user can then choose any of the commands, which then display specific menus for the commands, such as the browse menu:

```
    Whose questionnaire do you want to browse?
    ?  __
```

A series of prompts guides the user through the browse process (although there is no way to list the registered users of the database, a functionality that might be added in a reengineering scenario). The other commands are similar, using a series of prompt-response dialogs to accomplish the user's tasks.

Analysis of the Models

This section presents the results of the thought experiments with each of ten user interface representation techniques. They are grouped into the following categories:

- Task Oriented Representations
- Linguistic Representations
- State-Transition Notations
- Production Systems
- Event and Process Algebras

- Formal Model Based Notations
- Knowledge Representations

The first two categories, Task-Oriented and Linguistic representations, are considered "User-Centered" in that they model the interaction of the user with the system from the user's view. They are typically used as evaluative models rather than generative models, and as such are weaker candidates for reverse engineering. The remaining five categories represent "System-Centered" models, presenting a view of the behavior and semantics of the system from inside its components. These, in varying degrees, are more promising as the target of the detection process.

Task Oriented Representations

Goals, Operators, Methods, and Selection (GOMS)

GOMS [CAR83] is an informal task analysis description method with a wide range of applicability. GOMS can express highly abstract goals (such as "wash the dishes") all the way down to the keystroke level ("press the escape key"). Often used in analysis techniques for existing systems, GOMS models the user's interaction with the system, organized in a hierarchical fashion. Following is a GOMS decomposition of the Date-A-Base application:

```
GOAL : ADD-NEW-USER
     GOAL: ANSWER-QUESTIONS  repeat until EOF
*    *    READ-QUESTION
*    *    RESPOND

GOAL : USE-DATABASE
     [select GOAL: ANSWER
*    *    ANSWER-QUESTIONS repeat until EOF
*    *    *    READ-QUESTION
*    *    *    RESPOND
     GOAL : BROWSE
*    *    ASK-WHO
*    *    READ-DISPLAYED-RECORD
     GOAL : MATCH
*    *    GET_PERCENT
*    *    GET_RESULTS ]
```

Figure 3 : GOMS Description

Analysis - GOMS can represent the *control flow* through the system, driven dynamically by the user (although this does not necessarily reflect the system architecture). It is possible that dynamic analysis could be used to generate GOMS hierarchies, but GOMS has many drawbacks for recoverability:

- Goal naming in an automated reverse engineering scenario would be difficult - without user input, goal names could be meaningless.

- There is no way to detect duplicate goals (such as the ANSWER-QUESTIONS goal, which is repeated in USE-DATABASE).
- It also includes physical user actions that are not recoverable from legacy code, such as “READ-DISPLAYED-RECORD”.
- Although multiple selections can be specified, concurrency is not explicitly representable. (For example, the “ANSWER” and “BROWSE” goals can be concurrent, but we cannot show this in the representation.)
- Semantics such as preconditions and postconditions cannot be represented.

Linguistic Representations

Task-Action Grammar (TAG)

TAG [PAY86] is used to predict learning and performance of command-language user interfaces. It uses parameterized grammar rules to enforce consistency in design. Since some text-based interfaces are purely command language driven, TAG is included for completeness. However, since the nature of the legacy system we are reverse engineering is menu-driven, it cannot be represented using TAG. Recoverability of linguistic representations can best be approached with reverse-compiling techniques (recovering the language grammar from a sample of the language).

State-Transition Notations

Finite State Machines (FSMs)

Also called State Transition Networks [DIX93], FSMs are one of the earliest representation methods for describing human-computer interaction. FSMs were originally developed for language processing, and for simple sequential systems, they are easy to generate and to read. However, their inability to represent complex systems, events, and parallelism make them generally inadequate for modern user interfaces. For example, here is the FSM for our Date-A-Base legacy system:

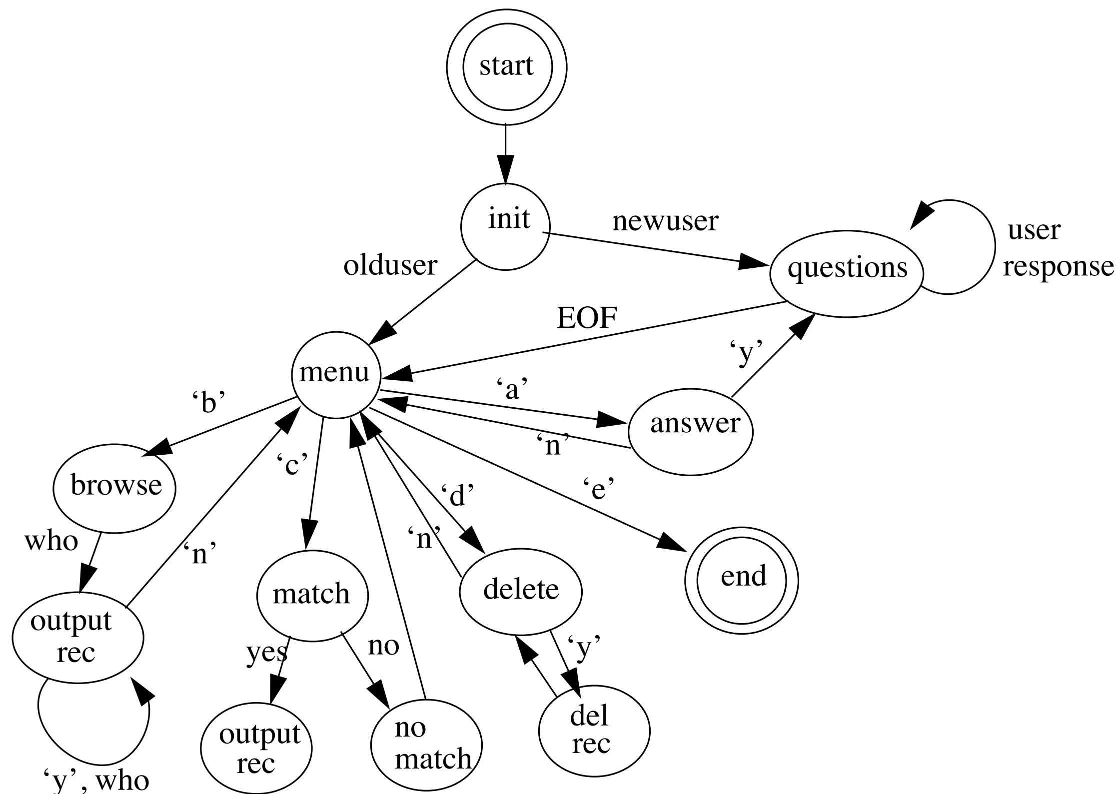


Figure 4 : Finite State Machine

Analysis - While FSM's are strong for representing ordered sequential dialogs, they have many limitations, including:

- There is no convenient way to specify an interrupt handler in a FSM. The only correct way would be to show a transition out of every state into the interrupt handler, which would be unmanageably messy.
- Transitions are initiated both by the system (such as the "newuser/olduser" transition in the initialization state) and by the user. There is no way to differentiate where the transition originated from in a FSM, and if the object of reverse engineering is to identify the user interface components, then information important to the application interface model cannot be represented.
- A purely state-based representation causes the loss of some of the information about the user interface. For example, when the menu state transitions to the browse state, there is a prompt for the name of the person to look for in the database. Since the transition out of the browse state depends on the user response, the original prompt information is lost.
- As with any graphical representation, in order to do automated analysis (and generation of the diagram), there would need to be an intermediate representation of the FSM in some non-graphical form.
- States do not necessarily have user interface actions associated with them - for example, the "delete record" state after confirmation. This mixes some of the computational model in with the user interface model.
- There is no good way to indicate possible concurrency - for example, we would like to repre-

sent that the user can answer questions while using the browse function to look at another record simultaneously. The FSM cannot express this.

- Another well-known problem with FSM's is state explosion on more complex systems - the number of states can grow exponentially if concurrency is modeled [DIX93].

Augmented Transition Networks (ATN), a variant of FSM that adds a stack and conditions to the transitions, allowing preconditions to be specified. However, the conditions do not support post-conditions or solve the other problems with FSMs.

State Charts

Harel's State Charts [HAR87], [HAR88] solve many, but not all, of the problems of FSMs. State charts allow hierarchies to be expressed, as well as concurrency. This allows constructs such as our interrupt handler to be expressed without having an explicit transition from each state to the interrupt handler. (However, resumption from the interrupt would be very hard to express, since the transition depends upon a previous state-transition pair, and State Charts do not have a concept of memory.) They also support the concept of a "broadcast" event, which allows several states to transition in parallel (although this is difficult to see from the diagrams - each transition has to be checked to see if the event that occurs affects it). Following is the State Chart representation of our legacy system, showing the concurrency between the "answer" and "browse" states:

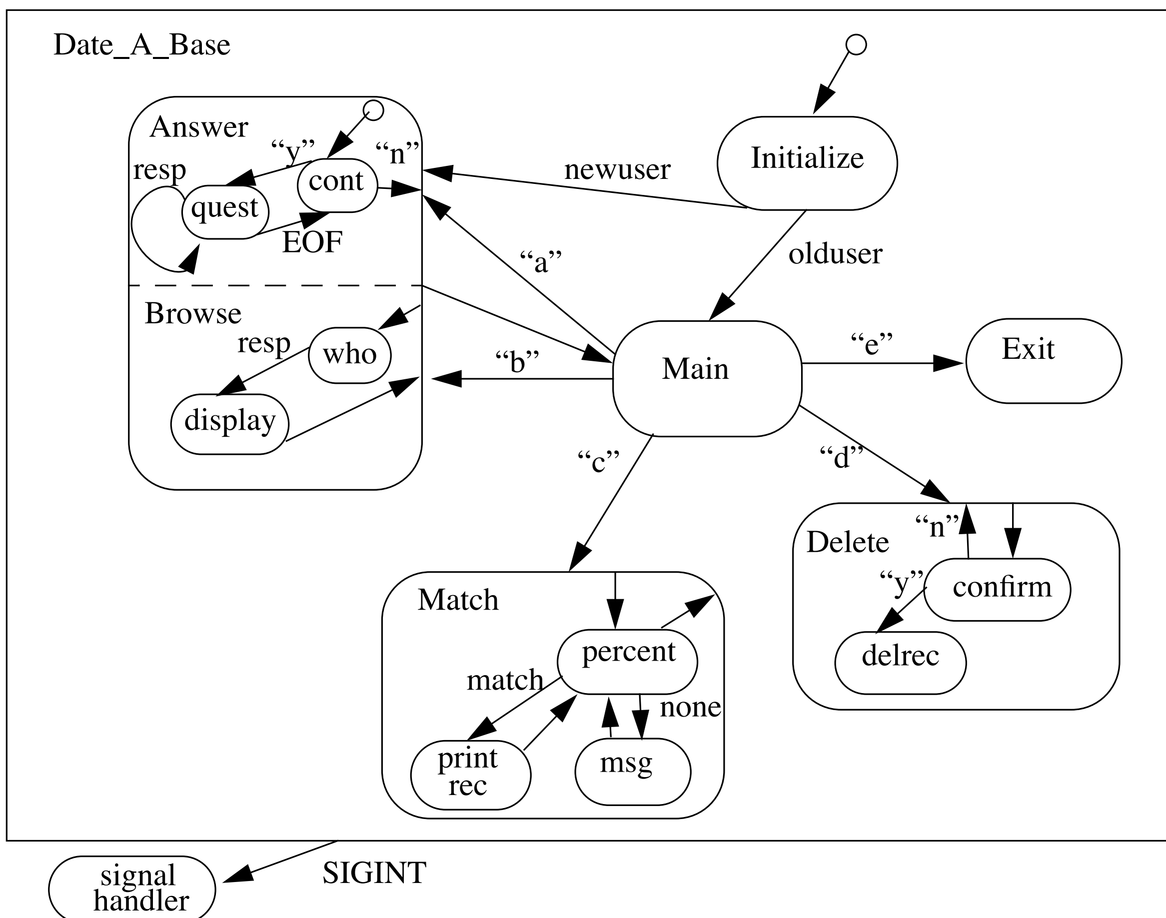


Figure 5 : State Chart

Analysis - State Charts have many strengths for reverse engineering menu-driven interfaces, including:

- The graphical notation is fairly easy to read, and the hierarchical capabilities of the State Charts keep the number of states and transitions to a minimum. A tool with a "zoom" capability would be able to show a big-picture view and also detail without overwhelming the user, even for a large complex system.
- The concurrency of the "browse" and "answer" states can be expressed quite adequately.
- The interrupt handler can be efficiently and succinctly represented as a state that can be transitioned to from any of the other states.
- The sequential nature of the dialogs is preserved, orderings can be easily observed.
- Control-flow analysis can be used to identify the states and transitions in the legacy code.

The limitations of State Charts for recoverability are:

- As a state-based representation, State Charts are subject to some of the same limitations as FSMs. For example, differentiating between user-caused and system-caused events (such as the "new user/old user" transition at initialization). Also, there are user interface states and system states, as in the "delete record" state in the Delete command - this is something accomplished by the application and not neccessarily reflected in the user interface.
- Preconditions cannot be expressed with State Charts - although adding the annotations of ATNs (described above) to each state node could provide this functionality.
- Postconditions (the result of a transition into a new state) also cannot be modeled and would need to be added.
- Since there is no data model in a State Chart, retaining information about the origins of data values in the code would be lost.
- Naming the states and transitions automatically would be a somewhat arbitrary process based on variable and procedure names - a facility to allow the analyst to modify the names would be useful.

<u>Petri Nets</u>

Petri Nets [PET62] were developed for automata theory and have been used for many applications, especially specifying highly parallel systems and real-time systems. The notion of state ("places") and transitions is similar to State Charts, except that the transitions are guarded by conditions that must be true in order for the transition to "fire". Tokens pass through the Petri net to represent control flow.

Analysis - Petri nets are a very powerful concept for highly parallel and concurrent systems, and as a result, are excellent for describing event-driven systems. Describing sequential flow through a Petri Net, while possible, is not as straightforward as State Charts. Since our legacy system model is largely sequential, the expressive power of Petri Nets somewhat wasted. Also, Petri Nets are somewhat more difficult to understand than a State Chart. Petri Nets also do not incorporate a data model or a notion of saved state information. Preconditions for transitions are directly supported in Petri Nets. In [GRE87], Green states that event-based representations are more powerful than pure state-transition networks, and also shows that an equivalent event representation can be derived for any state-transition network. Petri nets do not have a hierarchical structure, and therefore scalability could be a problem, as with FSM's.

Production Systems

Propositional Production Systems (PPS)

A PPS [OLS91] combines the notions of states, transitions, and events. Propositional logic is used to determine which states (using productions defined by a set of rules) should be transformed upon the occurrence of a particular event. A PPS consists of a state space, a set of rules, and a description of semantic actions for the states. Following is a PPS for the Date-A-Base application:

```
State Space:
Field ActiveCommand (AnswerCmd, BrowseCmd, MatchCmd,
                    DeleteCmd, ExitCmd, NoCmd)
Semantic Field CommandSelected (AnswerSel, BrowseSel,
                    MatchSel, DeleteSel, ExitSel, NoSel,
                    QuestionsSel)
Query Field YesNo   (YesAns, NoAns)
Field Action (DrawMain, DrawAnswer, DrawBrowse, DrawMatch)
Input Field Input (NoInput, ‘a’, ‘b’, ‘c’, ‘d’, ‘d’, ‘e’,
                    TextEntry)
Semantic Field Questions (QuestDone, QuestNotDone)

Production Rules:
1. NoCommand, ‘a’ -> AnswerCmd
2. NoCommand, ‘b’ -> BrowseCmd
3. NoCommand, ‘c’ -> MatchCmd
4. NoCommand, ‘d’ -> DeleteCmd
5. NoCommand, ‘e’ -> ExitCmd
6. AnswerCmd, YesAns -> Questions
7. Questions, QuestNotDone -> Questions
8. Questions, QuestDone -> NoCommand
9. AnswerCmd, NoAns -> NoCommand
10. BrowseCmd, GetWho, WhoValid -> DisplayWho, NoCommand
11. MatchCmd, GetPercent -> DoMatch, NoCommand
12. DeleteCmd, YesAns -> DelAction, NoCommand
13. ExitCmd, YesAns -> ExitProg

Semantic Actions  (note: These are all outputs)

NoCommand : Display Main Menu
AnswerCmd : Display Answer Prompt
BrowseCmd : Display Browse Prompt
MatchCmd  : Display Match Prompt
DeleteCmd : Display Delete Prompt
ExitCmd   : Exit System
YesNo     : Display YesNo Prompt
```

Figure 6 : PPS Representation

Analysis - While Production rules are very good for describing concurrency and state transitions, they are not good for describing sequential dialog. The dialog control model is expressed in the PPS, but it is difficult to see. For example, the state "NoCommand" actually is the main menu, which is expressed in the semantic actions, but piecing this together from the production rules takes some study. A PPS can handle interrupts quite easily since it is event-driven. Preconditions also can be implemented through the rules, although the rule syntax does not provide expressions, these could be calculated in the semantic actions. A positive feature of PPS for recoverability is the inclusion of code as the semantic actions; legacy code could be directly tied into the PPS.

Event and Process Algebras

Communicating Sequential Processes (CSP)

CSP [HOA85] has been incorporated into several dialogue specification notations, as described in [ABO90]. It has the advantage of being able to specify sequential dialogue and concurrency. Following is a CSP description of the Date-A-Base application:

```
Init = (olduser [] newuser)
Newuser = Askquestions
Askquestions = Giveprompt; Getanswer? -> Askquestions
               [] EOF -> SKIP
MainMenu = (Answer -> DoAnswer
          [] Browse? -> DoBrowse
          [] Match? -> DoMatch
          [] Delete? -> DoDelete
          [] Exit? -> SKIP
DoAnswer = Askquestion; GetResponse?
DoBrowse = AskWho? -> DisplayRec!
DoMatch = AskWho? -> GetPercent? -> DoMatch
DoDelete = Delete
```

Figure 7 : CSP Model

Analysis - CSP is very effective for modeling events and also for retaining the structure of sequential dialogues. Even repetition can be expressed using recursion, as in the "Askquestions" example above. However, there is no notion of conditions on the states, so preconditions are not represented. Concurrency is inherent but not explicit; it might be difficult for a user to quickly see points of concurrency from the representation. CSP events also do not retain causality, so a system-generated event (such as EOF) is treated the same as user inputs, making it difficult to differentiate. CSP also does not contain a data model, which is why it has been combined with other representations such as Z for user interface specification. Recoverability issues for CSP:

- Determining sequential dialogs can be accomplished with control-flow analysis, so identifying the processes within sequence should be possible.
- With sequential text-based systems the events map to keystroke commands that change the state of the system. This is also detectable with control-flow and dataflow analysis.

Abstract User Interface Description Language (AUIDL)

AUIDL [MER93],[MER95] is actually a hybrid of an object oriented representation combined with Milner's process algebra [MIL89]. The structure of the data model in the legacy system is accomplished with inheritance, and system behavior is described with the process algebra. AUIDL is the only representation that was developed specifically as the target of reverse engineering, and it has successfully been used to represent reverse engineered COBOL programs. Following is a partial AUIDL specification for the Date-A-Base application:

```
instance MACHINE : MAIN
contains
     init; answer; browse; delete; exit;
parameters
     x : char;

behavior
     C = if (Get_X = 'a')
          then
               answer.display
          else if (Get_X = 'b')
          then
               browse.display
          else if (Get_X = 'c')
          then
               match.display
          else if (Get_X = 'd')
          then
               delete.display
          else
               exit

instance ANSWER
```

Figure 8 : AUIDL Specification

Analysis - Since AUIDL was specifically designed for reengineering, it naturally scores highly on our evaluation criteria. The incorporation of the process algebra allows process behaviors to be described and reasoned about, supporting the dialog control model. The behavioral description of the state and transition sequences can be detected by control flow analysis techniques. It incorporates an object-oriented data model that allows presentation information and application interfaces to be specified as well. It supports events from both the system and the user that affect the user interface. It can also handle preconditions that link conditions with behavioral agents, and can keep status information as well. AUIDL is also hierarchical, which makes it good for scalability. Although process algebras can be difficult to read for the analyst, there is also an associated graphical representation for AUIDL.

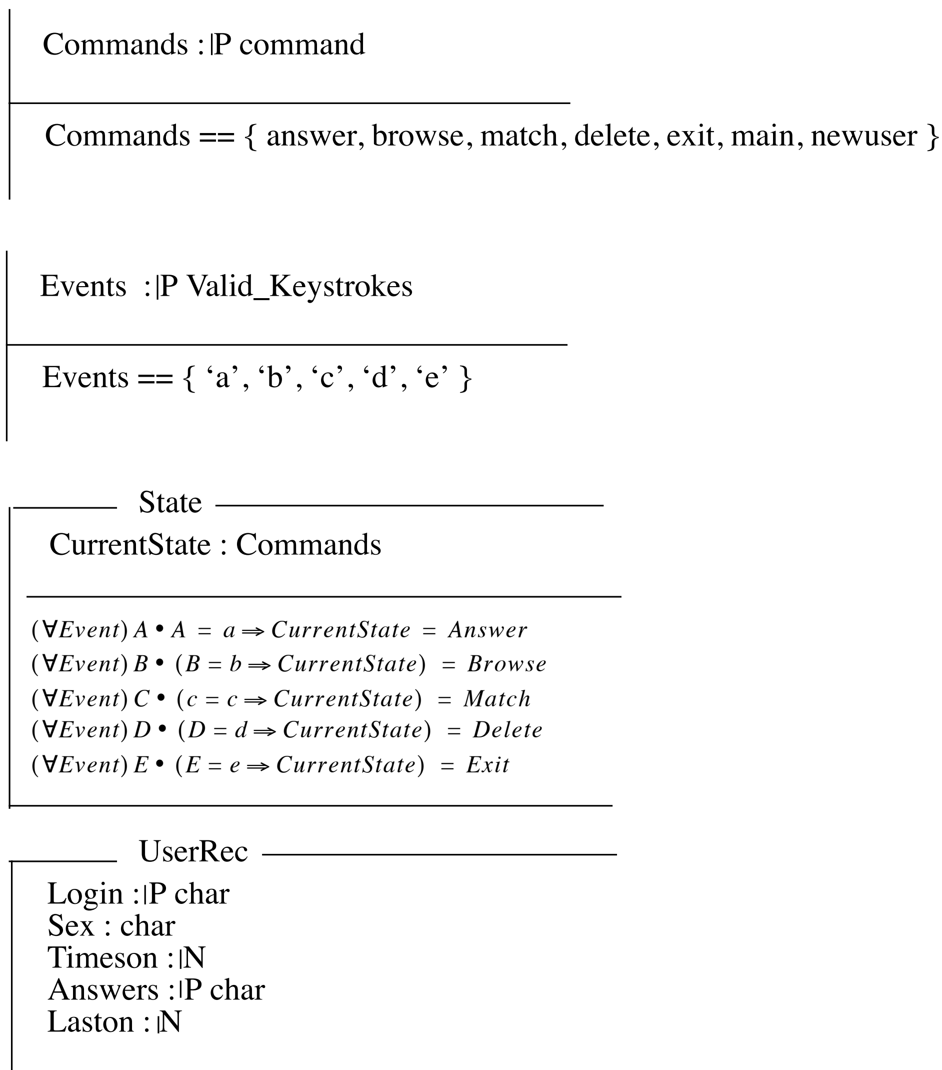
The drawback of AUIDL for our particular purpose is that AUIDL produces a single incorporated

specification for the entire user interface model. The aspects of the Seeheim architectural organization are present, but they are not separated out as we would desire. AUIDL also has a pragmatic drawback in that it is a proprietary representation, owned by a private corporation. This could make it difficult to obtain specifications and tools for AUIDL that might be eventually used for commercial purposes.

Formal Model-Based Notations

Z

The Model-based notations such as Z [SPI88] are based on the definitions of data objects (sets) and functions that act on the data objects. Functions may be used to assign or calculate values, or to provide mappings between objects. Following is a Z representation of the Date-A-Base:



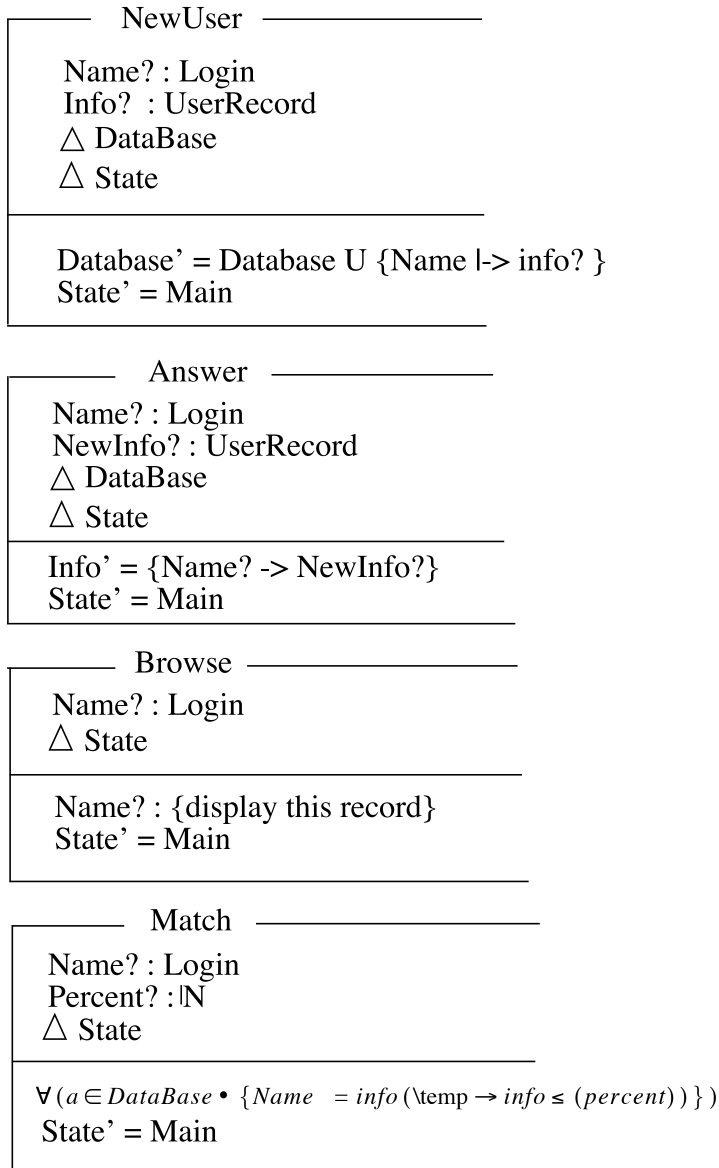


Figure 9 : Partial Z Model

Analysis - Z is a powerful notation for expressing rigorous mathematical specifications of systems. It has been incorporated into user interface representations in conjunction with event models [ABO91]. As seen in the example above, Z is capable of richly defining the data model of the system and part of the control flow. However, the obvious difficulty of using pure Z is in representing events and concurrency. Sequential dialog can be expressed, but in general the mathematical notation of Z requires a significant background in discrete mathematics that may be beyond the capabilities of the average systems analyst. There are also several recoverability issues for Z:

- User Interface variables and types are a matter of simple translation from declarations in the legacy code. However, identifying “arity” from legacy code - which variables denote single instances, and which denote sets - may not be trivial. Especially if variables are dynamically allocated, dynamic analysis may be required to determine whether a given variable is a set. One assumption that could be made is that anonymously-typed variables are singletons, but any variable declared from a type definition could be a set.
- Detecting the existence of a relationship between variables can be done with slicing and data-flow analysis, but characterizing the relationship would be very difficult. For example, it would be very hard to tell if a relationship was a partial injection vs. a true one-to-one onto relationship without extensive dynamic analysis.
- Detection of functions requires extensive understanding of program semantics which may not be possible with current recovery techniques.
- Constraints on values could be detected from declarations in the legacy code, but this is not guaranteed - much range constraint checking is done in the computational code. Range-constraints could be detected with cliches, but if filters are implemented in the code it could confuse the recognizer.

In general, legacy code contains too little information to recover significant Z specifications with current reverse engineering technology. A small subset of the Z notation could be recoverable, but the full power of Z could not be utilized with purely automated detection (analyst input could add human judgement and detail to the specifications). Z is weak for describing dialog control, but it would be a powerful notation for describing the data model of the system, used both in the presentation component and in the application interface. Z is relatively difficult to learn as well, requiring the analyst to have an extensive knowledge of discrete mathematical notations.

Knowledge Representations

User Interface Development Environment (UIDE)

A possible candidate for the target of reverse engineering that is not as ubiquitous in the HCI literature is Knowledge Representations. The UIDE system [FOL91] uses a series of knowledge schemata to represent objects and their attributes, actions and associated parameters, preconditions, and postconditions. With this design knowledge at the base of the system, the user interface can be generated automatically. It also allows the automated generation of context-sensitive automated help.

Analysis - The UIDE representation is organized around an object-oriented paradigm. The majority of text-based legacy systems are organized in a structured manner, and a shift to objects would require restructuring the code. There is ongoing research in the area of object detection [BAI90],[GAR90], mostly with the goal of extracting reusable components from legacy code. These techniques are based on identifying key variables as the objects, and then tracing parameters types in the functions to group them and to associate actions with the data objects. While this has met with some success as an extraction process for individual objects, whether they could be used to restructure an entire program to an object-oriented architecture would require experimentation.

The UIDE knowledge base maintains a great deal of semantic information about the objects in the action schema descriptions. This feature would make it a possibility for describing the application interface. The knowledge representation itself does not describe the dialog control function

in UIDE, that implemented in a separate module. Hence, the knowledge representation would not be a good candidate for the Seeheim dialog control model. For presentation semantics and application interface components, however, a knowledge representation scheme would be a good candidate.

Summary of Findings

In order to get a "big picture" view, we can compose a matrix of results from our thought experiments. The following matrix (Table 1) summarizes the characteristics of each representation technique according to our evaluation criteria. Each representation is rated on a scale of low to high for each of the criteria.

Table 1: Comparison of Representation Features

Representation	Recoverability	Human Understanding	Scalability	Expressive Power	Concurrency
GOMS	low	high	high	med	yes
TAG	N/A				
FSMs	high	high	low	low	no
State Charts	high	high	high	high*	yes
PPS	med	med	med	med	yes
Petri Nets	med	med	low	med	yes
Z	low	low	low	high	no
CSP	med	med	low	med	yes
AUIDL	high	med/high	high	med	yes
Knowledge Rep (UIDE)	low	med	high	med	yes

* Assuming annotations (such as in ATN) have been added to represent preconditions.

The next table (Table 2) indicates which of the three models, Presentation, Dialogue Control, or Application Interface, a particular representation might be appropriate for. "Yes" means that the representation could be applied without modification, "no" means that the representation would not be a *good* candidate for that model (even though it might be possible to describe the model in that representation). "Maybe" means that parts of the model, but not the entire model, could be expressed in that representation:

Table 2: Applicability of Representations

Representa-tion	Presentation Component	Dialog Control	Application Interface
GOMS	no	maybe	no
TAG	no	no	no
FSMs	maybe	yes	no
State Charts	maybe	yes	no
PPS	no	yes	no
Petri Nets	no	yes	no
Z	maybe	no	yes
CSP	no	yes	no
AUIDL	yes	yes	yes
Knowledge Rep (UIDE)	yes	no	yes

Conclusions

It is clear from observing and experimenting with user interface representation techniques that no one technique is sufficient to fully describe the models generated from reverse engineering. Not surprisingly, the AUIDL specification language, developed expressly for reengineering, fulfills the bulk of the requirements. However, even AUIDL does not separate out the components of the models in order to describe a Seeheim architecture for the new system. Therefore, the solution to our problem lies in developing a hybrid representation, combining different techniques for different models in the architecture.

References

[ABO89] Abowd, Gregory, and Bowen, Jonathan, *User Interface Languages: A Survey of Existing Methods,* Technical Report PRG-TR-5-89, Programming Research Group, Oxford University, October 1989.

[ABO90] Abowd, Gregory D. "Agents: Recognition and Interaction Models", *Proceedings of Interact '90,* North-Holland, Amsterdam, 1990.

[ABO91] Abowd, Gregory D. *Formal Aspects of Human-Computer Interaction*, Ph.D. Thesis, Oxford University Computing Laboratory Programming Research Group, 11 Keble Road, Oxford OX13QD, England, 1991.

[BAI90] Bailey, John W., and Basili, Victor. "Software Reclamation: Improving Post-Development Reusability", in *Software Reengineering,* by Robert Arnold, IEEE Computer Society, 1993.

[CAR83] Card, S.K.; Moran, T.P.; and Newell, A. *The Psychology of Human Computer Interaction.*, Lawrence Erlbaum Associates Publishers, 1983.

[DIX93] Dix, Alan; Finlay, Janet; Abowd, Gregory; and Beale, Russell. *Human-Computer Interaction*, Prentice Hall International (UK) Limited, 1993.

[FOL90] Foley, James D., van Dam, Andries, Feiner, Steven K., and Hughes, John F. *Computer Graphics Principles and Practice*, Second Edition, Addison-Wesley Publishing Company, Addison-Wesley Systems Programming Series, 1990.

[FOL91] Foley, James; Kim, Won Chul; Kovacevic, Srdjan; and Murray, Kevin. "UIDE - an Intelligent User Interface Design Environment", in *Intelligent User Interfaces*, Edited by Sullivan and Tyler, ACM Press, 1991.

[GAR90] Garnett, E.S., and Mariani, J.A., "Software Reclamation", in *Software Reengineering,* by Robert Arnold, IEEE Computer Society, 1993.

[GRE85] Green, Mark. "The University of Alberta User Interface Management System", *Proceedings of SIGGRAPH, 12th Annual Conference*, San Francisco, CA, July 1985.

[GRE87] Green, Mark. "A Survey of Three Dialogue Models", *ACM Transactions on Graphics*, Vol. 5, No. 3, July 1986.

[HAR87] Harel, David; Pnueli, A.; Schmidt, J.P.; Sherman, R. "On the Formal Semantics of Statecharts", *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, IEEE, 1987.

[HAR88] Harel, David. "On Visual Formalisms", *Communications of the ACM*, Volume 31, Number 5, May 1988.

[HAR89] Hartson, H. Rex, and Hix, Deborah. "Human-Computer Interface Development: Concepts and Systems for Its Management", ACM Computing Surveys, Vol 21., No. 1, March 1989.

[HOA85] Hoare, C.A.R. *Communicating Sequential Processes,* Prentice-Hall International, Englewood Cliffs, N.J., 1985.

[MER93] Merlo, E.; Girard, J.F.; Kontogiannis, K.; Panangaden, P.; and De Mori, R., "Reverse Engineering of User Interfaces" Proceedings of the Working Conference on Reverse Engineering, Baltimore, MD, May 21-23, 1993.

[MER95] Merlo, Ettore; Gagne, Pierre-Yves; Girard, Jean-Francois; Kontogiannis, Kostas; Hendren, Laurie; Panangaden, Prakash, and DeMori, Renato; "Reengineering User Interfaces" *IEEE Software*, Vol. 12 No. 1, January 1995.

[MIL89] Milner, Robin. *Communication and Concurrency,* Prentice Hall International Series in Computer Science, 1989.

[MOO94a] Moore, Melody. “A Technique for Reverse Engineering User Interfaces”, *Proceedings of the 1994 Reverse Engineering Forum*, Victoria, B.C., Sept 1994.

[MOO94b] Moore, Melody, Rugaber, Spencer, and Seaver, Phil, “Experience Report: Knowledge-Based User Interface Migration”, *Proceedings of the 1994 International Conference on Software Maintenance*, Victoria, B.C., Sept 1994.

[OLS91] Olsen, Dan. *User Interface Management Systems: Models and Algorithms*, Morgan Kaufman Publishers, 1991.

[PAY86] Payne, S.J., and Green, T. “Task-Action Grammars: A Model of Mental Representation of Task Languages”, *Human-Computer Interaction*, Vol 2, No.2, 1986.

[PET62] Petri, C.A., “Kommunikation mit Automaten”, Ph.D. Dissertation, University of Bonn, Germany, 1962.

[RUG93] Rugaber, Spencer, and Clayton, Richard. “The Representation Problem in Reverse Engineering”, *Proceedings of the Working Conference on Reverse Engineering* May 21-23 1993, Baltimore, MD. IEEE Computer Society Press, 1993.

[RUG95] Rugaber, Spencer; Stirewalt, Kurt; and Wills, Linda. “Detecting Interleaving”, *Proceedings of the International Conference on Software Maintenance*, Nice, France, 1995.

[SPI88] Spivey, J.M. *The Z Notation: A Reference Manual*. Prentice-Hall International, Hemel Hempstead, 1988.