Design of Knowledge-Based Systems with a Knowledge-Based Assistant

ERIC SCHOEN, REID G. SMITH, MEMBER, IEEE, AND BRUCE G. BUCHANAN

Abstract—**Intelligent assistants facilitate design and construction of complex software. In this article, we propose a model for an intelligent assistant to aid in building one kind of software—knowledge-based systems—and discuss a preliminary implementation. The assistant participates in knowledge-based system (KBS) construction, including acquisition of an initial model of a problem domain, acquisition of control and task-specific inference knowledge, testing and validation, and long-term maintenance of encoded knowledge. We present a hypothetical scenario in which the assistant and a KBS designer cooperate to create an initial domain model, and discuss five categories of knowledge the assistant requires to offer such help. We then discuss two software technologies upon which the assistant is based—an object-oriented programming language, and a user-interface framework.**

Index Terms—**Knowledge acquisition, knowledge-based systems, object-oriented programming, program editing, programming environments, user interfaces.**

I. INTRODUCTION

INTELLIGENT assistants facilitate design and construction of complex software. In this article, we propose a model for an intelligent assistant to aid in building one kind of software—knowledge-based systems—and discuss a preliminary implementation.

The development of knowledge-based systems (KBS's) is a complex process. Often applied to poorly understood domains—for which standard programmatic solutions do not exist—successful KBS development at present requires collaboration between *domain specialists* and *knowledge engineers*. The process is one of iterative refinement: the specialist describes domain knowledge to the engineer, who in turn encodes that knowledge using tools provided by a KBS "shell," and then elicits further knowledge from the specialist. The approach is far from ideal; difficulties can arise in any of the many steps involved in the transfer of knowledge from specialist to system (e.g., in elaborating the domain knowledge, in encoding the domain knowledge, or in verbal communication).

We view the central problem in KBS design to be knowledge acquisition—moving domain knowledge into a software system by whatever means. This is not simply a problem encountered in the initial stages of design. Rather, it extends over the complete lifetime of a system. Continuing expansion and modification of the system's knowledge base is driven by two factors: changing domain knowledge, and broadened scope of applicability. Designers of existing KBS development environments recognize this problem, and provide a wide variety of tools to reduce its difficulty. Some of these tools—such as syntax checkers—derive from conventional software engineering; others—such as multiple knowledge representation paradigms, explanation facilities, and rule debuggers—have evolved within the KBS community specifically to address the iterative and uncertain nature of knowledge acquisition.

While contemporary KBS development environments contain many tools to help manage the complexity of knowledge acquisition, they offer little or no guidance on how to *use* these tools. As a result, knowledge engineers are often faced with a bewildering set of choices: how to represent domain terminology and relations, how to determine completeness of encoded domain knowledge, how to enter additional domain knowledge without introducing inconsistencies, and so on.

The central theme of this article is a new approach to knowledge acquisition, in which an intelligent assistant joins the domain specialist and knowledge engineer. This third participant aids the engineer in using the tools of the KBS development environment to encode knowledge of the domain. Its support takes a number of forms: recognition of both incomplete knowledge and opportunities for further specialization and elaboration of domain knowledge, consistency and "style" maintenance, automatic "classification" of domain terms [11], debugging aids, explanation concerning the evolution, current status, and dependencies of the knowledge base, and validation of encoded knowledge.

We have come to view the interaction between the knowledge engineer and the engineer's interface as a discourse, albeit one carried out in the "language" of the designer's tool. All too often, system builders are forced to communicate with design aids at a syntactic level, within a vocabulary that is difficult to learn and use. By contrast, communication with a human assistant is at a high level, in which a designer can assume the assistant knows something about the design task and the system (or device) being designed, as well as about the design tools. For effective communication, each participant must be aware of the preconceptions, intentions, and specialized

Manuscript received May 28, 1987; revised January 19, 1988.

E. Schoen and R. G. Smith are with Schlumberger Palo Alto Research, Palo Alto, CA 94304.

B. G. Buchanan is with the Knowledge Systems Laboratory, Stanford University Computer Science Department, Stanford, CA 94305.

IEEE Log Number 8824633.

vocabulary of the other. Further, much is often left unstated during discourse, to be inferred by the participants. In light of this, we propose elevating the status of design tools to that of intelligent assistants, which share this "high-level communication" property with human assistants.

A. The Need for Intelligent Assistance

Developers of early rule-based expert systems, such as MYCIN [14] and EMYCIN [36], were simultaneously attempting to understand conventions for building rule sets and editing knowledge bases (KB's) of rules. Work by Davis [21] and Suwa [56], and later Bennett [3], focused on representing such conventions so that intelligent assistance could be offered at the time of KB design or KB debugging. In particular, the programs resulting from this research encoded knowledge of the *syntax* of rules and some knowledge of the *semantics* of the domain, in order to guide acquisition and maintenance of rule sets.

Knowledge in EMYCIN-based systems was represented primarily in production rules. While constrained in format, and intended to represent independent "packets" of knowledge, these rules also proved surprisingly flexible. However, they frequently contained unforeseen interactions. Thus, it was incumbent upon the knowledge acquisition tools to prevent anomalous rules from becoming part of the rule base. To this end, the EMYCIN interface provided a number of services: syntactic analysis could discover (and often correct) typographical errors; semantic analysis could reveal subsumption or contradiction relationships between new and existing rules; statistics on rule application helped detect overly general or overly specific rule premises.

While EMYCIN was successful in these specific areas, no attempt was made to include in it more general "guidelines of *good practice.*" Yet, such guidelines—concerning efficiency, clarity, and elegance in rule-based system design—were learned and communicated informally among persons using EMYCIN and similar systems.

Unfortunately, the ramifications of some of these guidelines were not totally understood at the time. For example, knowledge engineers were encouraged to add so-called "screening clauses" to rule premises; these clauses defined a narrow context within which the remaining premise clauses made sense. The primary reason for including such clauses in a rule was to prevent the consultation system from asking for specific information prior to establishing the general context for the question. In Fig. 1, for example, the second clause is too specific, unless the first clause establishes that the context is appropriate. The first clause keeps the second clause from being checked when it is not likely to be true.

In retrospect, the use of screening clauses complicated the tasks of maintaining [2] and explaining [17] the rule base. Having no explicit concept of screening clauses, the EMYCIN knowledge acquisition interface could neither detect when one was necessary in a newly entered rule, nor could it identify the screening clauses in the existing

```
IF:  1) There is evidence for pseudomonas, and
     2) There are pseudomonas-type skin lesions
THEN: ...
```

Fig. 1. Sample premise of a MYCIN rule with screening clause.

rule base. EMYCIN was quite successful at syntactic manipulation of rule bases, but much less successful at semantic manipulation.

The EMYCIN experience provided a number of valuable lessons concerning the design of knowledge-based systems; principal of these is the need to represent information explicitly and declaratively. Since then, we have seen the power of this lesson applied in a number of different ways: Aikins' CENTAUR system [2] is a reimplementation of the rule-based PUFF system [31] in a mixed rule-and-object paradigm, replacing context-setting clauses by an explicit taxonomy of disease *prototypes* to which rules are attached; Clancey's NEOMYCIN system [16] builds on Davis' meta-rules [21] to explicitly represent diagnostic strategy, producing a system which can explain and tutor medical diagnosis at a much more detailed level than could MYCIN. When this information is made explicit in the knowledge base, the designer's assistant can use it (e.g., for explanation [57] and for knowledge base debugging [21]). Because much information was left implicit in MYCIN, a substantial amount of work was required when meningitis rules were added to the existing bacteremia rule base [1].

We believe it is now possible to construct a more powerful knowledge base designer's assistant. This assistant will act as an interface between a KBS designer (e.g., a knowledge engineer) and a KBS development environment; it will not only provide the sort of passive support provided by EMYCIN, but will also actively support encoding of domain knowledge.

B. Knowledge-Intensive Development Environments

Key progress in two areas—representational frameworks and computing technology—has enabled us to conceive of tools like the knowledge base designer's assistant. Object-oriented programming permits us to model underlying domain principles with greater clarity and efficiency than was possible in purely rule-based formalisms. Recent advances in computing technology (e.g., powerful personal workstations) permit us the "luxury" of combining interaction and inference while retaining a reactive environment, so that usable interfaces can also perform significant amounts of deduction.

The designer's assistant forms part of a larger vision—which we call a *knowledge-intensive development environment*. This environment has two essential, interrelated components; a *representation and reasoning substrate*, and an *interaction substrate*. Extending the key principle of knowledge-based systems—separation of domain knowledge and problem solving methods—this approach further separates problem-solving methods and interaction.

The representation and reasoning substrate integrates

distinct problem-solving subsystems involving objects, rules, constraints, contexts, and explanation. This substrate has knowledge of the use of the individual subsystems, problem-solving methods, and basic knowledge of the domains in which it is applied. Our current substrate is based upon Strobe, an object-oriented language [50]. It has been augmented by a rule interpreter [48], a constraint manager, and a task configuration manager [32], [52].

The interaction substrate provides a set of tools for constructing interactive user interfaces to knowledge-based systems. It must support at least three different perspectives, corresponding to three different types of user: 1) the developer/maintainer; 2) the domain specialist; and 3) the end user. (Each of these, in turn, may require different perspectives as well.) The substrate must provide a reactive environment for developer/maintainers. It must allow domain specialists to focus on the encoded domain knowledge, hiding the underlying representational mechanisms, and provide direct expression and interaction in terms natural to the domain. Finally, the substrate must permit the construction of transparent and "easy-to-use" interfaces for end users. These three interfaces may appear as separate tools, but in fact, they all share a common, underlying software architecture. Our recent work on Impulse-86 [49], [51] indicates that it is possible to construct an interaction substrate that provides this architecture, and supports the needs of all three user groups.

The representation and reasoning substrate already contains tools well-suited to user interface design. The object-oriented paradigm used in domain knowledge base construction is equally viable for encoding and organizing interface constructs like editors, windows, menus, and views. Knowledge base construction and maintenance is itself a knowledge-based task, and thus, the interaction substrate must be a knowledge-based system. Reasoning mechanisms from the representation and reasoning substrate can be brought to bear on the management of user interaction. For example, constraint systems which support problem solving can be used to help maintain consistency during user interaction. Rules can be used to infer missing or dependent information. The same browsing techniques used by the developer to explore the system code can support graphical explanation—for developers, domain specialists, and end users alike. Analogies to discourse can be drawn in the knowledge-intensive development environment. The interaction substrate must reason about the background, sophistication, and intentions of its user, as well as the available domain knowledge, to offer meaningful assistance. It also must often infer information left implicit by the user.

The complete knowledge-intensive development environment we describe is as yet a research goal; however, many of the individual components of the substrates do exist and are in daily use. Our first step towards integrating the components into the complete development environment is the knowledge base designer's assistant, a component of the interaction substrate. In Section II, we describe our current efforts aimed at constructing this tool. In the following sections, we describe the software technologies upon which it relies: in Section III, we discuss the role of object-oriented programming; in Section IV, we describe interaction substrates, using Impulse-86 as an example. We present samples of the sorts of interfaces it can construct and discuss its utility for knowledge-based system design. We discuss related work in Section V, and summarize our approach to knowledge acquisition in Section VI.

II. Example

In this section, we discuss the role of an *active agent*—the knowledge-based designer's assistant—in the design process. When completed, it will offer the designer a wide spectrum of assistance, ranging from instruction in use of available tools to selection of representation, reasoning, and interaction paradigms. The assistant will appear as an expert—albeit computerized—user of the KBS development environment. In this section, we describe the various services that will be provided by the assistant.

The interface between the designer and the assistant is Impulse-86. The assistant monitors knowledge base editing commands issued by the designer, and interacts by displaying suggestions in a set of windows. The assistant is aware of all changes made to an evolving knowledge base. (As with the original conception of the "Programmer's Assistant" in Interlisp [28], the assistant can be viewed as looking over the designer's shoulder.)

Throughout this section, examples are drawn from a hypothetical session in which the designer and assistant collaborate on the design of the kernel of a semiconductor fabrication knowledge base. (Information regarding VLSI technology is taken from [58].) The dialog between the designer and assistant is in reality graphical in nature, driven by interaction with customized interfaces; for clarity, we present the scenario as a transcript, in stylized natural language, of that interaction. In Section IV, we show some of the Impulse-86 editors which underlie the scenario.

Our convention is as follows: text in *italics* is commentary on the transcript; text in **bold font** is either input from the designer, or represents concepts related to the knowledge base under construction; text in roman font is output from the assistant. The assistant uses **bold font** when referring to concepts in the designer's knowledge base.

A. Assistance in Building a Domain Model

In this example, we consider only the early phase of KBS construction. The major goal of knowledge acquisition at this point is to build a model of the domain, in terms of a vocabulary of domain-specific terms and relations. (This is a *model-based* approach to knowledge acquisition: we construct this model prior to considering specific problem cases and the knowledge needed to solve such cases. In the alternative *case-based* approach, specific problem cases are used to identify the necessary domain terms and relations.) Terms and relations from this model are used in task-specific inference and control

knowledge, as well as in explanations and conclusions produced by the system. For purposes of illustration, we assume this information is represented as objects in an object-oriented language (see Section III).

During this phase, the assistant will guide acquisition of terms and relations so as to produce a model which encodes domain information in as clear, consistent, and complete a manner as possible. By this, we intend the knowledge base to contain little redundant information, so that information pertaining to any group of objects is associated with the maximally general object in the group. Further, the knowledge base should contain no contradictory information. Finally, the knowledge base should contain all terms and relations which are used by inference and control rules, as well as system-generated explanations.

Clearly, this represents an ideal, and probably unrealizable, goal. We expect inconsistencies and incomplete information to become apparent as the domain model is applied to actual problems. A designer cannot guarantee that the domain model is complete before beginning to add inference and control knowledge to the system (since unforeseen terms and relations may be necessitated by adding such knowledge). Instead, we expect the assistant to employ heuristics that judge when enough vocabulary has been acquired to begin the next phase of knowledge acquisition. Acquisition and correction of the domain model will not cease with the end of the early phase of KBS construction; rather, it will likely continue in a fashion interleaved with acquisition of inference knowledge. The assistant will be able to detect when the addition of an inference or control rule necessitates extending or correcting the domain model, and temporarily switch back to the model acquisition "mode."

Our example shows the assistant helping the designer during model acquisition by suggesting terms and relations to be specialized and/or elaborated. (The former involves defining new, more specific, subclasses of objects in the knowledge base; the latter, adding slots to existing objects.) It will do this by analyzing the evolving domain model to produce a set of *goals*, and then analyzing those goals to produce a set of knowledge base *modifications* to be made to achieve those goals. The modifications can then be communicated to the designer in a number of ways: as a description of the changes to be made to the information encoded in the knowledge base (i.e., changes to the meaning of the knowledge), as a description of the changes to be made to structures in the knowledge base (i.e., changes to the implementation of the knowledge), or as a set of editing commands to be performed to change those structures. These alternatives represent explanations of deficiencies in the model, at increasingly lower levels of abstraction; the assistant may choose the explanation which is most appropriate, given knowledge of the designer's experience.

The assistant needs to employ at least five categories of knowledge to be able to offer meaningful advice:

Type 1: Knowledge of "generic tasks," (e.g., knowledge of broad categories of application areas),

Type 2: Knowledge of knowledge base design (e.g., procedures for acquiring and elaborating knowledge, and commonly-used knowledge base structuring techniques [clichés]),

Type 3: Knowledge of the semantics of the language used to encode the domain model (e.g., general knowledge about relations, and knowledge about mapping between meanings and representation of meanings in the language),

Type 4: Knowledge of the editing tools available to the designer (e.g., available editing commands and their effects), and

Type 5: Knowledge of discourse (e.g., knowledge of how to infer the goals, plans, and intentions of the user).

B. Hypothetical Design Session

We present an excerpt from the beginning of the hypothetical design session, in which the designer and assistant begin to encode the domain model. For each inference made by the assistant, we indicate which of the above types of knowledge has been used.

Assistant: What sort of task will this KBS be performing? I currently know about the general characteristics of diagnosis, manufacturing, and design tasks.

Designer: **Manufacturing.**

> *The assistant now examines its generic model of manufacturing, and finds that manufacturing can be viewed as a sequence of steps which consume raw materials to produce a finished part (Type 1 knowledge). The assistant now creates a goal of mapping terms from this particular manufacturing task—semiconductor fabrication—onto terms in the generic model (Type 2 knowledge).*

Assistant: You could now add information to the domain model to describe the sort of **Finished-Part** being manufactured.

> *These suggestions—which the designer is free to ignore—appear in a number of forms. The assistant maintains a prioritized agenda of goals, which it displays in a window. The assistant operates continuously "in the background," and reorders and redisplays its goal agenda in response to changes in the knowledge base. The assistant makes other suggestions by highlighting portions of Impulse-86 editing windows already visible on the screen (e.g., the assistant's last suggestion could involve flashing the* **Finished-Part** *node in the graph displayed in Fig. 2, accompanied by explanatory text, such as above).*

Designer: **How?**

> *The assistant's last suggested change was phrased in terms of a change to information encoded in the domain model. The designer asked how to implement this change, which suggests to the assistant that the designer does not understand how this information is to be encoded (Type 5 knowledge). It explains the change, in slightly less abstract terms—as changes*

to the structure of the knowledge base (Type 3 knowledge).

Assistant: Create a subclass of the concept **Finished-Part.**

Designer: **How?**

The assistant infers that the designer does not understand how to use the available editing tools (Type 5 knowledge). The assistant repeats the last suggestion, in terms of Impulse-86 editing commands (Type 4 knowledge).

Assistant: Select the **Create Object** command in the **KB Commands** menu attached to the KB editor for the **Manufacturing** knowledge base. Respond to the **Object Names:** prompt with the name of the **Finished-Part** produced by this manufacturing application, to the **Object Type:** prompt with **class,** to the **Generalizations:** prompt with **Finished-Part,** and to the **Groups:** prompt with any symbol you choose, or simply with ⟨**return**⟩.

The designer carries out the above steps, and names the specific kind of finished part being produced.

Designer: Creates a subclass of **Finished-Part** *called* **Integrated-Circuit.)**

Assistant: You should now add information which names the **Raw-Material** used to manufacture **Integrated-Circuits.** Create subclasses of **Raw-Material** using the **Create Object** command as you did previously.

(Types 2, 3, and 4 knowledge.)

Designer: (Creates subclasses of **Raw-Material** *called* **Silicon, Dopant,** *and* **Aluminum.)**

At this point, the model contains information which indicates that silicon, dopant, and aluminum are the raw materials used to manufacture integrated circuits. Continuing in the same manner, the assistant prompts the designer to specialize **Step,** *thereby defining the steps which compose the manufacturing process. As this contributes nothing new to our example, we leave out this part of the hypothetical design session.*

In this example, Type 1 knowledge includes a knowledge base which contains a generalized domain model of manufacturing. Figs. 2 and 3 illustrate portions of this model. Fig. 2 portrays a taxonomy of manufacturing terms (e.g., **Raw-Material** and **Finished-Part** are both subclasses of **Material**). The assistant copies this taxonomy for use as the kernel of the semiconductor model. Wherever the assistant prompted the designer with, "You should now add information. . . ," it was suggesting a term in the kernel domain model which could be specialized for semiconductor manufacturing. Fig. 3 indicates that the **Manufacture** object is described by two slots, **Raw-Material** and **Finished-Parts.** Each of these slots is further annotated by a **Role** facet, which encodes how the slot contributes to the meaning of the **Manufacture** object; values stored in the **Role** facet are drawn from a small

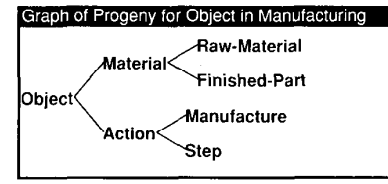


Fig. 2. Taxonomy of terms in the Manufacturing knowledge base.

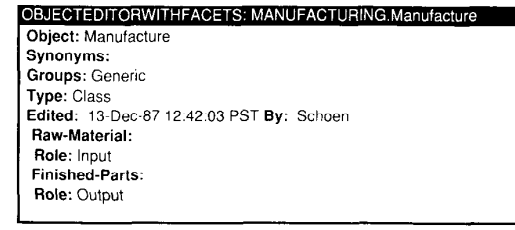


Fig. 3. Slots and facets in the **Manufacture** object.

set of terms defined elsewhere in the manufacturing knowledge base (e.g., **Input** indicates that the slot identifies a quantity which is an input to the manufacturing process). Neither slot contains a value in the **Manufacture** object; however, sometime later in the design session, the assistant will suggest that the designer specialize this object, and provide fillers for the slots.

Figs. 4 and 5 illustrate the state of the model sometime after the end of the session excerpt. The designer has created a subclass of **Manufacture,** called **Process,** which represents the semiconductor manufacturing process. He or she has also created a number of subclasses of **Step,** each of which represents a distinct subtask in the manufacturing process. Fig. 4 displays the current taxonomy of domain terms. Fig. 5 displays the input and output relationships between materials in the actual manufacturing process.

Knowledge of knowledge base design techniques, Strobe semantics, and Impulse-86 editing commands (Types 2, 3, and 4 knowledge) enables the assistant to translate high-level goals into concrete suggestions. For example, in order to explain its suggestion, ". . . add information to the domain model to describe the sort of **Finished-Part** being manufactured," the assistant refers to its knowledge regarding the representation of domain terms. The designer is being asked to provide a term which specializes the class **Finished-Part** for the semiconductor domain. The knowledge that classes represent domain terms, and that specialization relationships between domain terms are represented as subclass relations between corresponding classes, enables the assistant to suggest that the designer should create a subclass of **Finished-Part.**

When asked for further help, the assistant refers to its knowledge of Strobe and Impulse-86 to direct the designer to create a subclass of **Finished-Part.** From knowledge of Strobe semantics, the assistant determines that a subclass of **Finished-Part** will be represented as a class object whose generalization is **Finished-Part.** From knowledge of Impulse-86, the assistant describes the sequence of editing commands which will create the desired

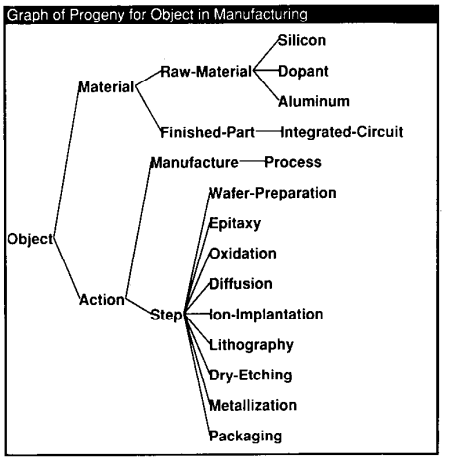


Fig. 4. Taxonomy of terms in the Semiconductor Manufacturing knowledge base.

OBJECTEDITORWITHFACETS: MANUFACTURING.Process
Object: Process
Synonyms:
Groups:
Type: Class
Edited: 13-Dec-87 16:37:49 PST **By:** Schoen
Raw-Material: Silicon, Dopants, and Aluminum
Role: Input
Finished-Parts: Integrated-Circuits
Role: Output

Fig. 5. Slots and facets in the **Process** object.

subclass. Note that advice is being stated in terms drawn from the evolving domain model; we believe this in itself is an important form of assistance. In-context advice is more useful than that drawn from a printed manual, or from a canned example domain. The instructions apply directly to the development task at hand; the user is freed from having to draw (possibly flawed) analogies between example and real domains. Furthermore, since the assistant knows the exact sequence of editing commands needed to implement its suggestion, it could execute the commands itself and relieve the designer of a potentially tedious task. We expect such on-line help will ease the process by which designers become experts at using the tools provided by their environments.

Another use of Type 2 knowledge we envision involves common practice in knowledge base design. We expect the assistant to know about *clichés*—common patterns of relationships between objects.[1] Some of these clichés (such as consistent refinement of slot values across a subclass taxonomy) represent "good practice" and will be encouraged by the assistant; others (such as misuse of the *is-a* relation) represent "bad practice" and will be discouraged. When the assistant recognizes a developing cliché, it will either offer to help complete the cliché, or suggest an alternative knowledge base structure which is more acceptable.

[1]We use the term *cliché* in much the same manner as in KBEmacs [60]. However, our clichés are structural—abstracting relations between concepts—while those of the Programmer's Apprentice are procedural—abstracting programs.

As an example of this sort of help, we present another, short excerpt from our hypothetical design session. This excerpt occurs sometime later in the session; in the interim, the designer has added a number of new terms and filled in a number of slots. See Figs. 6–8.

Assistant: I notice you've created a number of subclasses of **Wafer-Preparation.** Let's look at **Produce-Electronic-Grade-Silicon.** The **Output** of this **Step** is **Electronic-Grade-Silicon.** Since **Produce-Electronic-Grade-Silicon** is a subclass of **Wafer-Preparation,** and **Electronic-Grade-Silicon-Wafer** is the output of **Wafer-Preparation,** I would have expected **Electronic-Grade-Silicon** to be a subclass of **Electronic-Grade-Silicon-Wafer.**

Perhaps you've incorrectly defined **Electronic-Grade-Silicon,** or perhaps you intended a relation other than *subclass* to hold between **Wafer-Preparation** and **Produce-Electronic-Grade-Silicon.** Since for each of the subclasses of **Wafer-Preparation,** the value of **Output** slots appears as the value of the **Input** slot of another subclass, I suggest each of the subclasses is actually a *substep* of **Wafer-Preparation,** ordered by the relation between the **Input** and **Output** slots. Is this what you meant?

> *A brilliant deduction by the assistant. How realistic is it? The generic manufacturing task includes information about likely relations that occur; substep relations are very common in manufacturing, so the assistant is justified in suggesting it as possible correction. Furthermore, the substep relation is a kind of* part-whole *relation; the assistant knows that in such relations, there will be a part or parts which share structure with the whole. In this case,* **Wafer-Preparation** *shares its input with* **Produce-Electronic-Grade-Silicon,** *and its output with* **Saw-Into-Wafers.** *Thus, the assistant concludes that* substep *is the appropriate relation.*
>
> *In a narrowly circumscribed domain, like semiconductor manufacturing, it is reasonable to expect that a small number of relation types will be applicable; however, in the general case, we expect this type of inferencing to become computationally intractable. It is therefore important for the assistant to have a facility for interacting with the designer to select (or define) the appropriate relation type. For example, if the assistant is not able to suggest a single most likely replacement, or if the designer disagrees with the assistant's suggestion, the assistant displays a menu of possibilities (potentially all manufacturing relation types, or all known relation types), and allows the designer to choose.*[2] *We assume here that the designer has agreed with the assistant; the assistant then rearranges the knowledge base and displays the result (Figs. 9 and 10).*

In the preceding excerpt, the assistant recognized a "misuse-of-is-a" cliché, by recognizing that the value

[2]The assistant could be expected to learn from its mistakes, using strategies similar to those employed in the LEAP system [38].

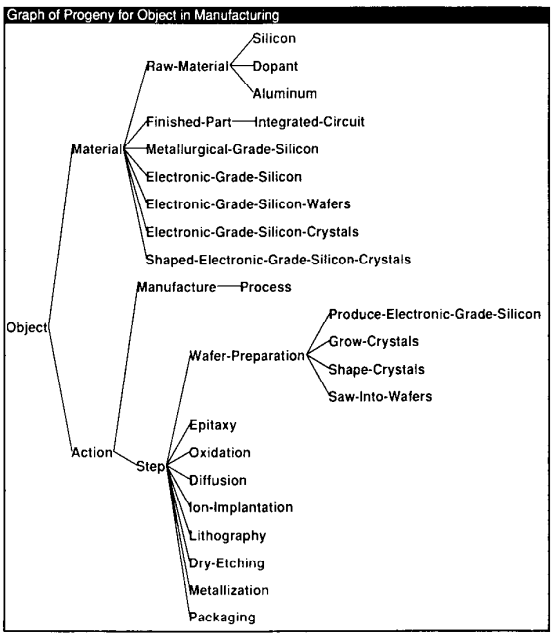


Fig. 6. Later taxonomy of terms in the Semiconductor knowledge base.

OBJECTEDITORWITHFACETS: MANUFACTURING.Wafer-Preparation
Object: Wafer-Preparation
Synonyms:
Groups:
Type: Class
Edited: 13-Dec-87 17:02:39 PST **By:** Schoen
Input: Metallurgical-Grade-Silicon
Role: Input
Output: Electronic-Grade-Silicon-Wafers
Role: Output

Fig. 7. The **Wafer-Preparation** object.

OBJECTEDITORWITHFACETS: MANUFACTURING Produce-Electronic-Grade-Silicon
Object: Produce-Electronic-Grade-Silicon
Synonyms:
Groups:
Type: Class
Edited: 15-Dec-87 15:12:09 PST **By:** Schoen
Input: Metallurgical-Grade-Silicon
Role: Input
Output: Electronic-Grade-Silicon
Role: Output

Fig. 8. The **Produce-Electronic-Grade-Silicon** object.

(**Electronic-Grade-Silicon**) of a slot (**Output**) in a subclass (**Produce-Electronic-Grade-Silicon**) was not a subclass of the value (**Electronic-Grade-Silicon-Wafer**) of the same slot in the superclass (**Wafer-Preparation**). It further suggested a correction by finding a better cliché (step-substep), which was both a likely candidate given the domain, and a fitting correction, given the information (input–output relationships) already encoded in the knowledge base.

C. Further Opportunities for Assistance

The sorts of assistance we have described so far pertain largely to the acquisition of the domain model—the vocabulary of terms and relations—of a domain. However,

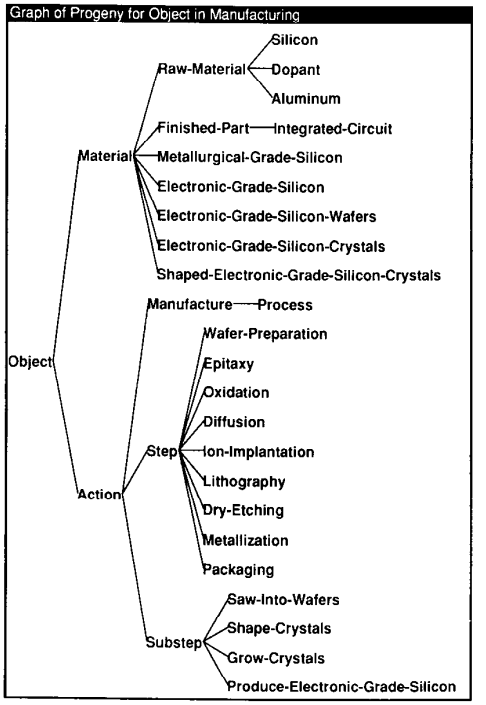


Fig. 9. Corrected taxonomy of terms in the Semiconductor knowledge base.

OBJECTEDITORWITHFACETS: MANUFACTURING.Wafer-Preparation
Object: Wafer-Preparation
Synonyms:
Groups:
Type: Class

Input: Metallurgical-Grade-Silicon
Role: Input
Output: Electronic-Grade-Silicon-Wafers
Role: Output
Subtasks: Produce-Electronic-Grade-Silicon, Grow-Crystals, Shape-Crystals, and Saw-Into-Wafers

Fig. 10. Corrected **Wafer-Preparation** object.

the role of an intelligent assistant is not limited to this aspect of knowledge acquisition. The domain model is not a complete knowledge base; it lacks necessary control and task-specific inference knowledge. Clearly, opportunities exist for an assistant to guide the knowledge engineer and domain specialist through the acquisition of these remaining classes of knowledge.[3] We intend to explore integrated acquisition of domain, control, and task-specific inference knowledge in the assistant. For these aspects of knowledge acquisition, the assistant will require knowledge about AI problem solving methods and strategies.

The assistant can also play a role in the testing and validation of a KBS; once the knowledge base contains enough information, the assistant is able to record the re-

[3]The process is most likely not a sequential ordering of phases. We believe that domain, control, and task-specific inference knowledge will be acquired in an interleaved fashion; attempts to correct deficiencies in one class of knowledge may highlight deficiencies in one or both of the other classes.

sults of individual tests and (perhaps) to generate test cases (by examining the encoded knowledge). As additional knowledge is acquired, the system must continue to successfully treat prior test cases. The assistant can play a useful role in checking that newly added knowledge does not conflict with existing knowledge—that the evolving knowledge base remains self-consistent. As in TEIRESIAS [21] and LEAP [38], failed test cases can identify missing and/or incorrect knowledge. Furthermore, the assistant can record dependencies between parts of the knowledge base; it is continuously involved in the evolution of the knowledge base and thus has a historical record of the development. This information can be used in two ways: the assistant can explain how a change to one part of the knowledge base will affect other parts, and can also undo changes, to allow alternate attempts at knowledge base evolution. As we have stated, maintenance of the knowledge base in this manner is likely to continue throughout the lifetime of a knowledge-based system.

In the next two sections, we discuss the two enabling technologies upon which our knowledge-intensive development environment rests: Strobe, an object-oriented programming language, and Impulse-86, a framework for constructing user interfaces. Whereas the assistant is an *active* tool, these are *passive* substrates which underlie it.

III. Object-Oriented Programming

Object-oriented programming is the foundation of our knowledge-intensive development environment. It contributes towards a programming methodology, which integrates diverse components of the development environment, and a representation language, which models both domain-level and development environment knowledge. The use of object-oriented programming for building expert systems and other knowledge-based applications, has been previously described [23]. In this article, we describe the role of object-oriented programming in building development environments for knowledge-based systems. In this section, we provide a brief overview of the object-oriented paradigm.

KBS's typically evolve in an incremental refinement process characterized by an exploratory programming style [27], [14], [46]. The representation and reasoning substrate is central to the KBS developer/maintainer in managing the complexity of the evolving system. A clear conceptual model of the domain of application is essential for managing this complexity.

Object-oriented programming has been found by simulation specialists, cognitive psychologists, and artificial intelligence scientists to be very useful in modeling—of physical systems, human systems, and artificial systems. In constructing software systems to model complex phenomena, it is advantageous to organize computation around programming constructs whose internal structure and interrelationships explicitly reflect those of constructs in the physical world in which the resultant systems are to operate. Clarity is especially important because of the inherent complexity of the physical world—this complexity must be managed in a software system if it is to be understandable, explainable, extensible, maintainable, and reusable.

Furthermore, the object-oriented style encourages modular code and encapsulation with well-defined interfaces. Inheritance of properties simplifies code-sharing. This leads to space-efficient code and to ease of maintenance and specialization, as has been found in a variety of systems [25], [61], [6]. In addition, many traditional KBS representational entities (e.g., rules, constraints) may be encoded as objects and invoked in a uniform manner—via messages.

A clear model of the evolving system is as important to the domain specialist as it is to the KBS developer. Objects appear to be a natural and understandable knowledge organizing mechanism to humans not normally involved in computation [18]. The concept of an object as a prototype for encapsulating information—both data and procedures—is well-understood and used by humans, as are the concepts of classes and instances, taxonomies—and taxonomic inheritance of properties (along with a number of other forms of inheritance).

A. Background

In an object-oriented software system, the central construct is the *object*—a data structure similar to a record. In a medical consultation system, such as NEOMYCIN [16], we find objects that correspond to important medical concepts, like patient, drug, organism, disease, and so on. Objects are linked together in a variety of relationships in an object-oriented system. In NEOMYCIN, it is explicitly recorded that viral meningitis is a kind of meningitis, which is in turn a kind of infection. This hierarchical *a-kind-of* relation—also called the *taxonomic* relation—is carefully supported by object-oriented programming languages. Unlike the simple support provided for records by ordinary programming languages, these object-oriented languages support *inheritance of properties*. If it is recorded that meningitis is an infection situated in the cerebrospinal fluid (CSF), then it is not necessary to separately record that viral meningitis is also situated in the CSF. This information is available—*transparently to the modeler*—via inheritance.

In object-oriented programming, the structure of the knowledge used in the software system is defined by the structure of the physical world—and not by the specific task at hand. Contrast this with the normal procedural structuring used in almost all computer programs. Because the computational structure is determined by the structure of the domain being modeled, object-oriented systems are more readily understood by both domain specialists and programmers. The objects themselves provide a natural atomic structutre—a level of detail appropriate to the world being modeled. The relationships that link the objects—especially the hierarchical taxonomic relationship—lend increased structure. The collection of objects that model a particular domain, often called a *knowledge base*, captures the static knowledge of that domain.

The knowledge base provides a kind of computational skeleton for a simulator, a reasoning system, or indeed any procedure that operates on it.

Over the years a number of object-oriented languages have been developed. One of the earliest was Simula, an extension to Algol designed for simulation [20], [5]. Simula defined much of the vocabulary of object-oriented programming. During the 1970's, Smalltalk was developed as a vehicle for making computers easier to use [24]. It has become the best-known object-oriented language, in large measure due to the revolutionary impact it had on integrated programming environments and user interfaces. In the artificial intelligence (AI) community, a number of object-oriented languages were developed to attack problems in vision and natural language understanding (e.g., KRL [7], FRL [44]). (The AI languages were often called *frame-based* at the time [37].) Whereas Simula and Smalltalk were independent, free-standing languages, most of the AI languages were embedded in an underlying programming language, Lisp. Commercial systems have started to become widely available in recent years (e.g., KEE [30], Loops [6]). Object-oriented programming concepts have started to appear in traditional programming languages; C++ [55] and Objective-C [19] are both object-oriented extensions to the standard C language.

B. Features of Object-Oriented Programming Languages

We devote the remainder of this section to the salient features of object-oriented programming languages. We concentrate on the features useful for modeling the static structure of a domain. We do not address the issue of messages and computation with an object-oriented skeleton. See [39], [54] for discussion of these issues.

The knowledge base designer's assistant itself is written in Strobe [50], a Lisp-based object-oriented programming language. Strobe is representative of many such languages, which provide a powerful set of tools that augment Lisp; moreover, such languages can themselves be embedded in higher-level systems that offer greater representational structure and modeling complexity. Strobe has been implemented in both Interlisp-D and Common Lisp; a subset of Strobe has been implemented in C.

Basic Concepts: To begin, we define the basic data structures. From largest to smallest level of granularity, the structures commonly provided by Lisp-based object-oriented languages are the following:

Knowledge Base: A taxonomically organized collection of objects of a domain. A knowledge base typically represents a model of some physical domain or system. Each knowledge base is a separate namespace of objects. An application may be organized around several knowledge bases.

Object: A record-like structure which encapsulates a coherent set of related data *and* procedures. An object belongs to a particular knowledge base. It typically represents a single concept in the domain or system modeled by the knowledge base. An object is identified by a unique name within a knowledge base—objects of the same name may exist in different knowledge bases.

Each object has an associated *type*. Objects that represent particular, unique individual entities in the model are called *individual* or *instance* objects. Objects that represent a *set* of entities (either entities that actually exist in the model or entities that could potentially be constructed), are called *class* objects.

Slot: A component of an object. A slot can represent a property or attribute of an object, a procedure that the object can execute (often called a *method*) to exhibit some behavior of the object, or a relationship between the object in which the slot resides and one or more other objects.

Facet: A component of a slot—a place to encode annotations regarding the meaning of the slot with respect to its object. Many object-oriented languages define a **datatype** facet, which indicates what type of datum is (or can be) stored as the value of the slot. Applications can further define their own facets (as in the **role** facets illustrated in Fig. 3).

Taxonomic Structure in Knowledge Bases: Objects form the atomic structure of knowledge bases. Additional structure is imposed by relationships that exist among the objects. Object-oriented languages support a variety of relationships—some of which are hierarchical. The foremost relation supported by all object-oriented languages is the taxonomic relation, sometimes also called the *a-kind-of*, *is-a*, or *generalization/specification* relation. Fig. 11 shows a taxonomic hierarchy associated with the assistant's knowledge of set-theoretic properties of relations (a form of Type 3 knowledge discussed in Section II).

From the figure, we see that **TransitiveSymmetricReflexiveRelation** is a kind of **Relation.** This is an expression of the universally quantified conditional:

$$\forall X(\textbf{TransitiveSymmetricReflexiveRelation}(X) \rightarrow \textbf{Relation}(X));$$

that is, if X *is-a* **TransitiveSymmetricReflexiveRelation,** then X *is-a* **Relation.** We can also say that **TransitiveSymmetricReflexiveRelation** is *a-kind-of* **Relation.** In addition, the taxonomic relation is transitive. So, because:

$$\forall X(\textbf{EqualityRelation}(X) \rightarrow \textbf{TransitiveSymmetricReflexiveRelation}(X)),$$

we can conclude that:

$$\forall X(\textbf{EqualityRelation}(X) \rightarrow \textbf{Relation}(X)).$$

Explicit representation of hierarchical taxonomic relationships is a powerful conceptual modeling tool. It has been used extensively in the physical sciences as a way of bringing order—through classification—to systems observed in the real world. For computational models, at the

Fig. 11. Taxonomic hierarchy of relations.

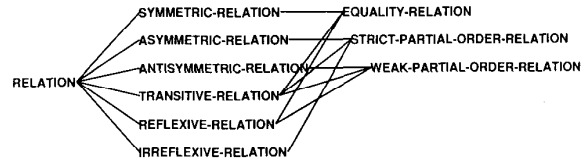


Fig. 12. Taxonomic lattice of relations.

very least, it imposes a perspicuous structure on the domain knowledge being represented. As such, it helps to manage the complexity inherent in models of substantive real world systems.

Most object-oriented languages support taxonomic *lattices*; that is, objects may have multiple generalizations. In Fig. 12, for example, the transitive, symmetric, and reflexive characteristics of relations are represented separately. This permits useful partitioning of information, advantageous because it fosters clarity and because it reduces the amount of information that otherwise would need to be duplicated in several classes. In the figure, transitivity information is stored in **TransitiveRelation,** rather than being duplicated in all transitive relations (see [54] for another example). Taxonomic lattices are often called *tangled* hierarchies. It is possible to get by without support for multiple generalizations. Many simple object-oriented languages do not support them because they increase the complexity of the implementation of the object-oriented language itself. However, if multiple generalizations are not supported, the application designer is often forced to write application-specific code that amounts to the same thing. Furthermore, the designer must spend time worrying about these situations rather than concentrating on the already difficult enough task of constructing an accurate and useful model.

Property Inheritance: If explicit representation of taxonomic relationships served as nothing more than a conceptual modeling tool, it would be very powerful. But, it has an additional purpose. It serves as the basis for *property inheritance*. For example, in the relation system, if we record that descendants of **ReflexiveRelation** are relations in which reflexivity holds, then we need not separately record that **WeakPartialOrderRelation** and **EqualityRelation** are reflexive. This fact is *transparently* accessible via the inheritance mechanisms supported by the object-oriented language. Inheritance of properties follows naturally from the the universally quantified conditions noted earlier: if X is an **EqualityRelation**—*a-kind-of* **ReflexiveRelation**—then the properties of **ReflexiveRelation** are also properties of *all* **EqualityRelations** and hence are properties of X. Were we to ask the question: Are *instances of* **EqualityRelation** *reflexive?*, an object-oriented system would respond affirmatively.

Inheritance is not only a powerful conceptual modeling tool; it is also a powerful programming tool. It encourages modularity of design through *elision* and *refinement*. We achieve *elision* of description of objects through inheritance, which promotes a compact representation—only those properties of an object which are specific to the object itself need be explicitly stated. We *refine* properties within specializations, thus simplifying incremental knowledge base extension; properties generic to an object's class can be replaced by similar, but more specific properties, appropriate to a new subclass, independent of extensions and refinements in different branches of the taxonomic hierarchy.

Classes and Instances: Figs. 11 and 12 show only *class* objects. A class object defines the generic properties of all of its specializations. Since **Relation** is a class object, it defines a set of properties for all subclasses and instances of **Relation.** A class object may also define *default* values for the properties it defines, although this is not necessary. Subclasses and instances of the class are assumed to have those values. For example, assume **Relation** contains a **detect** slot which holds rules valid for detection of equality relations. The rules in the **detect** slot are by default assumed to be valid for detection of *all* subclasses and instances of **Relation.** If a default value is not specified for a property, nothing is assumed about its value in subclasses and instances.

In many object-oriented languages, default values may be overwritten, refined, or *cancelled* in specializations. For example, the **detect** slot of **Relation** may be specified as unknown; however, it may be refined in the class **EqualityRelation to (rule184 rule116** . . .). Defaults are very useful in knowledge-based systems—they can be used to make weak inferences in the absence of more specific information—and later overridden as more specific information becomes available.

Most object-oriented languages treat the *is-a* relation, defaults, and cancellation informally. For example, the *is-a* relation often conflates the notions of a prototypical member of a set and a universally quantified conditional. Further, permitting cancellation of inherited defaults can weaken the descriptive power of an object-oriented representation language (simply because an object matches the description of some class X, it may not necessarily match the description of Y—a superclass of X—due to possible cancellation of properties in X); hence, if an object-oriented language allows cancellation, we can only use the taxonomic hierarchy for weak inferencing. See [9] and [8] for a detailed discussion of these issues.

IV. Impulse-86: Support for User Interaction

Impulse-86 [49] is a Strobe-based tool which supports user interface design. Responsible for the low-level "mechanics" of user interface support, it is a key component of the interaction substrate in the knowledge-intensive development environment. In this section, we present a set of requirements for user interfaces and user interface

toolkits, and discuss how Impulse-86 fulfills these requirements. By way of illustration, we present four examples of Impulse-86 interfaces, each intended to support the activities of a specific class of user. We then discuss the framework in which these interfaces are constructed, using the examples to illustrate the main points.

Impulse-86 consists of an extensible kernel, providing support for the basic requirements of window-oriented user interfaces, and a set of application-specific extensions built around the kernel. It is the third in a series of systems beginning with Impulse [45]; while the first two were designed for the explicit purpose of editing Strobe knowledge bases, Impulse-86 is a general-purpose user interaction framework. It enables developers and end users to construct customized, domain-specific interfaces for their systems, without being experts in interactive graphics (knowledge base editing is a specialized extension). Our approach is to provide an extensive set of "building blocks," and a uniform framework in which to assemble them.

Although Impulse-86 has evolved from a knowledge base editor to a user interaction framework, we continue to view its kernel task as a form of editing. We take the point of view that *editing* entails many kinds of interactions between a user and a software system. The user *views* the state of the system, and *controls* or *changes* the state of the system. An editor is an entity that mediates these viewing and controlling interactions—presenting the user with a view of the system and effecting the desired control. This definition encompasses traditional text editing, browsing, program development, debugging, and end-user interaction.

Knowledge-based systems typically consist of many complex structured objects, connected by several relationships, and are intended to model richly structured domains. As a result, both the KBS developer and end user are typically interested in interacting with (i.e., understanding, changing, extending, and using) three major kinds of entities: objects and their internal structure, relationships among objects, and complete systems of objects, relationships, and code—including their dynamic behavior.

Based on the considerations thus far presented, we have been led to the following design goals for interfaces constructed with Impulse-86:

- Effective integration and use of high resolution bitmapped displays, pointing devices, and keyboards.
- Flexible support for varied methods of user interaction for both viewing and controlling systems (e.g., graphics, animation, menus, pointing and *smart* typein, with facilities like partial name recognition, spelling correction, and line editing). The editor should support interaction with domain-specific information in a form natural to an end user familiar with the domain.
- Specific aids for *browsing*. A user should be able to quickly examine an evolving knowledge base, alternate hypotheses constructed by a KBS, user-defined interobject relationships, and so on.
- Unlimited interaction contexts for viewing and changing different parts of a system simultaneously.
- Programmatic access to interaction mechanisms. A user should be able to call upon building blocks in the editor in the midst of normal programs.
- Organizational support for customization and extension.[4]

There are several other useful goals that Impulse-86 itself does not address, including user modeling, assistance to a user in managing trouble, recovery from error, checkpointing, security, and so on. Extensions along these dimensions (such as the designer's assistant we discussed in Section II) are more appropriately constructed on top of the kernel environment, rather than inside it.

Fig. 13 is a snapshot of a typical Impulse-86 screen, in which six interaction contexts are active. Many Impulse-86 interfaces follow a convention in which information is displayed in a central window, to which one or more menus are attached. For example, the small window in the upper right corner of the screen, labeled "Impulse-86," displays the names of each loaded Strobe knowledge base; the menu attached to this window contains commands related to loading, storing, and editing Strobe knowledge bases. The editor in the bottom right corner of the screen is a Strobe knowledge base editor; the central window contains information about the knowledge base, the upper menu contains knowledge-base editing commands, and the lower menu contains an alphabetized list of each of the objects in the knowledge base. Selecting an object from this latter menu provides a "focus" for some commands (e.g., the **Edit Object** command) in the former menu. In the following, we discuss the remaining editors in Fig. 13 in greater detail.

A. Examples of Impulse-86 Editors

In this section we show a number of brief examples of specialized Impulse-86 editors. Our aim here is to demonstrate the diversity of interface styles that can be supported by the framework. Our examples depict editors applied to knowledge bases which form part of the designer's assistant, and which encode information about both relation types and management of task agendas.

The Object Editor: The object editor is used for editing a single Strobe object and its internal structure. Fig. 14 depicts a user's view of the standard object editor; the object being edited encodes knowledge about relations.

The object is presented to the user in a window with attached menus. The first five lines show the values of properties that are common to all objects; the remaining lines display the object's slots. The name of a property or slot is shown in boldface; if a slot has synonyms, they are enclosed in set braces; the slot's value (if any) is shown in lightface, following a colon; (↑) indicates that the slot

[4]Stallman defines extensibility relative to EMACS: ". . . the user should be able to add new editing commands to change old ones to fit his needs, while he is editing" [53]. We go beyond *command* extensibility, aiming for extensibility with respect to what is viewed, how it is viewed, and how it may be changed.

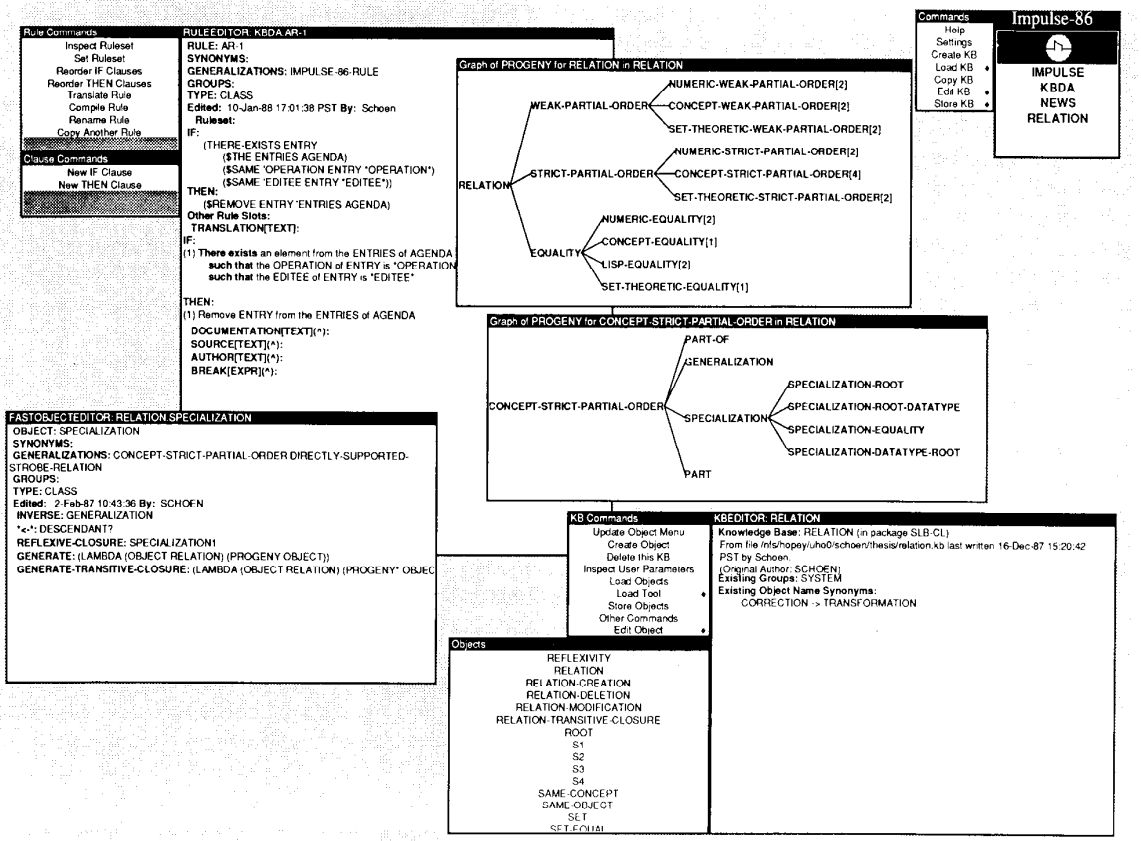


Fig. 13. Example screen during a typical Impulse-86 editing session.

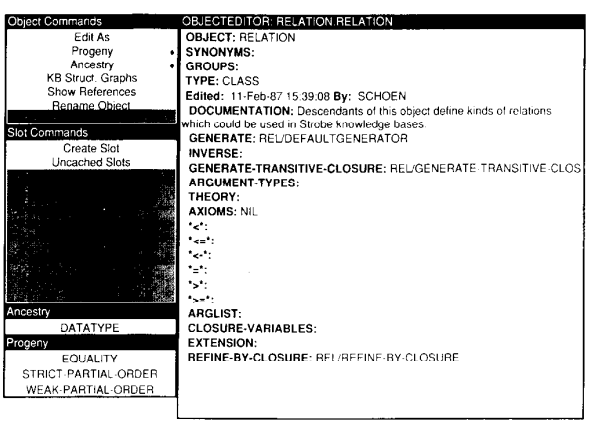


Fig. 14. Object editor.

is inherited from a more general object. We have found these conventions useful, but they are only defaults. A user can override them to customize the way this information, or other information, is presented.

The top two command menus enable a user to (for example) rename the object, change any of the slots, create new slots, delete or rename slots—standard editing operations. A small mark to the right of a command indicates the existence of subcommands that extend the functionality of that command. We have adopted the discipline that a *left* mouse button selection invokes the command shown, while a *middle* mouse button selection pops up a menu of subcommands (or a menu of arguments that further specify a command).

The two menus show the objects most closely related (taxonomically) to the object being edited; they can be used to invoke a new editing context in which one of those relatives is the center of attention.

Items in the display may be selected with the mouse as arguments to a command. A selection stack is maintained, with individual selections (*foci*) indicated by different styles of highlighting. As with other defaults, the highlight styles may be changed by the user.

A variant of the object editor is visible in the lower left corner of Fig. 13. This variant—the fast object editor—is designed for more experienced users of Impulse-86. It displays precisely the same information as does the standard object editor, but presents no command menus. Instead, these menus are available as "pop-up" menus, and appear when a mouse click occurs over the object name or any of the slot name captions. Because the fast object

Fig. 15. Rule editor.

editor spends no time creating permanent menus, it can be displayed faster than can the standard object editor; furthermore, since it obscures less screen space, more of these editors can be made visible without occluding one another.

The object editor is used mainly by the knowledge base maintainer—typically a computer scientist—although it can also be useful to a domain specialist. It offers a view and interaction style for the domain knowledge encoded by the system—more as a computational entity than a mathematical entity (i.e., the implementation of the mathematical concept of a relation as a software object encoded in an object-oriented language). In later examples we will see other ways of interacting the domain knowledge—each appropriate to a specific task.

The Rule Editor: The rule editor is used for editing a single Strobe rule. Fig. 15 depicts a user's view of the standard rule editor; the rule being edited encodes a single piece of knowledge used by the designer's assistant in managing task agendas. This editor is commonly used by developers and by domain specialists. It is not intended for end-users of a knowledge-based system.

A Strobe rule is encoded as an object. As a result, the rule editor is rather similar to the object editor. In this case, however, commands specifically useful for dealing with rules replace or extend the basic object-related commands (e.g., the **Translate Rule** command, used to generate the English translation of the rule shown at the bottom of the figure). Furthermore, the slots of the rule are grouped together and captioned in a different manner. For example, the slots that encode left-hand side conjuncts are grouped under the **IF:** caption. Similarly, the slots that encode right-hand side actions are grouped under the **THEN:** caption.

In the figure, we see another standard attention-focusing mechanism supported by Impulse-86. Some commands in the top two menus on the left-hand side of the main window are grayed over. If a user selects a datum for which these commands are relevant, then Impulse-86 will make them visible to allow their selection. (Another example of this behavior is also visible in the top two menus of Fig. 14.)

The Graph Editor: Graphs are a natural representation for viewing relationships between and among several objects in a system. Impulse-86 provides a simplified, object-oriented interface to a variety of graphing primitives, so users can edit interobject relationships graphically.

A graph editor displays some or all of the objects tied together by a relationship (or set of relationships). For example, the graph editor in Fig. 16 shows a taxonomic hierarchy associated of relations. Each node represents one object. More specialized objects appear to the right of their generalizations.

Graph editors enable a user to see and rearrange the overall structure of the system as implied by the graph relationship. Through pop-up menus, the user can add or delete links, and can easily invoke an interaction context that shows more detail for any object in the graph. Graph editors are like other editors with regard to defaults and foci. A user may further customize the graph editor with respect to fonts, layout (e.g., vertical versus horizontal), link types (e.g., dashed versus solid), node display (e.g.,

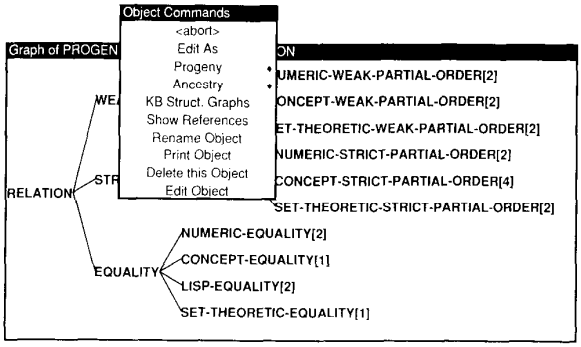


Fig. 16. Graph editor: taxonomic hierarchy of relations.

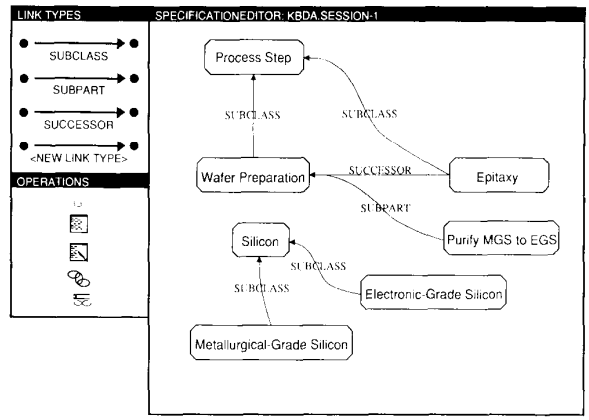


Fig. 17. The specification editor.

bitmaps versus names), amount of information presented, and so on. These customizations typically require no more than simple declarative changes to default values.

Impulse-86 has a number of built-in graphing relationships, such as specialization and generalization. In addition, it supports construction of customized graphs, based on user-specified functions, that, given a datum (e.g., an object), generate its successors one *graphic link* away (e.g., according to a *part-of* relationship). Impulse-86 takes care of the mechanics of computing and displaying the transitive closure of the graph generator function, making the nodes of the graph sensitive to selection with the mouse, attaching the menus, and so on.

The views offered by graph editors are useful to both developer/maintainers and to domain specialists. In addition, we will later see how some graphs can be useful to end users of systems. For example, the graph of Fig. 16 is particularly useful to a domain specialist in understanding how much of the standard set-theoretic relationship vocabulary has been encoded. Such graphs are pervasive in the sciences. They help show skeletal models of physical domains.

The Specification Editor: The forms of assistance we discussed in Section II require the assistant to understand an evolving domain model as encoded by the designer in an object-oriented framework. The assistant must "abstract" the domain model from the implementation-level data structures in the knowledge base. It depends on the designer having chosen an appropriate *grain-size* at which to represent the model.

An alternate approach we are considering involves providing an Impulse-86 editing interface customized to the task of entering terms and relations in the domain. This interface would encourage the designer to "sketch" information at an appropriate level of granularity, and would prevent him or her from entering the implementation-level knowledge base directly. Instead, the assistant would produce the knowledge base by analyzing information recorded by the custom interface.

Fig. 17 illustrates our current implementation of the specification editing interface. The interface allows a designer to enter the primary terms and relations of a domain, without having to declare how the terms and relations are actually encoded in a knowledge base. The editor consists of a "workspace" (the main window), and two iconic menus. The *operations* menu selects the editor mode; from top to bottom, the available modes are **select concept or relation, create concept, delete concept, create relation,** and **delete relation.** The *link type* menu in the middle selects the type of relation to be created when the editor is in **create relation** mode. There are two predefined relation types—**subclass** (i.e., the *is-a* relation) and **subpart** (for encoding part-whole decompositions). The designer can define additional relation types by selecting the ⟨**New Link Type**⟩ entry in the *link type* menu; for example the **successor** relation was defined in this manner. The workspace depicts a collection of terms and relations; in Fig. 17, for example, the **Process Step** term has two subclasses: **Wafer Preparation** and **Epitaxy. Wafer Preparation** has a successor—**Epitaxy**—and a subpart—**Purify MGS to EGS.**

When the designer is satisfied with the model depicted in the workspace, the assistant constructs a Strobe implementation of the model. Once again using its knowledge of KBS design techniques and of Strobe semantics (Types 2 and 3 knowledge), the assistant chooses to implement domain terms as Strobe classes, the relations between terms as slot values. Knowledge about relations (Type 3 knowledge, such as that illustrated in Section III) can help characterize new relations defined by the designer. For example, if the **successor** relation is seen to be transitive, irreflexive, and asymmetric, the assistant can assume the relation is a strict partial order; this information may be useful later when verifying the consistency of information encoded in the knowledge base, as well as in presenting a graphical depiction of such information.

We believe this type of interface will help the designer concentrate on specifying domain principles, rather than computational entities. The designer is not required to declare how terms and relations are implemented (i.e., which are the objects and which are the slots); the actual encoding of the domain knowledge can be performed later, at least partially by the assistant. By contrast, an interface composed of Impulse-86 object and slot editors

would immediately force the designer to decide how to represent terms and relations, a decision which might have to be reconsidered later as the knowledge base evolves.

B. The Impulse-86 Substrate

The Impulse-86 substrate contains five major building blocks: *Editor*, *EditorWindow*, *PropertyDisplay*, *Menu*, and *Operations*. Each building block, or part, is a Strobe object in the **Impulse** knowledge base. Each embodies special knowledge that enables it to fulfill a particular role in an interface (described below).

Editor: The editor is the central object. It mediates interactions between the user and the *editee*—the domain focus—typically a part of a knowledge base (analogous to the Smalltalk *model* [24]). Together with its components, it *defines* a user interface—the way information is viewed and controlled.

Editor *instances* are built from an editor *class* by template instantiation. By instantiating a separate editor for each editee, Impulse-86 enables an unlimited number of independent interaction contexts to exist simultaneously.

Editors have components drawn from any of the five major classes (or from additional classes defined by a user). Editors are explicitly permitted to have other editors as components (and so on, in a recursive fashion). Support for recursive decomposition is important. It enables the construction of an editor whose composite structure parallels the composite structure of its editee. Knowledge about the way composites are structured in the application domain may therefore be embedded in the structure of its interfaces.

Fig. 18 shows the editor structure for the class of object editors. Each object editor instance is generated from this template. Fig. 19 shows the instance corresponding to the object editor shown in Fig. 14. In both graphs, an arc indicates that the object on the right is a *component* of the object on the left. Instantiated components are shown in lightface; non-instantiated components are in boldface. (Figs. 18 and 19 were generated from Impulse-86 specialized graph editors.)

In this example, **ObjectEditor** and **SlotEditor** are both editors. The **ObjectEditor** mediates interactions with the object-specific properties (e.g., the *object's* name and synonyms), while the **SlotEditor** mediates interactions with the slot-specific properties (e.g., the *slot's* name and synonyms).

Subeditors can be dynamically added to, and removed from, an instantiated editor. Impulse-86 allows a user to specify (via a declaration or a method) a mapping between a class of objects in a domain knowledge base and an editor class to be used for them (e.g., the **RuleEditor** for **Rule** objects).

EditorWindow: The editor window manages the screen context of a collection of editors. It is responsible for performing the usual window operations (e.g., scrolling, repainting, reshaping). It also maintains a correspondence between editees and the window regions in which the editees are displayed. This enables both data selection by mouse pointing and efficient update when parts of an overall domain structure are changed. The window also records *foci*—items selected by the user as arguments to a command.

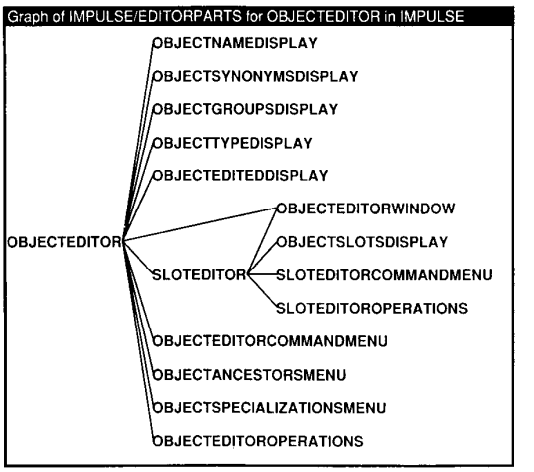


Fig. 18. Components structure of the object editor.

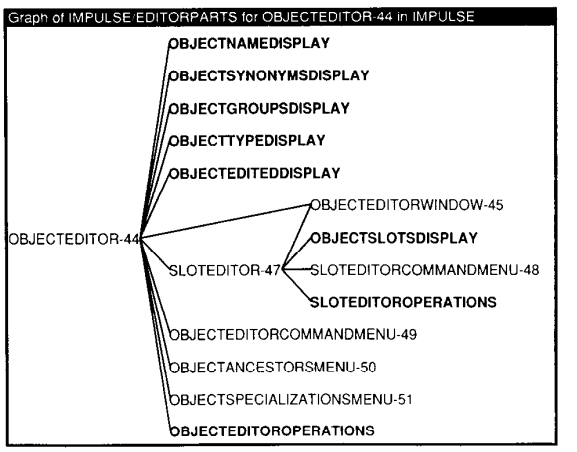


Fig. 19. An object editor instance.

Each editor may have at most one window among its components. When an editor has subeditors, some or all may share the same window, or utilize separate windows. Fig. 19 shows that **ObjectEditor** and **SlotEditor** instances both share the same instance of **ObjectEditorWindow.**

PropertyDisplay: The property display presents a view of an editee in a window. Impulse-86 has a number of different kinds of display, each implementing a distinctive visual style—and a user can define new types of display. A property may be *active* (i.e., have regions sensitive to mouse selection).

A property display is also responsible for setting up a correspondence between its editee and the window regions in which the editee is displayed. Impulse-86 has been designed so that property displays need not be instantiated (although it is straightfoward to do so if the

property display requires more context than just the correspondence between its editee and displayed window region). Editor windows, on the other hand, must be instantiated in any case—and are therefore used to maintain the correspondences set up by property display classes.

There are six displays in the object editor example. The first five are components of **ObjectEditor;** each display one of the five properties associated with the object itself. The sixth (**ObjectSlotsDisplay**) is a component of **SlotEditor;** it iterates over all of the slots.[5] Each of these displays prints a single datum in the **ObjectEditorWindow.**

The graph editor has a single specialized property display. It displays the entire graph in the editor window. (Graphical displays can also be combined with other types of property displays in editor windows.)

There is a large number of **PropertyDisplay** templates. For example, many property displays have a similar *caption* and *value* format. Each of these property displays has a caption, typically sensitive to selection with the mouse (*active*), followed by a region that contains a *value* (e.g., the property display for the **DOCUMENTATION** slot in the object editor of Fig. 14). The **Caption&ValuePropertyDisplay** class is the common generalization. Its specializations provide alternative ways to display captions and values, such as displaying captions as bitmaps.

Menu: Impulse-86 provides a large number of built-in menu styles, ranging from static menus to pop-up and pushbutton menus.

Menus are used to display choices to be made (*choice* menus) or operations to be invoked (*command* menus). When a selection is made in a command menu, Impulse-86 informs the editor with which the menu is associated. The responsibility for executing the selected operation lies with the editor (see below). Menus may be unique to a particular editor or shared among a collection of editors.

Menus are implemented as specialized windows. They typically have a restricted format for items (e.g., rows and columns), and a single method for handling mouse button events. In principle, they could be implemented as windows with simple property displays. However, performance considerations on current workstations and window systems often dictate a specialized implementation. Hence, analogous to the **Window** object, the **Menu** object is an interface to one or more menu packages associated with the underlying window system of the target machine for Impulse-86.

Impulse-86 provides support for different menu types. These include: **StaticMenu** for menus that remain on the screen [e.g., the menus associated with the **ObjectEditor** (Fig. 14)]; **DynamicMenu** for menus that pop up in response to some selections, and **MultipleChoiceMenu** for menus that allow a set of choices to be made before finishing an interaction and causing some action to occur. In addition, there is support for menus that contain sets of operations to be performed (**CommandMenu**), menus that contain a set of arguments required to *complete* specification of operations (**CommandArgumentMenu**), a variety of different menu styles (e.g., **PushButtonMenu** for control panel buttons, **StatzDisplayMenu** for large, scrolling *status* menus).

The topmost menu in Fig. 14 is a command menu associated with the **ObjectEditor.** The second menu is a command menu associated with the **SlotEditor.** The next two menus (**Ancestry** and **Progeny**) are command menus that show the immediate relatives of the editee object (and new interaction contexts can be invoked by selecting objects in those menus).

The rule editor specialization of the object editor (Fig. 15) has two new command menus for **Rule Commands** and **Clause Commands.** It uses a specialized form of the standard slot editor command menu (**SE Commands**), and inherits the **Ancestry** menu.

The separation between **ObjectEditor** and **SlotEditor** is used to advantage for indicating which commands are appropriate to a user-selected item in the editor window. For example, when an object-specific property has been selected, Impulse-86 grays over the menu of slot-related commands. This helps to focus the attention of the user on the relevant commands.

Operations: Methods that perform the operations defined for an editor are grouped in operations objects. These methods are invoked by a message from the editor.

ObjectEditorOperations knows how to execute the operations listed in the **ObjectEditorCommandMenu; SlotEditorOperations** knows how to execute the operations listed in the **SlotEditorCommandMenu.** For example, when the **SlotEditor** receives a message that the user selected **Rename Slot,** it relays that message to **SlotEditorOperations,** which has a method that actually renames the slot.

Separating operations from menus permits the same editing operations to be invoked in a variety of ways, including menu selection, typein, function invocation, or message from a remote processor.

In addition to the five major building blocks, there are a few minor building blocks. For example, Impulse-86 provides high-level support for keyboard interaction through **TTYInteractionWindows.**

C. Discussion

Impulse-86 has been used to construct a wide variety of user interfaces. These range from simple "caption and value" editors (e.g., the object and rule editors described above), to graph editors (each with a specialized strategy for handling information overload). They include two-dimensional iconic graphics editors (used in program design) and data graphics editors (used in scientific data interpretation). See [49] for a complete discussion of Impulse-86.

Specialized Impulse-86 interfaces typically require short development times. We attribute this to the particular set of behaviors encapsulated in Impulse-86 building blocks.

[5]Each property display is typically responsible for one editee. However, a property display (indeed, any component) can be iterated over a list of editees.

The designer is insulated from the details of manipulating bitmaps, windows, mouse interactions, typein streams, and menus—I/O details which traditionally require painstaking attention. Attention can therefore be concentrated on what the customized interface should look like, what information it should show the user, and what commands the user should be able to request.

Several features of the Impulse-86 substrate contribute to this ability. The building blocks are quite general, covering a wide range of interaction styles. Implementation as an object-oriented system makes it easy to specialize and modify. A uniform, relatively fine-grain protocol is used throughout, together with a number of known methods and defaults. These characteristics ease the problem of understanding what is in the system and what mixins to use as a starting point for customized interfaces. Furthermore, no distinction is made between default methods and default values; this simplifies declarative specialization of the kernel. Finally, Impulse-86 contains a number of archetypal interfaces; modifying one of these is an effective strategy for creating a new interface.

Other groups are currently working on systems that support construction of interactive interfaces. The GUIDON-WATCH system [43] demonstrates the utility of a user interface tuned to the operation of a particular class of consultation systems. The authors note that knowledge-base editors orginally intended for use by knowledge base developer/maintainer are typically inappropriate interfaces for the end user. Impulse-86 offers a substrate that bridges the gap between the tools required by a developer/maintainer, by a domain specialist, and by an end user. Its extensibility further enables it to support the construction of specialized interfaces for each type of user.

The SIG system [35] is much closer to Impulse-86. Built on top of Smalltalk-80 [24], it too offers an extensible kernel that supports generation of interactive displays. In the terminology of [35], SIG emphasizes the *view* aspect of interaction; it addresses the *control* aspect in a less structured manner. In contrast, we have found it useful in Impulse-86 to provide a considerable amount of structure to support the control aspect of interaction as well as the view aspect. We have also found it useful to provide a relatively fine-grained structure to support user extension.

Another related effort is EZWin [33], an object-oriented editing system which provides three object classes for constructing editors. Although the two systems have similarities, there are several important differences. "EZWin systems are basically editors for graphical objects." [33, p. 186] Systems constructed in Impulse-86 are interfaces for knowledge-based systems; the interaction objects and routines are completely separate from the knowledge-base being edited. This separation allows the interface and application to be modified independently and supports the reuse of interfaces with different knowledge-based systems.

Perhaps closest to Impulse-86 is PSBase [15], a prototype system for modeling and constructing user interfaces. Like Impulse-86, PSBase separates the interface from the application. It also considers the view aspect of interaction (called presentation), and the control aspect (called recognition). Ciccarelli describes the implementation of a number of well-known text-oriented and icon-oriented interfaces in PSBase (e.g., the Xerox Star interface [47]). Impulse-86 and PSBase share much in spirit, but differ in specifics. Both systems explicitly consider composite interfaces and sharing of structure. Because Impulse-86 has been carried through to production use in a wide variety of applications, it provides a larger number of interaction styles.

Much of the recent work on graphical programming systems is relevant to Impulse-86. These systems provide iconic interfaces for creating, editing and animating programs. Program animation is triggered by application-specific events which update the display. While the interaction with and appearance of these interfaces are similar to interfaces built with Impulse-86, the underlying structure is quite different. First, most graphical programming interfaces [12], [13], [34], [40]–[42] are tightly coupled with their application: interaction and animation are managed by low-level calls from the editor and the interpreter. Impulse-86 separates the interface and application, thereby supporting reuse and modification. Second, only some of these systems [34], [40], [42] provide support for relationships such as composition and dependency. These relationships, and their explicit representation in objects, allow important structural features of the application (such as connectivity) to be maintained. The differences between Impulse-86 and these systems stem from the attempted scope of the systems. All of these systems are closely tied to the domain of programming—extension and reuse of these interfaces is limited to this domain. Impulse-86 aspires to provide interface support for a variety of applications, not just programming.

V. Related Work

The philosophy embodied in this work can be traced back to the MYCIN project, described fully in [14]. The concept of an intelligent assistant derives directly from TEIRESIAS [21] system. Our conception of task models follows from TEIRESIAS's *rule models*. Its technique of "acquisition in context"—eliciting information by asking questions relevant to specific situations, rather than asking general questions—is embodied in the assistant's use of task-specific schemata to drive acquisition of domain knowledge. TEIRESIAS was designed to assist with the debugging nearly complete knowledge bases; it worked by guiding a domain specialist through a faulty reasoning trace, searching for missing and incorrect knowledge. By contrast, the designer's assistant addresses knowledge acquisition techniques for all phases of KBS development. The EMYCIN system [36] defined the concept of a KBS shell; our vision of a knowledge intensive development environment is a descendant of this approach.

Acquisition of task-specific inference and control knowledge is addressed in a number of systems. MOLE

[22] acquires heuristic classification knowledge in a two-phase process. In the first phase, an initial knowledge base is constructed; the domain specialist is asked to supply common hypotheses and evidence, and then draw associations between them. In the second phase, MOLE and the specialist interact to refine the knowledge base. MOLE performs static analysis of the knowledge base to discover ambiguities and inadequacies in the knowledge base; it can recommend corrections for some knowledge base deficiencies discovered by this process. The system also performs dynamic analysis; by examining its failures on known test cases, it is able to suggest knowledge base alterations to remedy its performance. MOLE is a self-contained environment; it contains an inference component as well, allowing it to both test its knowledge, and run actual diagnostic sessions. The approach to knowledge acquisition taken by the designer's assistant differs from that in MOLE. The assistant begins by acquiring the conceptual structure of a domain, without regard for the type of problem solving to be performed. It is our hypothesis that the conceptual structure can serve as the basis for a number of systems. In addition, the assistant is not a self-contained environment; it is a component of an extensible problem solving framework; it must be able to guide acquisition of knowledge for any number of representation languages and inference procedures.

In ROGET [3], Bennett describes a prototype knowledge-based system for acquiring the *conceptual structure*—descriptions of input evidence, inference steps, and output advice—of diagnostic expert systems. ROGET interacts with a domain specialist to acquire the structure of the new system, comparing the task requirements to abstract categories of requirements derived from previously constructed diagnostic systems. The system makes recommendations concerning the scope of the system to be constructed and the specific knowledge engineering techniques to be employed. The principal similarity to the work described in this article is the use of abstract task models to focus knowledge acquisition; however, ROGET is targeted only towards the initial phase of KBS construction. The result of a ROGET consultation is an initial EMYCIN knowledge base; additional terminology, domain-specific inference rules, and control structure must all be acquired within EMYCIN.

The Knowledge-Based Software Assistant (KBSA) is a proposed architecture to aid "... the development, evolution, and maintenance of large software projects" [26]. Software development and maintenance under the KBSA paradigm is fundamentally different from current practice; changes are made to the software specification, rather than to the software itself. The implementation of the software specification is rederived with each change. This is similar to the assistant's providing a concept and relation level interface for knowledge capture, deriving the computational representation of the knowledge base itself. In addition, the KBSA captures design decisions regarding the software project, and can act as an intelligent software assistant to developers, maintainers, project managers, and end-users. This parallels our goals for the KBS designer's assistant quite closely. The KBSA and the designer's assistant differ, however, in a number of important aspects. Whereas the KBSA emphasizes capturing the validating design decisions, the designer's assistant emphasizes helping the designer formulate the conceptual structure of a KBS. The KBSA creates and maintains an *executable specification* of a software system, which it assumes is simpler to debug and validate than the actual implementation. The designer's assistant, however, works with the implementation itself; in KBS development, the concept of a specification is informal, and often evolves as the KBS evolves.

KL-ONE [11] is a knowledge representation framework based on a semantic network model. It and its successors, including Krypton [10] NIKL [29], and KL-TWO [59], have made important contributions towards a formal theory of semantic network languages. Of particular relevance to the KBS designer's assistant is the notion of a *classifier*, a procedure for determining the correct place for a description in a KL-ONE network. Stated simply, the classification algorithm places a new concept in the network such that it is "above" (is an *ancestor of*) all concepts it *subsumes*, and is "below" (is a *descendant of*) all concepts that subsume it. One concept subsumes a second concept if all instances of second would also be recognized as instances of the first. KL-ONE, KL-TWO, NIKL, and Krypton knowledge bases are constructed by classification. The designer's assistant defines neither subsumption nor classification; however, by helping designers produce knowledge bases which encode information in the most general manner possible, it does offer a similar service. The assistant's library of modeling clichés is a means by which it can detect poorly structured knowledge bases; these clichés also allow the assistant to recommend a greater variety of improvements to knowledge base structure. For example, the KL-ONE classifier can only move concepts in a taxonomy; the assistant's "classifier" can in addition recommend changes in the structure of concepts, so as to make explicit generalizations not properly encoded in the knowledge base.

We expect classification to play a larger role as we experiment with larger knowledge bases. As the size of a knowledge base increases, it becomes increasingly difficult to find the appropriate concepts to be specialized and elaborated. (This is analogous to the well-known problems of *finding* and *understanding* in discussions of the software reuse [4].)

VI. Conclusion

Starting from the widespread belief that knowledge acquisition is the central problem in knowledge-based system design, we have argued that a broad spectrum of tools—ranging from powerful representation languages to sophisticated editors and debuggers—is required to manage this complex process. While contemporary KBS de-

sign environments provide many such tools, little or no support in the *use* of these tools is currently available for domain specialists and designers.

Our central thesis is that it is possible to construct an automated assistant which actively participates in the knowledge acquisition process. This assistant guides designers and domain specialists in the proper selection and use of development environment tools, and helps formulate, extend, visualize, and verify the conceptual structure of domain knowledge bases. The assistant is a knowledge-based system whose domain is the design of knowledge-based systems. Its domain knowledge includes models of various classes of knowledge-based systems (such as diagnostic, design, and simulation systems), as well as models of proper usage of both its representation language and development environment tools. The assistant is not limited to helping construct the conceptual structure of a knowledge-based system; it can also offer support throughout the lifetime of the system, in areas such as testing, validation, and knowledge base maintenance. We have partially implemented such an assistant.

Collective experience with tools for KBS development has shown us that a good development environment is critical in helping knowledge engineers produce workable, extensible, and maintainable systems. As with any complex system, poor design choices made during the beginning phase of knowledge base construction may propagate far into an evolving system, often necessitating substantial redesign after the system has grown quite large. The knowledge base designer's assistant can assist in early detection and correction of design errors, thus simplifying later debugging and extension of the knowledge base.

The designer's assistant is part of a larger vision: a knowledge-intensive development environment, which supports development and use of knowledge-based systems from the separate perspectives of developers, domain specialists, and end-users. The environment is composed of two interrelated components: a representation and reasoning substrate, which integrates various categories of problem solving tools, and an interaction substrate, which consists of tools for constructing interactive user interfaces to knowledge-based systems. This organization reflects our view that reasoning and interaction are both knowledge-based tasks; the environment provides the "traditional" representation and reasoning support found in all KBS shells, as well as support for powerful user interfaces—such as the designer's assistant—which themselves can be knowledge-based systems.

Our emerging knowledge-intensive development environment is founded upon two key technologies: Strobe, an object-oriented programming language, and Impulse-86, a user interface framework. Strobe is useful in two ways: as a programming paradigm, it is the "glue" which joins together distinct software subsystems in a uniform manner; as a simple representation language kernel, it supports the construction of computational models which mirror the organization of the physical world in which the software systems are to operate. Impulse-86 is a key component of the interaction substrate; it implements the low-level mechanics of user-interfaces—support for windows, menus, typein, and pointing devices—as well as providing editors and browsers for domain knowledge bases. It further serves as an extensible kernel from which application-specific user interfaces can be constructed.

Strobe and Impulse-86 already contribute much to control the complexity of KBS design. These components, while passive, are the prerequisites for the more autonomous designer's assistant.

Acknowledgment

P. Patel-Schneider, H. Brown, G. Lafue, and M. Tenenbaum provided significant comments on the content and organization of this article; our thanks to them and to the reviewers. We also wish to thank our colleagues at Schlumberger-Doll Research and Schlumberger Palo Alto Research for their suggestions and contributions to the Strobe and Impulse systems.

References

[1] J. S. Aikins, "Prototypes and production rules: A knowledge representation for computer consultations," Ph.D. dissertation, Stanford Univ., Tech. Rep. STAN-CS-80-814, 1980.

[2] ——, "Prototypical knowledge for expert systems," *Artificial Intell.*, vol. 20, pp. 163–210, 1983.

[3] J. S. Bennett, "ROGET: A knowledge-based consultant for acquiring the conceptual structure of a diagnostic expert system," *J. Automated Reasoning*, vol. 1, pp. 49–74, 1985.

[4] T. Biggerstaff and C. Richter, "Reusability framework, assessment, and directions," *IEEE Software*, vol. 4, no. 2, pp. 41–49, 1987.

[5] G. M. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA BEGIN*. New York: Petrocelli/Charter, 1973.

[6] D. G. Bobrow and M. J. Stefik, *The LOOPS Manual*, Xerox Palo Alto Research Center, Tech. Rep., Dec. 1983.

[7] D. G. Bobrow and T. Winograd, "An overview of KRL, a knowledge representation language," *Cognitive Sci.*, vol. 1, no. 1, pp. 3–46, Jan. 1977.

[8] R. J. Brachman, "'I lied about the trees' or, defaults and definitions in knowledge representation," *AI Mag.*, vol. 6, no. 3, pp. 80–93, Fall 1985.

[9] ——, "What IS-A is and isn't: An analysis of taxonomic links in semantic networks," *Computer*, vol. 16, no. 10, pp. 30–36, Oct. 1983.

[10] R. J. Brachman, V. P. Gilbert, and H. Levesque, "An essential hybrid reasoning system: Knowledge and symbol level accounts of KRYPTON," in *Proc. Ninth Int. Joint Conf. Artificial Intelligence*, Aug. 1985, pp. 532–539.

[11] R. J. Brachman and J. G. Schmolze, "An overview of the KL-ONE knowledge representation system," Fairchild Lab. Artificial Intell. Res., Tech. Rep. 30, 1984.

[12] G. P. Brown, R. T. Carling, C. F. Herot, D. A. Kramlich, and P. Souza, "Program visualization: Graphical support for software development," *Computer*, vol. 18, no. 8, pp. 27–35, Aug. 1985.

[13] M. H. Brown and R. Sedgewick, "A system for algorithm animation," *Comput. Graphics*, vol. 18, no. 3, pp. 177–186, July 1984.

[14] B. G. Buchanan and E. H. Shortliffe, Eds., *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Reading, MA: Addison-Wesley, 1984.

[15] E. C. Ciccarelli, IV, "Presentation Based User Interfaces," Ph.D. thesis, MIT, Tech. Rep. AI-TR-794, Aug. 1984.

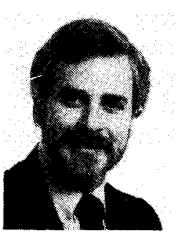
[16] W. J. Clancey, "The advantages of abstract control knowledge in expert system design," in *Proc. Third Nat. Conf. Artificial Intelligence*, Aug. 1983, pp. 74–98.

[17] W. J. Clancey, "The epistemology of a rule-based expert system: A framework for explanation," *Artificial Intell.*, vol. 20, pp. 215–251, 1983.

[18] B. Cohen and G. L. Murphy, "Models of concepts," *Cognitive Sci.*, vol. 8, pp. 27-58, 1984.
[19] B. J. Cox, *Object Oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley, 1986.
[20] O. Dahl and K. Nygaard, "SIMULA—An ALGOL-based simulation language," *Commun. ACM*, vol. 9, no. 9, pp. 671-677, Sept. 1966.
[21] R. Davis and D. B. Lenat, *Knowledge-Based Systems in Artificial Intelligence*. New York: McGraw-Hill, 1982.
[22] L. Eshelman and J. McDermott, "MOLE: A knowledge acquisition tool that uses its head," in *Proc. Fifth Nat. Conf. Artificial Intelligence*, 1986, pp. 950-950.
[23] R. Fikes and T. Kehler, "The role of frame-based representation in reasoning," *Commun. ACM*, vol. 28, no. 9, pp. 904-920, Sept. 1985.
[24] A. Goldberg, *SMALLTALK-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley, 1984.
[25] A. Goldberg and D. Robson, *SMALLTALK-80: The Language and its Implementation*. Reading, MA: Addison-Wesley, 1983.
[26] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich, "Report on a knowledge-based software assistant," in C. Rich and R. C. Waters, editors, *Artificial Intelligence and Software Engineering*, C. Rich. and R. C. Waters, Eds. Los Altos, CA: Morgan Kaufmann, 1986, ch. 23, pp. 377-428.
[27] F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, Eds., *Building Expert Systems*. Reading, MA: Addison-Wesley, 1983.
[28] *Interlisp-D Reference Manual*, Xerox Artificial Intell. Syst., Pasadena, CA, Oct. 1985.
[29] T. S. Kaczmarek, R. Bates, and G. Robins, "Recent developments in NIKL," in *Proc. Fifth Nat. Conf. Artificial Intelligence*, 1986, pp. 978.
[30] *KEE Software Development Systems User's Manual*, 3rd ed., IntelliCorp, 1986.
[31] J. C. Kunz, R. J. Fallat, D. McClung, J. J. Osborn, B. A. Votteri, H. P. Nii, J. S. Aikins, L. M. Fagan, and E. A. Feigenbaum, "A physiological rule-based system for interpreting pulmonary function test results," in *Proc. Computers in Critical Care and Pulmonary Medicine*, 1979, pp. 375-379.
[32] G. M. E. Lafue and R. G. Smith, "A modular tool kit for knowledge management," in *Proc. Ninth Int. Joint Conf. Artificial Intelligence*, Aug. 1985, pp. 46-52.
[33] H. Lieberman, "There's more to menu systems than meets the screen," *Comput. Graphics*, vol. 19, no. 3, pp. 181-189, July 1985.
[34] R. L. London and R. A. Duisberg, "Animating programs using Smalltalk," *Computer*, vol. 18, no. 8, pp. 61-71, Aug. 1985.
[35] D. Maier, P. Nordquist, and M. Grossman, "Displaying database objects," in *Proc. First Int. Conf. Expert Database Systems*, Apr. 1986, pp. 15-30.
[36] W. V. Melle, A. C. Scott, J. S. Bennett, and M. Peairs, "The EMYCIN manual," Dep. Comput. Sci., Stanford Univ., Tech. Rep. STAN-CS-81-885 (HPP-81-16), Oct. 1981.
[37] M. Minsky, "A framework for representing knowledge," in *The Psychology Of Computer Vision*, P. H. Winston, Ed. New York: McGraw-Hill, 1975.
[38] T. M. Mitchell, S. Mahadevan, and L. Steinberg, "LEAP: A learning apprentice for VLSI design," in *Proc. Ninth Int. Conf. Artificial Intelligence*, 1985, pp. 573-580.
[39] D. A. Moon, "Object-oriented programming with Flavors," in *Proc. First ACM Conf. Object Oriented Systems, Languages, and Application*, Sept. 1986, pp. 1-8.
[40] M. Moriconi and D. F. Hare, "Visualizing program designs through PegaSys," *Computer*, vol. 18, no. 8, pp. 72-85, Aug. 1985.
[41] S. P. Reiss, "Graphical program development with PECAN program development system," *Sigplan Notices*, vol. 19, no. 5, pp. 30-41, May 1984.
[42] ——, "An object-oriented framework for graphical programming," *ACM SIGPLAN Notices*, vol. 21, no. 10, pp. 49-57, Oct. 1986.
[43] M. H. Richer and W. J. Clancey, "GUIDON-WATCH: A graphic interface for viewing a knowledge-based system," *IEEE Comput. Graphics Applications*, vol. 5, no. 11, pp. 51-64, 1985.
[44] R. B. Roberts and I. P. Goldstein, "The FRL manual," MIT, AI Memo 409, Sept. 1977.
[45] E. Schoen and R. G. Smith, "Impulse: A display-oriented editor for Strobe," in *Proc. Nat. Conf. Artificial Intelligence*, Aug. 1983, pp. 356-358.
[46] B. A. Sheil, "Power tools for programmers," *Datamation*, pp. 131-144, Feb. 1983.
[47] D. C. Smith, E. Harslem, C. Irby, and R. Kimball, "The Star user interface: An overview," in *Proc. Nat. Comput. Conf.*, AFIPS Press, 1982, pp. 515-528.
[48] R. G. Smith, "Programming with rules in Strobe," Schlumberger-Doll Res., Tech. Rep. SYS-84-12, Dec. 1984.
[49] R. G. Smith, P. S. Barth, and R. L. Young, "A substrate for object-oriented interface design," in *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Eds. Cambridge, MA: MIT Press, 1987, pp. 253-315.
[50] R. G. Smith and P. J. Carando, "Structured object programming in Strobe," Schlumberger-Doll Res., Res. Note, Dec. 1986.
[51] R. G. Smith, R. Dinitz, and P. Barth, "Impulse-86: A substrate for object-oriented interface design," in *Proc. First ACM Conf. Object Oriented Systems, Languages, and Applications*, Sept. 1986, pp. 167-176.
[52] R. G. Smith, G. M. E. Lafue, E. Schoen, and S. C. Vestal, "Declarative task description as a user interface structuring mechanism," *Computer*, vol. 17, no. 9, pp. 29-38, Sept. 1984.
[53] R. M. Stallman, "EMACS: The extensible, customizable, self-documenting display editor," in *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. New York: McGraw-Hill, 1984, pp. 300-325.
[54] M. J. Stefik and D. G. Bobrow, "Object-oriented programming: Themes and variations," *AI Mag.*, vol. 6, no. 4, pp. 40-62, 1986.
[55] B. Stroustrup, *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1986.
[56] M. Suwa, A. C. Scott, and E. H. Shortliffe, "Completeness and consistency in a rule-based system," in *Rule-Based Expert Systems*, B. G. Buchanan and E. H. Shortliffe, Eds. Reading, MA: Addison-Wesley, 1984, ch. 8, pp. 159-170.
[57] W. R. Swartout, "XPLAIN: A system for creating and explaining expert consulting programs," *Artificial Intell.*, vol. 21, pp. 285-325, 1983.
[58] S. M. Sze, Ed., *VLSI Technology*. New York: McGraw-Hill, 1983.
[59] M. Vilain, "The restricted language architecture of a hybrid representation system," in *Proc. Ninth Int. Joint Conf. Artificial Intelligence*, Aug. 1985.
[60] R. C. Waters, "KBEmacs: A step towards the programmer's apprentice," MIT AI Lab, Tech. Rep. 753, 1985.
[61] D. Weinreb *et al.*, *Lisp Machine Manual*, Symbolics, Inc., 1984.

Eric Schoen received the B.S. and M.S. degrees in electrical engineering from Stanford University, Stanford, CA, in 1981 and 1982, respectively.

He is an Associate Member of the Technical Staff at Schlumberger Palo Alto Research, Palo Alto, CA. From 1982 to 1984, he was an Associate Member of the Professional Staff at Schlumberger-Doll Research, Ridgefield, CT. He is currently a doctoral candidate in computer science at Stanford. His research interests include knowledge-based systems, object-oriented programming, and user interfaces.



Reid G. Smith (S'67-M'69) received the B.Eng. and M.Eng. degrees in electrical engineering from Carleton University, Ottawa, Ont., Canada, in 1968 and 1969, respectively, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1979.

He is the Director of Schlumberger Palo Alto Research, Palo Alto, CA. His current research focuses on object-oriented programming environments, knowledge-based system design, and AI applications in engineering. He has previously worked on knowledge-based systems for oilfield data interpretation, tools for knowledge-based system construction, and learning apprentice systems. He is the author of *A Framework For Distributed Problem Solving* (UMI Research Press, 1981).

Dr. Smith is a member of ACM, AAAI, CSCSI, and AAAS, and serves on the editorial board of *Expert Systems: Research and Applications*.

Bruce G. Buchanan received the Ph.D. degree from Michigan State University, East Lansing, where he was later on the faculty.

He is a Professor of Computer Science and Medicine (by courtesy) at Stanford University, Stanford, CA. In 1966, he joined Stanford University as a Research Associate in Computer Science. From 1971 to 1976, he was a Research Computer Scientist, while holding a National Institutes of Health Career Development Award. In 1976, he was appointed to his present position. He is currently working on the interpretation of data about the three-dimensional structure of proteins. His other research interests include: representing knowledge, reasoning about complex and uncertain situations, explaining lines of reasoning, and acquiring new knowledge.

Prof. Buchanan is on the editorial boards of *Artificial Intelligence*, *Machine Learning*, and *The Journal of Automated Reasoning*, and is Secretary-Treasurer of the American Association for Artificial Intelligence.