ISCS - A TOOL KIT FOR CONSTRUCTING KNOWLEDGE-BASED SYSTEM CONFIGURATORS

Harry Wu, Hon Wai Chun, Alejandro Mimo

Honeywell Information Systems
300 Concord Rd., MS895A
Billerica, MA 01821.
617-671-3663

ABSTRACT

This paper describes an integrated programming environment which is specially tailored to the development of knowledge-based system configurators. It is designed with three major objectives: to provide an integrated representational framework for the various knowledge sources relevant to the configuration task, to assist a knowledge engineer in the development and administration of the knowledge base, and to aid a knowledge engineer in the actual construction of a system configurator. Particular attention is placed on the engineering aspects of the system construction process. Specifically, we describe how knowledge acquisition is eased through the use of a configuration language which is specially designed to represent the various knowledge sources for configuration, how knowledge encoding and modification may be aided by the knowledge engineer assistant module, and how the development of user interface may be aided by a generic user interface generator. A prototype of the special purpose environment is implemented on a XEROX 1108 workstation and is being used to develop configuration expert systems at Honeywell.

I. INTRODUCTION

System configuration is the process by which a formalized description of a computer system and its interconnected components is produced based on an initial user account of the individual components. In recent years, expert system technology has been applied successfully to the construction of automatic system configurators such as R1/XCON (McDermott, 1982), OCEAN (Szolovits, 1985), ISC (Wu, 1985), and SYSCON (Rolston, 1985). Progress has been made in understanding the process of configuration and in identifying the general representational and functional requirements of these systems. This paper describes an intelligent system configuration shell (ISCS) which is specially designed as a tool-kit to assist knowledge engineers in the construction and maintenance of system configurators. The power of the tool kit is derived from AI and software engineering techniques, leveraging on the accumulated knowledge on configuration.

In recent years, many major advances in knowledge-based or expert system technology have been made (Hayes-Roth, 1983). In particular, certain features of knowledge representations (e.g. frame, rule, demon, etc.) and inference mechanisms (e.g. agenda, forward and backward chaining, etc.) have become the building blocks of many commercially available tools (shells) for developing knowledge-based systems (Richer, 1985). These commercial AI tools provide excellent development environments for experienced AI programmers but not domain experts; they are general purpose in nature but not problem oriented. The goal of our project is to adapt the existing INTERLISP-D and LOOPS (a multi-paradigm language -- Bobrow, 1981) environment on the XEROX 1108 workstation into one which is specially tailored for the configuration problem. By specialization to a particular problem domain, we created a shell which is much more convenient to use and provides better integration of the various concepts and control mechanisms required for this problem domain.

While the basic theories are interesting on their own and fundamental to AI, it is often the programming aspects that make the AI tools attractive to the practitioners. Some people even argued that a significant part of the applicability of AI derives not from the AI techniques per se but from the underlying software technology (Sheil, 1983). Therefore in addition to knowledge engineering techniques, the ISCS shell also contains several additional software features which facilitate the process of constructing system configurators.

The ISCS system has three main objectives: (1) to provide an integrated representational framework for the various knowledge sources relevant to the configuration problem, (2) to assist a knowledge engineer in the development and administration of the knowledge base, and (3) to aid a knowledge engineer in the construction of a system configurator. A prototype of the system is implemented in INTERLISP-D and LOOPS on a XEROX 1108. The shell is being used to construct configuration expert systems at Honeywell.

In the following sections, we describe how knowledge acquisition is eased through the use of a configuration language which is specially designed to represent the various knowledge sources for configuration, how knowledge encoding and modification may be aided by the knowledge engineer assistant module and how the development of user interface may be aided by a generic user interface generator. Implementation details are also given in the discussion.

II. CONFIGURATION

The task of configuration is one which selects and organizes objects into some system so that, functioning as a whole, it produces certain desired system behavior. In this paper, we restrict the task to computer system configuration; that is, we are only interested in selecting and organizing devices and components to form a computer system. A configuration task may choose to address only software components, only hardware

components, or both; we describe them as software configuration, hardware configuration, and integrated system configuration respectively.

Configuration activities are traditionally carried out by different organizations within a computer company for different purposes. For instance, sales representatives "configure" computer systems to fit the needs of customers while field engineers "configure" computer systems to fit physical installation requirements. R1 (McDermott, 1982) was developed before XSEL (McDermott, 1982) and thus are sometimes viewed as separate systems. In reality, the two activities complement each other. The difficulty of one activity may be reduced if knowledge relevant in the other activity can be made available; usually the sales representatives and field engineers maintain contact with each other in order to consummate an order. The design of ISCS is such that it may be used to construct a *sales* configurator, an *engineer* configurator, or one which encompasses both activities.

In order to come up with a shell for the system configuration problem domain, one must first identify the tasks and knowledge common to system configuration problems in general and then decide how they might be represented conveniently. This problem is first studied by McDermott in his seminal paper (McDermott, 1982) where he demonstrated that this can be solved by a rule-based system using only forward chaining. In a later system, OCEAN (Szolovits, 1985), the developers used a richer environment involving hybrid representations. Our approach is similar to that of OCEAN. While a knowledge-based configurator system can be implemented by forward rules alone, we feel that by using a richer environment and representation, knowledge encoding is made conceptually closer to actual expert knowledge and hence easier to construct. In particular, maintenance of a knowledge base is made simpler by having modular sources of configuration knowledge.

One may view the configuration process as consisting of a data entry and value validation phase, a completion phase, and an assignment phase. The process usually starts with the specification of the components to be included in the target system and site-specific information. A configuration expert sorts through the mass of data to validate the values and to see if the information is complete. If components are missing, they are added to the system. Finally, the expert attempts to build a viable configuration which satisfies the site-specific requirements. The output from the configuration is a highly detailed and formalized document recording the spatial, or functional layout of the components. The configuration task is usually tedious and error-prone due to the large amount of information involved. Moreover, the information changes with time because old components and systems become obsolete while new ones are introduced. The task is also complicated by the fact that the initial specification is seldom accurate or complete many iterations of the configuration process actually take place between the time a sales proposal is first initiated till the system is delivered and installed.

Several sources of knowledge are required in the system configuration process: knowledge about the individual components, knowledge about the inter-relationships among components and between components and systems, knowledge about the configuration procedures, and knowledge about the format of the output description. In ISCS, separate knowledge representations are used to capture these knowledge sources. Although some of these ideas have been studied and advocated elsewhere (Hayes-Roth, 1983), our goal is to illustrate how they may be packaged and utilized in the domain of system configuration. In this section, we provide a brief conceptual overview of three of the more interesting knowledge sources - the *component model*, the *constraint demons*, and the *configuration knowledge*. Details on each knowledge source are provided in the next section.

Information about components and system types are described by the component model which defines the attributes and options of the components and systems. The component model constitutes a *taxonomic* hierarchy of class and sub-classes and a *part-of* hierarchy to depict the relationship between parts and sub-parts. Properties may be inherited along the descendant path within a hierarchy.

Constraints and dependencies may arise when different types of components are assembled together into a system. Constraint demons are provided to capture this type of dependency knowledge. Constraint demons have two basic constituents: descriptive predicates and sets of imperative actions. The predicates define the constraints and the context under which the demons should be triggered, while the actions indicate the activities (usually remedial corrections) to follow.

The configuration knowledge consists of a hierarchy of procedural knowledge involved in configuration. At the bottom level are *operations* where each operation acts on a set of objects of the same component type. For example, an operation may be selecting an object from a set, or modifying every object in a set. At the next level are *tasks* which are composed of sequences of operations (hence tasks may act on objects of different component types). For instance, a task may act first on an aggregate and then its sub-parts by a sequence of operations along a "part-of" path. Above the tasks are *plans* which decide which tasks to execute.

III. ISCS

Although there are a large variety of tools and knowledge representations embedded in ISCS, integration and coherence are achieved by relying on a object-oriented programming paradigm. A similar strategy has been adopted in another tool-kit (Lafue, 1985). Figure 1 provides a schematic diagram of ISCS. In the middle of the figure are the modules of the shell consisting of a knowledge engineer assistant module, a control and inference module, an user interface generator, and a site database manager. The knowledge engineer assistant, KEA, provides "structure-based" editors for the various types of knowledge representations. It is the main vehicle for knowledge entry and maintenance. The KEA also interprets knowledge structures created through the ISCS configuration language into the internal representations. The control and inference module, CI, decides how a configuration session proceeds by looking up site-specific data from the data base and system-specific configuration knowledge from the knowledge base. The user interface generator, WSI, is a tool that allows a knowledge engineer to create and customize an end-user interface. The database manager handles the storage and retrieval of site-specific data.

A. ISCS knowledge sources

This subsection presents the major knowledge sources in ISCS. These knowledge sources correspond to the various types of knowledge required in the configuration task.

1. The Component Model

The component model captures information about individual systems and components. Each type of system or component is defined by a *class*, i.e. "prototype record", which contains *variables*, i.e. "attributes", related to that object type. *Instances*, i.e. "objects", of the same class will have identical record structure. Two kinds of relationships may be specified between classes.

The taxonomic hierarchy: In ISCS, this class-subclass hierarchy permits the knowledge engineer to define paths for class inheritance. For example, in figure 2, the "7301" class is a subclass of "synchronous terminal" class which in turn is a subclass of "terminal" class. All properties, such as standard 1200 baud rate, about a terminal will also be inherited by a 7301 terminal unless indicated otherwise.

The part-of hierarchy: Variables of a class may possess the "contain-parts" property which is a list of class names. Instances of classes in the list are considered sub-parts. For instance, the example in figure 2 illustrates that terminals of type 7301, 7303, or 7305 may be hung on a synchronous communication line. The "part-of" and "contain-parts" are inverse relationships; as soon as one direction is defined, the other direction will be automatically added. The variable name in the other direction is given by adding a suffix "-INV" to the variable name. The value for either the part-of or contain-parts relation is always a list to indicate an one-to-many relationship. Values may be inherited along the part-of hierarchy as well. In the example of figure 2, the channel number of a 7301 terminal is obtained from the synchronous communication line it is attached to. Unlike the taxonomic case where inheritance is at the *class* level, the part-of inheritance is at the *instance* level; values are automatically copied when two objects are linked together by an "AddTo" operation; e.g. (AddTo <an instance of SynchronousCommunicationLine> 'TerminalAttached <an instance of T7303>).

```
(CLASS Terminal
        ...        (BaudRate 1200)        ... )

(CLASS SynchronousTerminal
        ...        (Super Terminal)        ... )

(CLASS T7301
        (Super SynchronousTerminal)
        (TerminalAttached-INV NIL
                Part-Of
                (SynchronousCommunicationLine))
        (ChannelNumber
                (Inherit TerminalAttached-INV))    ... )

(CLASS SynchronousCommunicationLine
        ...
        (TerminalAttached NIL
                Contain-Parts (T7301 T7305 T7307))
        (ChannelNumber NIL)    ... )
```

Figure 2. Example of taxonomic and part-of hierarchy.

2. The Constraint Demons

Constraint demons are the means by which a knowledge engineer may specify restricted conditions and remedial corrections if these conditions are violated. A constraint demon has three properties: scope, predicate,

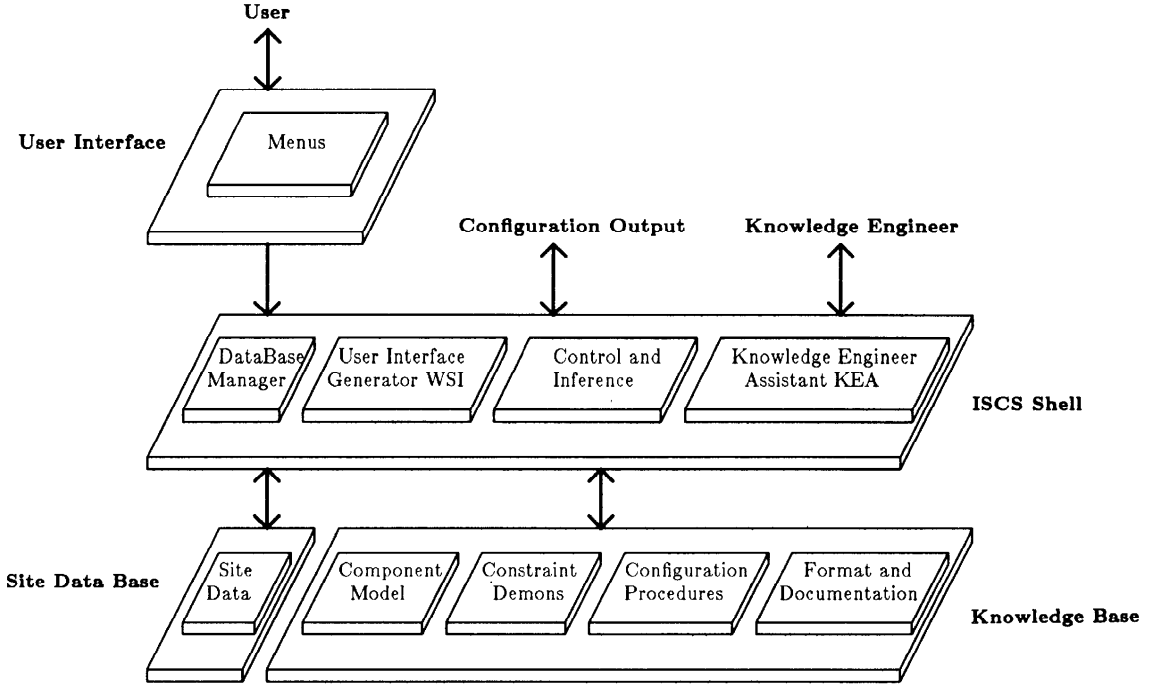

Figure 1. Schematic diagram of ISCS and its environment.

and action. Currently ISCS support two types of demons which are "triggered" under different circumstances. For a *value demon*, the corresponding predicate is tested at the time when a value is being stored into an instance variable of an object. For a *class demon* the corresponding predicate is tested every time a new instance from the (immediate) class is created. If the predicate evaluates to false then the actions are executed. The scope property of a demon indicates whether it is a value or class demon. The action property contains a list of actions that may be performed if the predicate is not satisfied. The conditional portion of the action specification allows a knowledge engineer to select appropriate actions by context. When a demons's predicate is found to be false, each action is then carried out sequentially. The variable "Predicate" is bound to the result of the evaluation of the demon's predicate and is made available to all the actions. Constraint propagation may be explicitly turn on or off for each demon action.

The first demon in figure 3 validates baud rates. The demon will check values that are to be stored into either a 7307 or 7309 terminal. The legitimate values are 1200, 4800, and 9600. There are two actions in the example. One of them will be executed during input phase when a user enters data into the system interactively, the other will be executed during the completion phase when the system automatically complete all necessary information. The second demon makes sure that there are enough synchronous communication lines for 7307 terminals given the fact that at most five 7307's can be hung from one line. If not enough lines are there, then a new one is added. In this case, constraint propagation is turned on because there might be other demons monitoring the creation of synchronous communication lines.

```
(DEMON CheckBaudRate
        (* 7307 & 7309 baud rates)
        (Scope    Value     (T7307 BaudRate)
                            (T7309 BaudRate))
        (Predicate  (MEMBER (@ BaudRate)
                            (1200 4800 9600)))
        (Actions  (IF InputPhase THEN
                            (<-@ BaudRate NIL)
                            (PROMPTPRINT <message>))
                  (IF CompletionPhase THEN
                            (<-@ BaudRate 1200)) ))

(DEMON LineForTerminalExists?
        (* max. of five 7307 terminals per syn comm line)
        (Scope    Class     (T7307))
        (Predicate  (LESS  (NumberOf
                            SynchronousCommunicationLine)
                            (/ (NumberOf T7307) 5)))
        (Actions  (IF InputPhase THEN
                            (PROMPTPRINT <message>))
                  (IF CompletionPhase THEN
                            (DemonPropagation 'On)
                            (CreateInstance
                            'SynchronousCommunicationLine))))
```

Figure 3. Example of constraint demons

3. The configuration procedural knowledge

ISCS provides a mean to decompose the procedural aspects of configuration into more manageable units. These units are organized in a hierarchy of three layers:

Operation: An operation applies the same procedure to all the objects of the same class; e.g. send the same message to every objects of the same class. In addition, there is a preconditional predicate which is evaluated before an object is operated on. If the predicate fails then that particular object is skipped. The number of objects that are to be tested and worked on depends on the "Repeat" attribute (see figure 4). There are two automatically generated variables associated with each operation. After the execution of an operation, these two variables will contain respectively the objects that have been worked on or not. Figure 4 shows an operation which iterates through all 7307 terminals and assign each one to a communication line. In the example the variable "self" is bound to a different object during each iteration. After the operation, the variables,
"T7307.AssignTerminalToCommLine.Done"
and "T7307.AssignTerminalToCommLine.NotDone",
will contain respectively the terminals that have been attached and those that are still loose.

```
(OPERATION AssignTerminalToCommLine
        (* attach each 7307 terminal to a comm line)
        (Class T7307)
        (Predicate T)
        (Repeat UntilExhausted)
        (Code (AddTo(GetAnInstance  SynchronousCommunicationLine)
                'TerminalAttached self))))

(HIERARCHICAL-TASK Documentation
        (* Output each comm line and its terminals)
        (Root SynchronousCommunicationLine)
        (Paths (SynchronousCommunicationLine
                TerminalAttached))
        (Code   ((Class  SynchronousCommunicationLine)
                 (IF <predicate> THEN (Document))
                 ((Class T7307)
                  (IF T THEN  (Document)))))

(ITERATIVE-TASK PutCardsOnBoards
        (* assign all cards to boards)
        (Type ALL)
        (Predicate (AND (NULL Type-1-Card.NotDone)
                        (NULL Type-2-Card.NOtDone)))
        (Epilog (IF (OR Type-1-Card.NotDone
                        Type-2-Card.NOtDone)
                    THEN (Createinstance Board)))
        (Variables (NewBoard))
        (Rules
          (R1:(SETQ NewBoard(Operation Board GetABoard)))
          (R2:(Operation Type-1-Card AllocateSlot NewBoard))
          (R3:(Operation Type-2-Card AllocateSlot NewBoard))))

(OPERATION GetABoard
        (* always returns the first board still in the "NotDone" list)
        (Class Board)
        (Predicate T)
        (Repeat 1)
        (Code (Return self)))

(OPERATION AllocateSlot
        (* Assign as many cards to a board as possible)
        (Class Type-2-Card)
        (Repeat UntilExhausted)
        (Predicate (Type-1-Card-Fit? NewBoard))
        (Code (Decrement-slot NewBoard)
                (Fix-Slot self NewBoard)))

(PLAN Card-type-1-Or-Card-type-2
        (* interchange r2 and r3 in task PutCardsOnBoards)
        (Rules (MyRule: (If <predicate> THEN
                        (XChange PutCardsOnBoards R2: R3:)))))
```

Figure 4. Example of configuration knowledge

Task: A task enables a knowledge engineer to perform "aggregate" work by invoking individual operations. It is especially useful when operations, acting on different classes of objects, achieve jointly a common objective. There are two types of tasks: hierarchical and iterative.

A hierarchical task is used to traverse a "part-of" hierarchy and work on each object in this hierarchy. An iterative task is used to carry out a sequence of operations repeatedly. The second example, in figure 4, is a hierarchical task which traverses the communication lines and the terminals attached to each line in pre-order; i.e. visit a communication line and then all its terminals before visiting the next communication line. As shown in the example, a hierarchical task is similar to an operation except for the number of classes involved and for the order in which the objects are fetched. "Pruning", i.e. the decision whether to follow a branch, is decided by the IF conditions. If an object fails its own predicate test then none of its descendants will be tested.

An iterative task is similar to a LOOPS ruleset in concept in that rules are partitioned according to the objects that they affect. An iterative task is composed of a list of IF-THEN rules and operations. There are two mode of iteration which dictates whether all rules are executed or only one rule is executed during each pass of the iteration. A knowledge engineer supplies a condition for terminating the loop; the default is to stop the task when all the rules and operations have failed. The rules and operations in an iterative task are scanned by their sequential ordering in the task. An optional piece of code, "Epilog" may be executed after each iteration before the start of the next. This Epilog can look at the "trace" of the previous iteration and make corrections and preparations for the next pass. Each iterative task only looks at certain classes of objects. For each operation, there are two values to indicate the objects that have been worked on and those that have not. These two values are not reset after an iteration (unless explicitly stated in the EPILOG); in fact when an operations appears in an iterative task, it will only look at the objects that have not yet been worked on. If new objects are created, within or without an iterative task, they are automatically appended to the appropriate lists so that the operations can work on them in a subsequent iteration. Figure 4 contains an example where cards are being assigned into slots on a board. If there are not enough boards to fit all type-1 and type-2 cards then a new one is added. The procedure repeats until all cards have been assigned to boards. Note that the Epilog checks to see if more boards are needed and creates one on demand so as to ensure some progress in the following iteration.

Plan: Plans decide which tasks are be executed as well as manipulate the tasks themselves. Each rule in a task is treated as a named object and thus can be manipulated. Plans can remove, replace, or relocate rules within a task. This is useful in cases where the relative locations of the rules are important. In Figure 4, a plan may interchange the two rules in task "PutCardsOn-Boards" so to assign card-type-2 before card-type-1. Note that a plan is composed of rules with unique names, so a plan may be modified by another plan, i.e. *meta-plan.*

B. Knowledge Engineer Assistant

The "Knowledge Engineer Assistant" (KEA) is a module within ISCS which provides the development environment for a knowledge engineer. KEA supplies three major features: "structure-based editors" to guide the knowledge engineer in constructing the knowledge base, "development tools" to facilitate the knowledge encoding and maintenance, and an "interpreter" to translate the ISCS knowledge sources into internal representation. The main purpose for KEA is to make available to a knowledge engineer enough tools and built-in mechanisms so as to reduce the total time and effort needed in creating a configuration knowledge base. KEA is built upon the DEDIT facility in the Xerox Lisp machines which is itself a "structure-based editor" for the INTERLISP-D and LOOPS environment. KEA commands are conveniently built into the DEDIT menu system. The KEA operations are tightly integrated within the DEDIT framework.

1. Structure-based Editors

One of the main focuses of ISCS is to provide an appropriate set of knowledge representations or structures to reflect conceptually the different sources of configuration knowledge found in experts. By providing the knowledge engineer with a more natural and convenient means of encoding expert knowledge, the process of knowledge acquisition is made easier. These knowledge structures are expressed using the ISCS configuration language. Each knowledge structure has its own syntax, internal structure and control mechanism. The "structure-based editors" guide the knowledge engineer by providing syntax templates for the various knowledge structures. Syntax and semantics are checked before it is entered into the knowledge base; preventing errors in the knowledge base at the earliest possible onset.

For example, the KEA will immediately create a new variable or class for a knowledge engineer, if a value demon is entered when the variable or class that it affects is not yet defined. A knowledge engineer is not allowed to exit form the KEA editors until all necessary information has been furnished. A future improvement would be for KEA to automatically keep track of all such "loose" pieces and prompts the knowledge engineer to complete the knowledge at appropriate times.

2. Development Tools

The "development tools" provide an integrated set of knowledge access facilities which allows the knowledge engineer to inspect the current status of the knowledge base, to list the library functions appropriate for the current context, as well as to integrate new knowledge structures. For example, just by simply marking a variable of a class and selecting appropriate "development tool" menu item, the system can list all the current constraint demons attached to this slot, or all other classes which also include this variable in their definition.

Each new piece of expert knowledge might have to be integrated with existing knowledge structures already in the knowledge base. This integration is made easy and error-free through the use of access functions and automatic encoding. For example, browsers may be used to display the relationships among pieces of knowledge, e.g. the "part-of" relationship.

There are also knowledge source dependent tools which reduce the amount of encoding required by the knowledge engineer. For example, in the "part-of" hierarchy, when a contain-parts pointer is set from an object to sub-part objects, the inverse pointer is automatically added into the parent object structure.

3. Interpreter

The "interpreter" translates knowledge sources developed using the ISCS configuration language into internal representations. The internal representation consists of INTERLISP-D and LOOPS structures. The main unifying mechanism is the object-oriented programming paradigm in LOOPS. In particular, we use the active value feature of LOOPS to implement the value demons, and the "New" method of a metaclass to implement class demons.

C. The User Interface Generator

The ISCS system provides a user interface generating facility called WSI (Mimo, 1986). The facility assists knowledge engineers in the development of user interfaces. Our design is based on an analysis of the environment that a human expert would work in. Consider the case of the sales or field representative who is configuring a computer system for a customer. The volume of data that one must manage is plentiful. After gathering the information, the representative stores the data on a collection of business data forms. The data entry process itself is intermittently spread out across many interactive sessions. The purchase order is usually reworked many times before it is finally sealed. When one builds a knowledge-based system for such a type of users, one must design the system so as to fit the user's working habit. The WSI facility attempts to simulate the field representatives' working behavior by providing *data sheets* for them to enter data and a *file cabinet* metaphor for storage and organization of data sheets. A knowledge engineer may develop customized data sheets and cabinets with the aid of the WSI facility.

The WSI facility is based on a single integrating concept, the data sheet. A data sheet is a business data entry form in concept and a screen image of an internal object in operation. Both end users and knowledge engineers perceive a WSI-derived user interface as one that provides capability for the organization and manipulation of data sheets. Through this integrating idea consistent user interface behavior is achieved. A *data sheet* may be filed in a *folder* which is stored in a *drawer* that, in turn, is contained in a *cabinet*.

A data sheet in the WSI facility is a scrollable window which displays the attribute names and values, of an object. The data sheet window is mouse-sensitive and both the attribute names and values may be selected by a mouse device. The behavior of the WSI facility may depend on the button pressed, the attribute selected, and the context of the system configurator at the time of the mouse selection. For instance, a user interface may be designed such that when a user "clicks" at the attribute name "Memory Size" in a "Site Information data sheet", menus with different items will appear depending on the model number of the site system. This feature is useful because different models of computers have different memory configurations.

The WSI facility supplies generic mechanisms to support the mapping between internal objects and external screen data sheets, operations on the data sheets, and storage and retrieval of data sheets. All the generic mechanisms are implemented by methods in "mixins". Moreover each mechanism has a full set of default behaviors. A knowledge engineer, by simply including the WSI mixins as supers in the class definitions of the components and systems, may immediately obtain a user interface with standard behavior. Customization may be obtained by adding special properties to instance variables in class definitions. WSI, by examining these properties of an object, determines whether a particular variable should be displayed or not in the corresponding data sheet, whether it is modifiable or read-only, whether menus are attached to the variable, what menu (dynamically or statically created) to show, etc. WSI is non-obtrusive because the properties that it relies on are separated from those used in the actual computational tasks. On the other hand, it is possible for a knowledge engineer to indicate to WSI to examine properties involved in computations so as to minimize redundant and inconsistent information between the user interface and the computation tasks.

The WSI facility is integrated into the ISCS programming environment through the inheritance and specialization techniques of object-oriented programming. A knowledge engineer may gradually enhance and refine the user interface by the addition of more and more properties to each instance variable in a class definition through a period of time. This incremental approach is very important in projects like knowledge-based systems where the initial phase of the project is focused mainly on knowledge acquisition and leaves little time for the user interface. By linking to the WSI facility via the "superclass" specification, a standard user interface is immediately obtained. Starting on day one, a knowledge engineer may use the WSI facility for demonstration and testing. As the knowledge base grows and is validated, a knowledge engineer can then pay attention to interface issues. Through the single concept of data sheets and object-oriented programming, a consistent user interface is achieved across the development cycle of a knowledge-based system. Figure 5 shows the interface of a system configurator that utilizes the WSI facility.

IV. SUMMARY

We have described a tool kit, ISCS, for constructing knowledge-based system configurators. ISCS is designed to assist a knowledge engineer in the encoding and maintenance of configuration knowledge, and to facilitate user interface development. The various knowledge representations and system modules are integrated through an object-oriented foundation.

The design and implementation of ISCS is leveraged upon our past experience in the development of knowledge-based system configurator (Wu, 1985). Many of the features in ISCS have been hand-coded into a configurator that we have developed recently. ISCS is the outcome of a post-mortem analysis of the former configurator. It is still under improvement and will eventually be used as a standard development vehicle by Honeywell knowledge engineers.

As ISCS is deployed, new requirements and functionality will emerge and incorporated into the system. At present, we have identified two areas that we will study after our current project. The knowledge engineer assistant is currently a "passive" module; a knowledge engineer must decide which knowledge representation to use and then invoke the appropriate editor or tool. We intend to add a "user dialog" feature

which will assist and guide a knowledge engineer in the selection of knowledge representation through an interactive sessions. The other area that we will enhance is in the provision of a facility that enables knowledge engineers to design form layouts; at the moment a WSI data sheet has only a very simple layout consisting of two columns, one for name and one for value.

REFERENCES

Bobrow D.G. and Stefik M., "The Loops Manual", Tech. Rep. KB-VLSI-81-13, Knowledge system area, XEROX Palo Alto Research Center, 1981.

Chun H.W., "The ISCS Knowledge Engineer Assistant", Honeywell SCOS/AST Technical Report, AST8603, 1986.

Hayes-Roth F., Waterman D., and Lenat D., "Building Expert Systems", Addison-Wesley, Reading, MA, 1983.

Lafue G. and Smith R., "A Modular Tool Kit For Knowledge Management", IJCAI, 1985.

McDermott J., "R1: A Rule-Based Configurer of Computer Systems", Artificial Intelligence 19, 1982.

McDermott J., "XSEL: A Computer Sales Person's Assistant", Machine Intelligence, 10, 1982.

Mimo A., Chun H.W., and Wu H., "WSI - A Facility for Organizing and Manipulating Data Sheets", Honeywell SCOS/AST Technical Report, AST8601, 1986.

Mimo A., "WSI: A Guide to its Implementation and Use", Honeywell SCOS/AST Technical Report, AST8602, 1986.

Richer M., "Evaluating the Existing Tools for Developing Knowledge-Based Systems", Standford Knowledge Systems Laboratory, Report KSL85-19, Standford University, 1985.

Rolston D., "An Expert System for DPS 90 Configuration", 9th Annual Honeywell International Computer Sciences Conference, 1985.

Sheil B., "The Artificial Intelligence Tool Box", Proceedings of the NYU symposium on Artificial Intelligence and Business, edited by Reitman W., ABLEX publishing Corp., 1983.

Szolovits P. and Clancey W., "Case Study: OCEAN", Tutorial 8, IJCAI, 1985.

Wu H., Virdhagriswaran S., Chun H.W., and Mimo A., "ISC- An Expert System for the Configuration of DPS-6 Software Systems", 9th Annual Honeywell International Computer Sciences Conference, 1985.

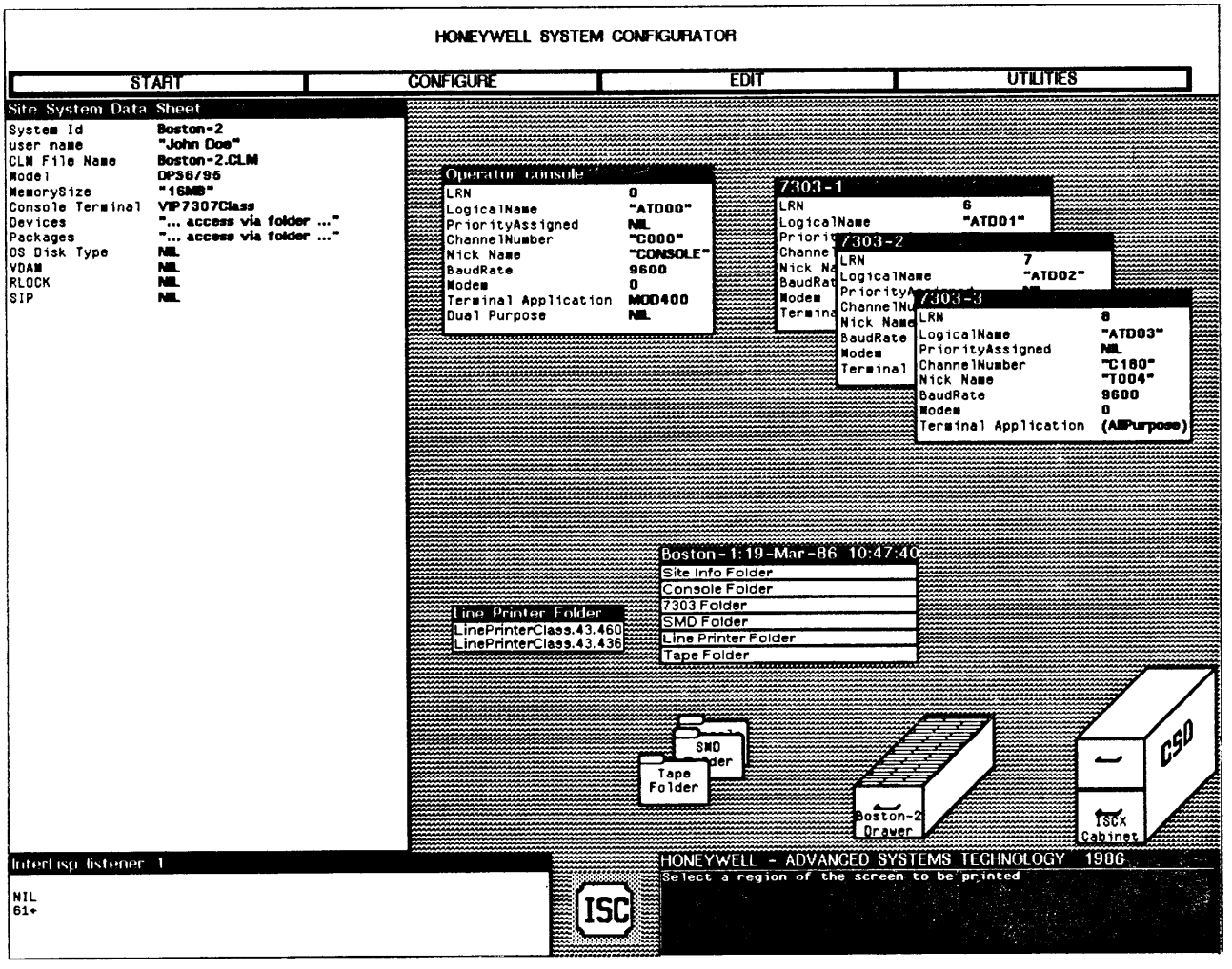

Figure 5. Sample session of a configurator system implemented on ISCS.