A Collaborative Interface for Editing Large Knowledge Bases

Loren G. Terveen and David A. Wroblewski

MCC Human Interface Laboratory
3500 West Balcones Center Drive
Austin, TX 78759
terveen@mcc.com and wroblewski@mcc.com

Abstract

A new generation of knowledge/databases is emerging. These systems contain thousands of objects, densely interconnected and heterogeneously organized, entered from many sources, both human and automated. Such systems present tremendous challenges to their users, who must locate relevant information quickly and add new information effectively. Our research aims to understand and support the knowledge editing task. The HITS Knowledge Editor (HKE) is an interface that supports browsing and modifying the CYC knowledge base (Guha & Lenat 1990). HKE has been designed to be a *collaborative* interface, following a set of principles for sharing tasks between system and user. We describe these principles and illustrate how HKE provides resources built according to those principles that collaborate with its users on a variety of knowledge editing tasks.

Introduction: problems with large knowledge bases

A new generation of knowledge/databases is emerging. These systems contain thousands or tens of thousands of classes and instances, densely interconnected and heterogeneously organized, entered from many sources, both human and automated. Such systems present tremendous challenges to their users, who must locate relevant information quickly and add new information effectively. Left to one's own resources, it is easy to get lost in data space (Carroll et al. 1990 and O'Shea et al. 1986) or make inappropriate or ineffective changes.

CYC (Guha & Lenat 1990) is an example of such systems. CYC consists of a representation language (CYCL) and a knowledge base expressed in that language. The knowledge base currently consists of approximately 35000 units (known in other systems as "frames" "schemata", etc.). Each unit consists of a set of slots containing a set of values. CYC units average 13 slots filled with 2 or 3 values, meaning that each unit bundles around 35 assertions.

Our research aims to understand and support the knowledge editing task. We consider knowledge editing to be a family of related tasks, rather than a single homogeneous task. Knowledge editing is different from ordinary data entry because the user must understand the structure and content of the knowledge base well enough to be able to locate information in a timely manner and add or modify information in harmony with the existing representation conventions; in this way it is much like programming. In this paper we consider two knowledge editing tasks, *browsing* and *entry*. Browsing, at its simplest, consists of determining the truth status of some assertion P; in reality, it involves acquiring a model of the relational structure of the knowledge base and landmarks from which important data elements can be found quickly. Entry, at its simplest, consists of setting the truth status of some assertion P; in reality, it consists of managing a coordinated series of changes to the knowledge base or the creation of a cluster of interrelated units each consisting of many slots and values. Browsing and entry are interleaved throughout a typical knowledge editing session.

Principles of collaboration

The HITS Knowledge Editor (HKE) is an interface that supports browsing and modifying the CYC knowledge base. This paper focuses on several of HKE's capabilities that illustrate our attempts to make it a *collaborative* system. This section defines collaborative systems and puts forth several design principles to which HKE conforms.

In collaborative systems tasks must be shared between systems and users based on their respective capabilities. In some sense, all interfaces at least attempt to be collaborative. What is needed is a set of principles that guide us in deciding *how* to divide tasks between system and user. These principles should emerge from general principles of communication and must acknowledge the highly asymmetrical abilities of people and computers. For this paper, the key principles are:

- do not force users to make decisions in a rigid order,
- provide resources that help users in making decisions,
- let users build their solutions to problems as modifications of prior solutions to similar problems,
- make *relevant* action possibilities apparent when the set of *possible* actions becomes large.

Collaboration on browsing tasks

Users can be overwhelmed with information when browsing complex data spaces. One of the motivating assumptions of our work has been that in such complex data spaces there is no one right way to view all the data, nor even a fixed set of ways. The best browsing display is highly dependent on the topic of the information being displayed, the task being performed with the data, and the experience of the user. We support this approach in HKE by allowing users to create, reuse, and share methods of viewing data.

There are several mechanisms that allow this in HKE; this paper discusses only one – *perspectives* on units. Perspectives are objects that dictate several parameters of tabular slot/value displays, such as attributes to display or suppress and the order in which to display them. Perspectives are constructed collaboratively by the system and the user.

Motivation for customizable perspectives

Several features of large-scale object systems affect browsing. First, each object may have many attributes. In CYC there are over 4000 defined slots, and, on average, 295 of them are relevant to a given unit. This means that users must manage a very large vocabulary for object features. Second, the vocabulary is complicated by the fact that slots often do not represent a simple set of unrelated attributes. Instead, relationships between the slots significantly constrain their semantics. For instance, the slot `parts` is known to be a generalization of the slot `subOrganizations`: any value filling `subOrganizations` automatically fills `parts`. Third, effective browsing hinges on seeing the appropriate subset of slots for the task at hand. Fortunately, entire groups of slots are devoted to specialized tasks such as consistency checking or bookkeeping; thus one may roughly partition the total set of applicable slots for a given unit into more manageable subsets for specialized tasks. This is what perspectives do.

This is clearly non-trivial work. A collaborative system can help by suggesting a reasonable initial perspective and allowing the user to improve it. The initial perspective might be generated from a variety of sources, such as a standard template, the user's past perspective on the same data, another user's perspective, or a model of the task the user is performing. This approach segments the work of browsing into three parts: specifying the format of the initial display, modifying the format of an existing display, and retrieving the format of past displays of the same object. HKE takes on the tasks of generating an initial display, storing modified displays, and retrieving those displays for reuse, while the user modifies the suggested perspective when it is ineffective.

Perspectives follow the principles of collaborative systems. First, HKE doesn't force the specification of perspectives before the act of browsing itself, but rather provides tools to evolve the perspective as the browsing proceeds. Second, HKE lets users build new solutions from previously generated solutions by always putting the user in the position of repairing rather than synthesizing perspectives.

Example: perspectives on Organizations

One section of the CYC knowledge base is devoted to representing organizations. The relevant class hierarchy is displayed in Figure 1.

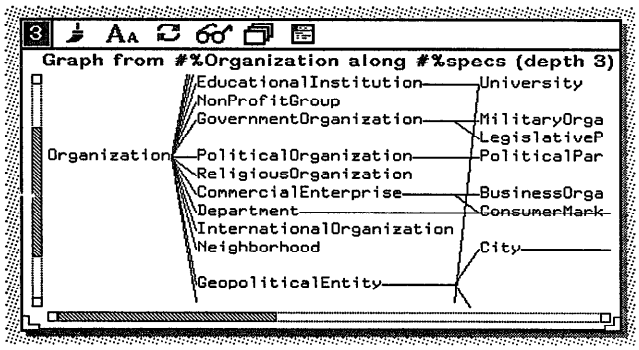

Figure 1: The `Organization` Class Hierarchy

The default perspective of any unit is created by composing the individual perspectives of all the classes of which the object is a member, in order of increasing generality. Thus, the default display of an instance of `CityOfAustinTX` would display the attributes given it because it was an instance of `City`, then those from `GeoPoliticalEntity`, then from `Organization`, and so on. To save space, empty slots are not displayed.

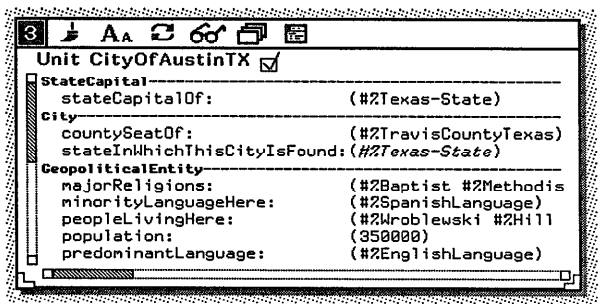

Figure 2: The unit `CityOfAustinTX`

This display may be unsatisfactory either because it is still too complex, eliminates important attributes that don't happen to be filled in the unit, or includes the right attributes in an unintuitive order. Each component of the composite perspective can be edited to change these features of the display.

Suppose after looking at `CityOfAustinTX`, the user decides that a number of slots always should be displayed, including `predominantLanguage`, `majorReligions` and `population`. This is accomplished by clicking on the

perspective labelled "GeoPoliticalEntity". Since the user has no custom perspective for GeoPoliticalEntity yet, one is created and initialized with the slots shown in the current display. A special perspective editor pops up to rearrange the new perspective. When the user is done, the new perspective is indexed against the user and the class GeoPoliticalEntity and then the unit CityOfAustinTX is redisplayed. Figure 3 shows the perspective editor and the resulting display of the CityOfAustinTX. All subsequent displays of instances of GeoPoliticalEntity will use this custom perspective.

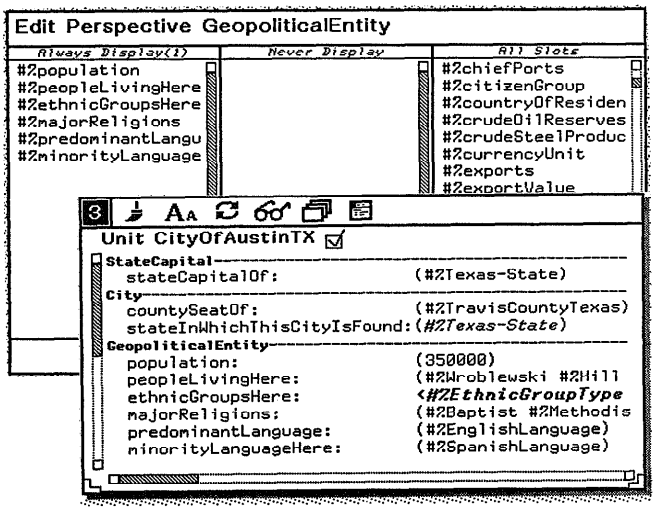

Figure 3: The perspective editor

Collaboration on entry tasks

Motivation

Basic knowledge entry tasks include (1) choosing which slots should appear on a unit, (2) specifying fillers for these slots, and (3) managing the creation of groups of interrelated units. Tasks (1) and (2) are made especially difficult by the sheer size of CYC: it is a formidable task to select 10 or 15 out of the nearly 300 slots that could appear on a unit, then choose several fillers from the hundreds or even thousands of legal values. Task (3) arises from the recursive nature of knowledge entry: when representing the organization MCC, for example, one may well want to create units that represent people employed there, their areas of expertise, etc. Keeping track of related representational tasks is a significant part of the overall entry task. The resources that HKE provides to collaborate on these tasks were designed in accordance with our principles of collaboration:

- *flexible decision making* – checklists package decisions that are relevant when creating a particular type of unit, agendas allow users to do unit creation tasks in whatever order they wish, and several repair facilities bring out information relevant to making a decision,
- *relevant action advertisement* – a display of the slots that a unit could have supports the user in choosing slots that should appear, and
- *build on previous solutions* – a unit is created by copying and editing an existing unit, and other similar units are presented as models from which the user can extract pieces to add to the new unit.

Example

We suppose that the user's task is to represent knowledge about the organization MCC. He has browsed through the section of the knowledge base representing organizations, identified ResearchOrganization as the right class for MCC, and has decided to copy-and-edit the unit UniversityOfTexasAtAustin to create MCC. Editing is initiated by clicking on the check mark displayed in the label line of the unit display (see figure 2). This causes HKE to construct and display a checklist for editing the unit.

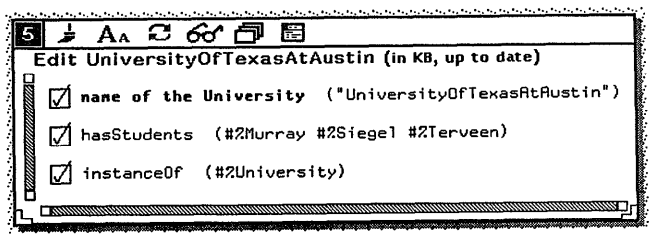

Figure 4: The task of editing UniversityOfTexasAtAustin

The initial resource

A checklist is a computerized version of the everyday to-do list. A checklist helps to organize an activity by reminding one what needs to be done and helping keep track of what already has been done. The checklist for editing a unit contains an *issue* (item) for each slot to appear on the unit. The initial set of issues is derived from the slots of the copied unit, here UniversityOfTexasAtAustin, using perspectives to filter and order the slots.

Checklists support flexible decision-making: the issues can be done in any order and can be revisited and modified any number of times.

Customizing the resource

The system-constructed checklist is a reasonable resource for editing a unit. It contains issues for specifying the value of a fairly small number of slots, filtered and ordered by the perspectives that apply to the unit. The user can access a menu that provides resources for customizing the checklist. These include: (1) **additional slots** – a display of slots the unit could have (in order from most to least specific and with uninteresting slots filtered out). The user can add an issue to the checklist for specifying the value of any of these slots by mousing it. (2) **model units** – a display of units that share characteristics of the unit being edited. The user can browse these units for slots and fillers that should appear on the unit being edited

and can add a slot-value pair to the checklist with a mouse gesture, thus customizing the checklist and answering the newly added issue at once.

In our example, the user will want to customize the checklist in several ways.

- Change the type of unit being created from `University` to `ResearchOrganization`. He does so by editing the answer to the "instanceOf" issue, replacing `University` by `ResearchOrganization`. This causes the "hasStudents" issue to be removed from the checklist, since a `ResearchOrganization` may not have the slot `hasStudents`. It also causes the set of **additional slots** to be recomputed.
- Add issues for specifying the `subOrganizations` and `organizationHasActivities` slots to the checklist. This has two effects. First, it signals that the user wants to take care of these two items as part of the task of representing MCC. Second, the user can state that he wants these changes to be recorded on the perspective used to construct the checklist, i.e., to add these slots to the `Organization` perspective. In this way, changes to a particular task resource will affect future browsing and entry of any instance of `Organization`.
- Customize the issues of the checklist. The text of the question may be changed, whether the issue is necessary or optional in completing the task can be indicated, and a textual explanation of the question can be entered.

After making these changes, the checklist looks like:

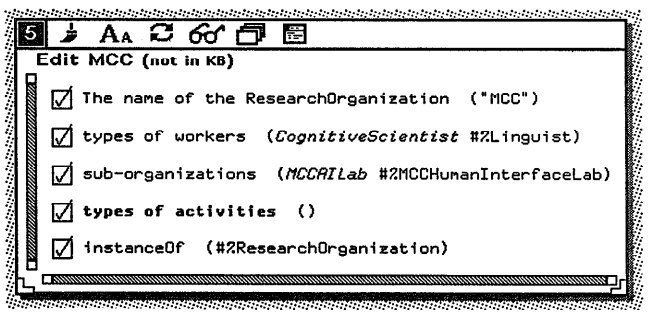

Figure 5: Customized checklist for editing `MCC`

Customizing the checklist illustrates the point that solutions should be built from prior solutions. Not only is the new artifact (the unit MCC) built on a previous artifact, but the main resource for creating the artifact (the checklist) evolves from the default constructed by the system. In addition, customizations to the checklist may cause the underlying perspective to be changed. The modified perspective is available for reuse and further evolution. This evolution results in knowledge entry resources that are abstractions of the pure example-based method. For example, if the perspective for `Organization` contains an issue for setting the value of the `subOrganizations` slot, then, even if the user selects an instance of `Organization` to copy-and-edit that does not specify a value for this slot, the checklist will contain an issue for it.

Managing the task context

As the user continues to enter knowledge about MCC, he will need to create additional units. For example, none of the sub-organizations of MCC or people who work at MCC are represented yet. When answering an issue, the user can press the HELP key to get a menu of all the units that could answer the issue. If the object the user is seeking does not yet exist, he simply can type in a new name for the unit he wants to create. This causes the system to (1) construct a checklist for creating an instance of the appropriate class, and (2) record this checklist on the task agenda.

The visible representation of a checklist serves as an **implicit agenda** of tasks associated with a checklist. For example, the "types of workers" issue of the "Edit MCC" task might refer to a unit that does not exist yet, `CognitiveScientist`. Such unmade units are displayed in *italics*, letting the user "read off" related tasks with a glance. These tasks can be worked on in any order and can be interleaved, illustrating the principle of allowing users to make decisions in whatever order they wish. The system manages certain types of dependencies between tasks. If the user performs the "Update" action on the "Edit MCC" task, the unit `MCC` is created, but the user is notified that since `CognitiveScientist` does not exist yet, the fact that it is one of the worker types of `MCC` cannot be asserted. When `CognitiveScientist` has been created, the system makes this assertion, and notifies the user that it has done so.

Repair

When a person cannot understand a communicative act well enough to respond as expected, he must engage in repair. For example, a user of HKE can have a problem in responding to an issue of a checklist when he does not know what object to supply. A menu of repairs is available for each issue. The repairs index the user into relevant sections of the knowledge base, direct the user to related tasks, or provide remedies for problems the system has detected with the user's answer. For example, if the user begins to represent the people who work at MCC (recorded on the `hasWorkers` slot), one repair would be to display the class `Person`, since fillers of `hasWorkers` must be instances of `Person`.

If the user chooses to display the collection `Person`, he can explore related knowledge using all the normal browsing facilities of HKE. One useful action is to find out the specializations of the concept `Person` using the Inspect Lattice command. This will trace out any relationship from a specified unit to a specified depth. In this case, the user wants to see the lattice constructed by traversing the specs relationship from `Person`.

However, when a user issues this or any other command, more goes on than meets the eye. First, command interpretation is done using a blackboard architecture (Cohen, McCandless & Rich, 1989). Second,

after the command has been fully interpreted but before the application program gets to execute the command, *angels* get a chance to inspect and possibly modify the command. An angel's knowledge consists of *anomalies*, problems that can occur in doing a particular task, and *strategies*, methods for repairing these problems. Thus, the system can repair the user's command specifications.

The angel competent about knowledge editing is named Hank. One of Hank's anomalies is HighBranchingFactor, which detects the condition that some units appearing in a lattice display have too many children. This is problematic because displaying a lattice with many highly branching nodes (1) can take a long time, and (2) can overwhelm the user with too much information. The anomaly is detected by doing a partial traversal of the lattice, locating all the units whose children exceeded the maximum branching factor. (The traversal can be done quite cheaply, so detecting the anomaly does not cause significant overhead.)

Thus, when the user issues the command to inspect the specializations of Person, Hank detects the HighBranchingFactor anomaly before the command is executed, modifies it to mark certain units not-to-be-expanded, and notifies the user what has been done.

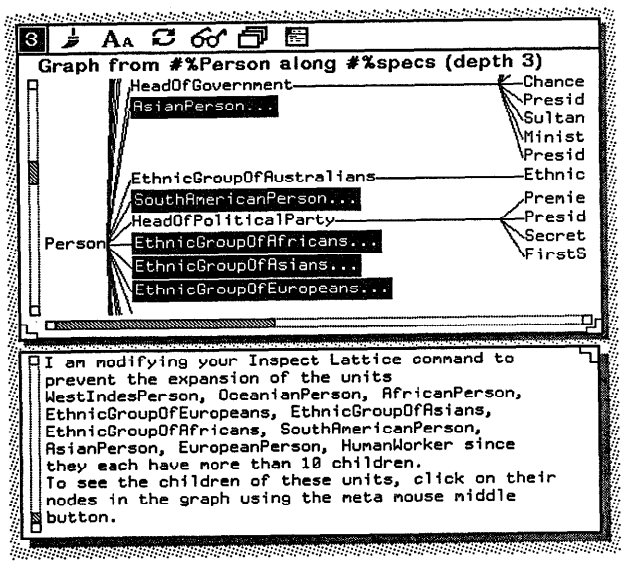

Figure 6: Angel intervening to modify a problematic action

Several comments are in order. First, the resulting lattice took seconds instead of minutes to produce and displays a manageable amount of information. Second, no options are taken away from the user. He still can expand any of the unexpanded units, but now will be aware of the cost of doing so. Third, we once again advertise potential actions, using reverse video to indicate units the user might wish to expand. Finally, we should emphasize that this anomaly could not be avoided by modifying the Inspect Lattice command to take an additional argument specifying units not to expand. Knowing which units not to expand requires knowing the structure of a particular section of the knowledge base, and since the user issued the command to find out about that structure, we cannot expect him to know it already.

Strengths and weaknesses

HKE's strengths are the set of collaborative resources it offers. Perspectives afford selective, task-sensitive, customizable views on complex objects. Checklists provide a flexible knowledge entry scheme and constitute implicit agendas of relevant tasks. Repair facilities help users overcome problems arising in their tasks.

A number of limitations of HKE's current collaborative facilities are not in-principle shortcomings: we "just haven't done them yet." However, there are several in-principle limitations as well.

Perspectives almost always are used to reduce the total information displayed. One possible disadvantage is that this could hide essential but infrequently used slots. In addition, we do not yet have any good schemes for managing large sets of perspectives. Extended use of HKE means generating many perspectives and evolving old perspectives in response to changing task demands.

Checklists are less useful in informal activities. The system can provide assistance in responding to individual issues of the checklist – as HKE does by indexing the user into relevant sections of the knowledge base – only if the issues themselves are relatively formal. Even in less formal activities, however, checklists perform valuable organizational and reminding functions.

Checklists are textual resources; however, many knowledge editing tasks are best expressed in a non-textual fashion. For example, laying out the ontology of a new domain usually consists of graphing out collections, relationships between the collections, and attributes they can have.

Finally, one might consider our reliance on users to customize resources a limitation – aren't we just adding to their burden? We advance four reasons why we expect users to do the customization we have described here.

First and most important, the work is done jointly with the system, in service of and in the context of the user's tasks. Changing the display of an object or adding a new issue to a checklist is done to help the user achieve his comprehension or editing goals and builds from a set of resources supplied by the system. Second, since a user may view or edit particular types of knowledge at infrequent intervals, the work invested in creating customized perspectives is paid off when he once again returns to a section of the knowledge base. Third, since perspectives are distributed along the generalization hierarchy, customizations made for one unit apply to whole classes of related units. Finally, since perspectives are stored in a shared knowledge base, views built up by one user are available to other users, too.

Related work

HKE is not a knowledge acquisition tool. Such systems address issues like techniques for eliciting knowledge from an expert and acquisition of domain or problem-solving method specific knowledge. For example, Protos (Bareiss, Porter, & Murray 1989) acquires knowledge used to do heuristic classification, TDE (Kahn et al. 1987) acquires troubleshooting hierarchies, and Luke (Wroblewski & Rich 1988) acquires linguistic knowledge. HKE provides functionality for browsing, entry, and task management that could be utilized by any of these tools.

RABBIT (Tou et al. 1982) introduced the use of perspectives for browsing; however, our implementation affords more extensive control of the display of information (but makes perspective management a task for the user) and applies perspectives to entry tasks, too. Object Lens (Lai, Malone, & Yu 1988) used checklist-like objects ("templates") for data entry; however, we use perspectives to filter the slots that appear in a checklist and provide repair facilities to help users fill out checklists. KREME (Abrett & Burstein 1987) provides browsing and entry facilities similar to HKE. It uses agenda-like structures to keep track of editing tasks. BACKBORD (Yen, Neches, & DeBellis 1988) focuses on retrieval of objects from knowledge bases and browsing of class hierarchies. It uses checklists to support several simple browsing and entry tasks. The distinguishing mark of our work is to make the system a more active collaborative partner.

Finally, the Framer system (Lemke 1989) is a good example of a collaborative interface. Our implementation of checklists is an adaptation of his.

Future work

Further development will come along three fronts. First, existing resources will be extended and several new types will be added. For example, we will use checklists to support several types of tasks more complicated than editing a single unit. Second, we will refine the principles of collaboration offered here. Finally, we will do empirical studies of users editing knowledge in HKE in order to test these principles. We already have anecdotal evidence that HKE succeeds in its aims, through our own use and through its use by a small community of users within MCC and its shareholder companies. Our work to date has given us a qualitative understanding of the nature of knowledge editing. We now are in a position to carry out empirical studies to test our hypotheses about collaboration for the knowledge editing task.

Acknowledgements

We thank Will Hill, Tim McCandless, Elaine Rich, and Steven Tighe of MCC and Robert Simmons and Bruce Porter of UT-Austin for their thoughtful review and discussion of this paper and the ideas presented in it.

References

Abrett, G., & Burstein, M.H. 1987. The KREME Knowledge Editing Environment. *International Journal of Man-Machine Studies* 27:103-126.

Bareiss, R., Porter, B.W., & Murray, K.S. 1989. Supporting Start-to-Finish Development of Knowledge Bases. *Machine Learning* 4:pp. 259-283.

Carroll, J.M., Singer, J.A., Bellamy, R.K.E., & Alpert, S.R. 1990. A View Matcher for Learning Smalltalk. In Proceedings of the 1990 ACM Conference on Human Factors in Computing Systems, 431-437. Seattle, WA: ACM Press.

Cohen, R.M., McCandless, T.P., and Rich, E.A. 1989. A Problem Solving Approach to Human-Computer Interface Management, MCC Technical Report Number ACT-HI-306-89, Microelectronics and Computer Technology Corporation. Austin, TX.

Guha, R.V. & Lenat, D.B. 1990. *Building Large Knowledge Based Systems.* Reading, MA: Addison-Wesley.

Kahn, G.S., Breaux, E.H., DeKlerk, P., & Joseph, R.L. 1987. A Mixed-Initiative Workbench for Knowledge Acquisition. *International Journal of Man-Machine Studies* 27:167-179.

Lai K.Y., Malone, T.W., & Yu, K.C. 1988. Object Lens: A "Spreadsheet" for Cooperative Work. *ACM Transactions on Office Information Systems.* 6:332-353.

Lemke, A. 1989. Design Environments for High-Functionality Computer Systems. Ph.D. diss., Department of Computer Science, The University of Colorado at Boulder.

O'Shea, T., Beck, K. Halbert, D., & Schmucker, K. 1986. Panel: The Learnability of Object-Oriented Programming Systems. Object-Oriented Programming Systems, Languages, and Applications: OOPSLA 86 Conference Proceedings, 502-503. New York, NY: ACM Press.

Tou, F.N., Williams, M.D., Fikes, R.E., Henderson, D.A., & Malone, T.W. 1982. RABBIT: An Intelligent Database Assistant. In Proceedings of the National Conference of the American Association for Artificial Intelligence, 314-318. Philadelphia, PA: American Association for Artificial Intelligence.

Wroblewski, D.A., & Rich, E.A. 1988. Luke: An Experiment in the Early Integration of Natural Language Processing. In Proceedings of the Second Conference on Applied Natural Language Processing, 186-194. Austin, TX: ACL Press.

Yen, J., Neches, R., & DeBellis, M. 1988. BACKBORD: Beyond Retrieval by Reformulation. In Architectures for Intelligent Interfaces: Elements and Prototypes, 219-235. Monterey, CA.