

An Integrated Environment for Knowledge Acquisition

Jim Blythe
Information Sciences Institute
University of Southern
California
Marina del Rey, CA 90292
blythe@isi.edu

Jihie Kim
Information Sciences Institute
University of Southern
California
Marina del Rey, CA 90292
jihie@isi.edu

Surya Ramachandran
Information Sciences Institute
University of Southern
California
Marina del Rey, CA 90292
surya@isi.edu

Yolanda Gil
Information Sciences
University of Sout
California
Marina del Rey, CA
gil@isi.edu

ABSTRACT

This paper describes an integrated acquisition interface that includes several techniques previously developed to support users in various ways as they add new knowledge to an intelligent system. As a result of this integration, the individual techniques can take better advantage of the context in which they are invoked and provide stronger guidance to users. We describe the current implementation using examples from a travel planning domain, and demonstrate how users can add complex knowledge to the system.

1. INTRODUCTION

An important area of user interface research is the development of practical approaches that enable users to add new knowledge to an intelligent system, which would bring computers closer to meeting the challenge of end-user programming. These acquisition interfaces need to have many intelligent capabilities in order to support the complex dialogues that they must conduct with the user, integrate the new knowledge with existing knowledge, and make appropriate generalizations.

In past research, we developed several acquisition interfaces [10, 2, 18], all using EXPECT as an underlying framework for knowledge representation and reasoning [17]. Each interface addressed different issues and helped the user in different ways as they add knowledge to a system, yet none could individually claim to be able to support a user appropriately. This paper presents an integrated acquisition interface that combines these approaches, providing stronger guidance to users.

The paper begins with a brief overview of the individual pieces of knowledge acquisition research that our interface integrates. Then we show an example scenario in which a user interacts with the implemented tool. Next, we analyze the different kinds of knowledge that users need to specify and discuss the challenges they pose to a knowledge acquisition tool. We then describe the components of our implemented system in detail, highlighting the benefits of the integrated acquisition environment.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IUI'01, January 14-17, 2001, Sante Fe, New Mexico.
Copyright 2001 ACM 1-58113-325-1/01/0001 ...\$5.00.

2. PREVIOUSLY DEVELOPED TOOLS AND TECHNIQUES

Our past research on knowledge acquisition can be described in terms of typical concerns that users may have about adding new knowledge to an intelligent system:

- *Users do not know formal languages.*
We have developed **English-based editors** that allow users to modify English paraphrases of the internal, more formal representations [3]. Users can only select the portions of the paraphrase that correspond to a valid expression in the internal language, and pick from a menu of suggested possible replacements for that portion. This approach enables the system to communicate with the user in English while circumventing the challenges of full natural language processing.
- *How do users know where to start?*
Intelligent systems use knowledge to perform tasks for the user. If the acquisition tool has a model of those tasks, then it can reason about what kinds of knowledge it needs to acquire from the user. We have developed acquisition tools that **reason about general task models** and other kinds of pre-existing knowledge (such as domain-specific knowledge that is initially included in the system's knowledge base) in order to guide users to provide the knowledge that is relevant to those tasks [2]. Our work has concentrated on plan evaluation and assessment tasks, but could be used with other task models.
- *How do users know that they are adding the right things?*
Users need to know that they are providing knowledge that is useful to the system and whether they have given the system enough knowledge to do something on its own. Our approach is to use **Interdependency Models** that capture how the individual pieces of knowledge provided work together to reason about the task [10]. These Interdependency Models are derived automatically by the system, and are used to detect inconsistencies and missing knowledge that turn into follow-up questions to the user. Users are often not sure whether they are on the right track even if they have been making progress, and we have found that it is very useful to show the user some aspects of this Interdependency Model (for example, showing the substeps involved in doing a task) and how it changes over time.
- *How do users figure out what to do next?*
A system that responds to the user with many sensible follow-up questions is helpful but not sufficient.

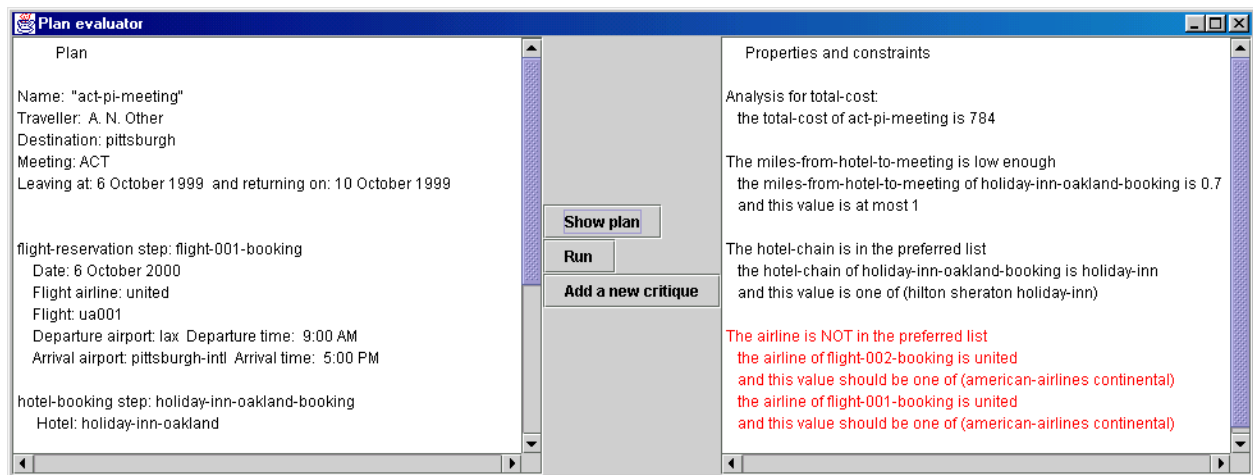


Figure 1: The acquisition interface is invoked from the application, in this case a travel assistant. A travel plan is shown on the left. On the right, the travel assistant has used its knowledge base of the user's preferences to generate salient properties important to the user (such as overall cost) and constraints (such as upper limits on the hotel rate.)

When users add a sizeable amount of knowledge, the system is likely to respond with a number of follow-up questions. Users also need help in formulating answers. We have **categorized and organized different kinds of questions**, as well as the possible actions that the user can take in order to address them [10, 8]. By understanding the nature of the questions, the system can help the user understand and prioritize these questions. By understanding the context in which each question was generated, the system can make good guesses about how the user may answer them.

- *It takes several steps to add new knowledge, so users will be easily lost.*

Entering new knowledge, even of moderate complexity, requires making several individual changes. Users often do not realize and/or forget the side effects of these individual changes that must be followed up (anyone who has ever programmed can relate to this). We have developed a framework to guide users through typical sequences of changes or **Knowledge Acquisition Scripts** (KA Scripts) [18]. As the user interacts with a KA Script and enters knowledge step by step, additional KA Scripts may be activated that contain follow-up questions about that new knowledge. The system not only points out the next steps but indicates its best guess about the knowledge the user needs to enter based on the knowledge that is already in the system. We have developed a library of general-purpose KA Scripts [18], as well as with scripts customized to the general task models mentioned above [2].

In the next section, we show through an example the capabilities of the acquisition interface that integrates these techniques and tools.

3. WALKTHROUGH EXAMPLE OF A USER'S INTERACTION

This section illustrates through a simple example how a user interacts with our integrated acquisition tool. In these examples, the user wants to tell a travel assistant about her personal constraints in selecting hotels, airlines, or making car rental reservations. Through a series of snapshots, we show how the user specifies that the cost of a hotel should not be more than \$120 per day, and should be less than the maximum rate specified in the contract that the trip is charged to. Later in the paper we show how a user adds more complex constraints.

The user invokes the acquisition tool from the application, in this case the travel assistant interface shown in Figure 1. The left side of the application window shows the travel plan, composed of several *steps* that specify flights, hotel reservations, and car rental reservations¹. The assistant presents to the user on the right-hand side salient aspects of the plan that the user indicated in the past as being important. Some of the items shown are *properties* of the travel plan that the system derives from the information shown in the left hand side. The total cost of the plan is an example of such a property, and the knowledge base specifies how it is calculated based on the daily rate of the hotel reservation, the number of days that the user will be away, the cost of the flight, etc. Based on these properties, the user can define *constraints* to express their travel preferences. For example, the user prefers to use Holiday Inn and the system is pointing out that this particular travel plan complies with this constraint. In short, the knowledge base of this assistant includes knowledge about how to derive these constraints and properties from a travel plan like the one shown on the left hand side.

First, the user wants to tell the system not to book her in hotels that cost more than \$120 per day. The user invokes the acquisition interface to add this new constraint. In Figure 2 the user names a new property `hotel rate`, and the system asks some questions about the nature of this new constraint. Figure 3 shows the window used to ask one of these questions, where the user is indicating that

¹Notice that the tool described here is not concerned with generating itineraries and travel plans, but could be integrated with a travel planning system that would use the same knowledge acquired by our tool to guide its selection of choices in generating a travel plan.

this is a constraint about the hotel's daily rate and not its weekly rate or exercise facilities. The user does this by choosing from the options presented in the bottom half of the window, which are generated by the system based on what the knowledge base says about hotels.

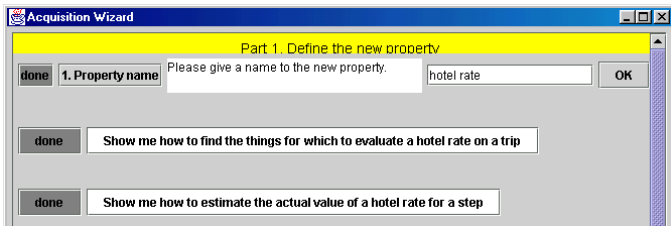


Figure 2: The system begins by guiding the user to define a new constraint.

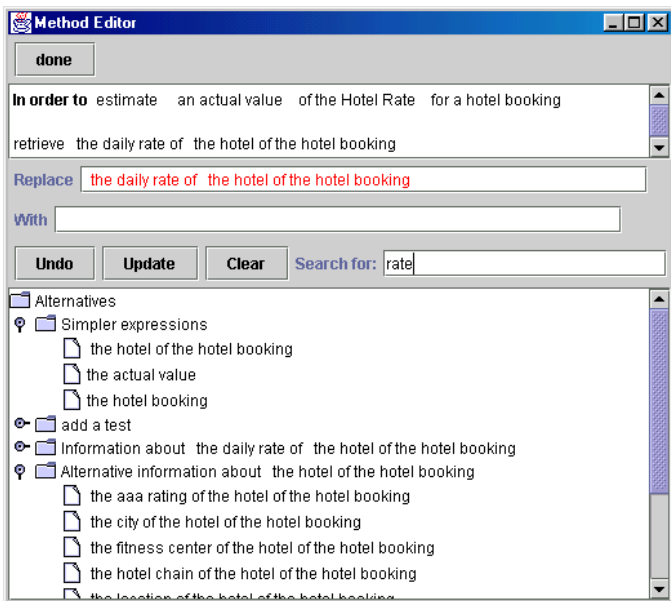


Figure 3: The user selects the daily rate among all the choices that the system presented based on its current knowledge about hotels.

The system continues with additional questions about constraints based on the new property, shown in Figure 4. Since it knows that daily rates are numeric, it asks whether the user wants to set an upper or lower limit for this amount. In Figure 5, the user indicates that the maximum amount is \$120 by typing that number in the appropriate place within the sentence that the system presents in the screen.

Later on, the user wants to add a similar but more complex constraint: she would like the system not to book her in hotels that have daily rates that go over a maximum rate specified in the contracts that she will charge for the trip. She follows the same basic steps shown in Figures 2 to 4, except that the upper limit is specified as in Figure 6. In this case, the system is showing the user all the things it knows about contracts (e.g., they have start and end dates, a funding agency) but there is nothing about a maximum allowed hotel rate. The pop-up window allows the user to define this. It also shows that the system expects the value to be a number based on what it knows so far about this constraint.

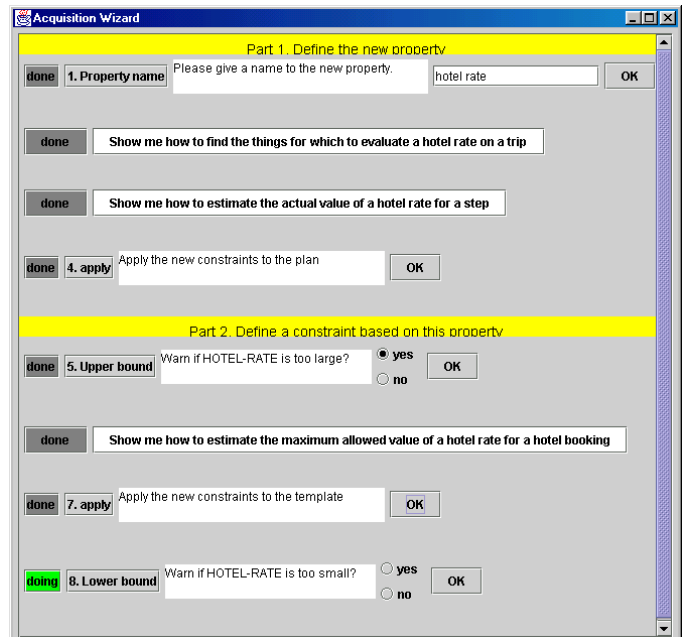


Figure 4: The system continues asking questions about the allowed values for the hotel rates.

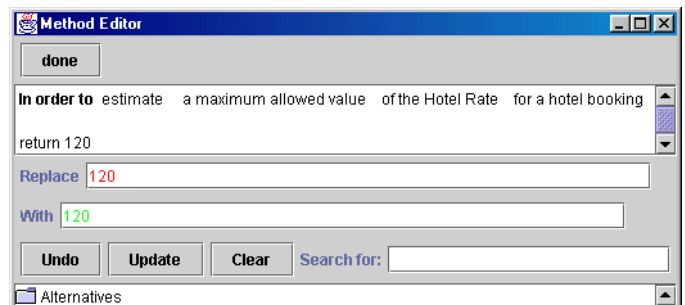


Figure 5: The user specifies a maximum hotel cost of \$120.

The user has now finished defining this new constraint. However, the system realizes that in order to check this constraint it needs to know what is the maximum allowed hotel rate for all the contracts that it knows about. Figure 7 shows how the user specifies the maximum rate for the EXPECT contract.

The system is providing significant assistance to the user in several ways throughout this scenario:

- the system *isolates the user from the internal formal representation* of the knowledge being entered. For example, the internal representation generated from the interaction in Figure 3 is

```
((capability (estimate (obj (?v is (spec-of actual-value))) (of (?c is (spec-of hotel-cost))) (for (?s is (inst-of step)))) (body (r-daily-rate (r-hotel ?s))) (result-type (inst-of number))))).
```
- the system *helps the user get started* by presenting initial questions based on the kinds of knowledge that can be specified for the travel advisor, in this case new constraints and properties.

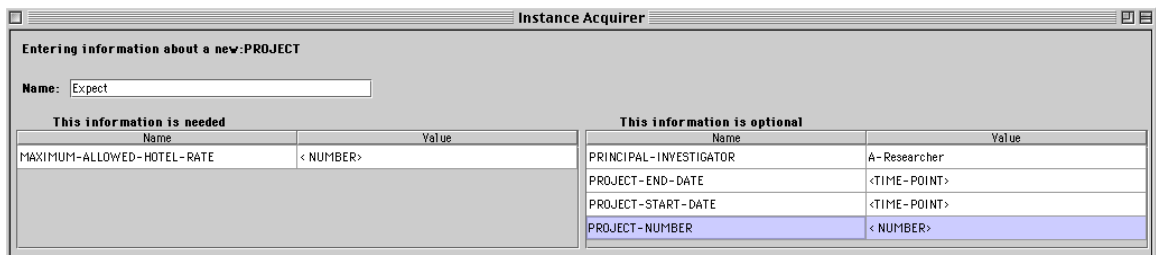


Figure 7: The system asks about the maximum allowed hotel rate for existing contracts.

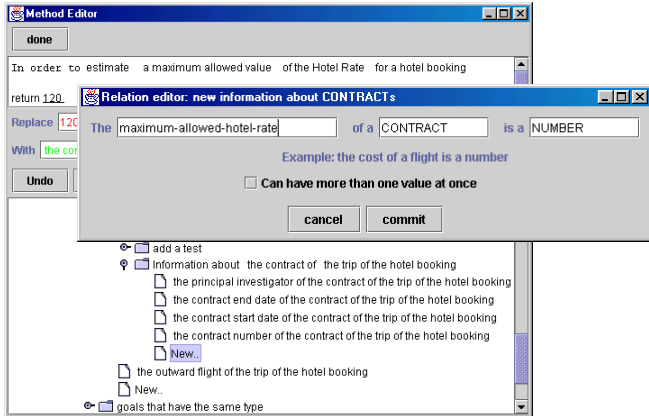


Figure 6: The user tells the system that contracts may specify a maximum hotel rate.

- the system *asks the user follow-up questions* that ensure that the new knowledge is usable by the system. For example, the system asks the user to extend the system's knowledge about contracts with regards to allowed hotel rates.
- the system *makes intelligent guesses* about how the user might answer each question. For example, based on what the user has already said the system shows that the default value of a maximum allowed hotel rate is a number.
- the system *brings to bear relevant background knowledge* that it already has about general kinds of constraints and preferences and how to check them. For example, it knows that some constraints are relevant to a whole trip while others concern only some portions, and it knows that some constraints are numerical and that arithmetic operations can be used to express those constraints.

Although the examples used in this paper are drawn from travel planning, the issues are motivated by our work in complex real-world planning domains, including logistics military planning, air campaign plan evaluation, course of action critiquing, and more recently biological weapon capability assessment. In these and other domains, users find great value in intelligent assistants that analyze a plan under consideration (created often by hand) and point out salient facts or violated constraints. This helps them evaluate and assess tradeoffs between different options, much in the way in which a travel plan is evaluated in our examples. The knowledge in these systems needs to be adapted to specific requirements of the situation at hand, user preferences, or novel practices. Knowledge acquisition tools that enable users to define constraints and preferences are key to the successful deployment of these and many other intelligent systems.

4. WHAT KINDS OF KNOWLEDGE NEED TO BE ACQUIRED?

Figure 8 summarizes the different kinds of knowledge that a user may want to specify in this type of application. In terms of the knowledge acquisition techniques that we believe are needed, we distinguish three main categories of knowledge:

Adding data

- **Situation-specific data**
Example: the location of the meeting that the user is attending
- **Persistent data**
Example: the user's home address

Adding object classes

- **Additional features of existing object classes**
Example: contracts have an upper limit on the hotel rates allowed
- **Additional object classes**
Example: security clearances, which are needed by the traveller for some meetings

Adding choice constraints and preferences

- **A constant criterion**
Example: the hotel cost should be less than \$120
- **A variable criterion**
Example: the hotel cost should be less than the maximum hotel rate allowed by the contract that is being charged for the trip
- **A tradeoff affecting only one decision**
Example: rent cars only from Hertz
- **A complex tradeoff affecting only one decision**
Example: rent a car only when using taxis to move around would be more expensive
- **A tradeoff affecting several decisions**
Example: choose hotels within walking distance, otherwise rent a car
- **Tradeoffs involving several criteria**
Example: use United, pick non-stop flights, but if United has no direct flights then still prefer United to a non-stop with another airline

Figure 8: Different kinds of knowledge need to be acquired in this kind of application. Constraints and preferences are the hardest kind, especially when they involve complex tradeoffs affecting several choices.

- **Data** denotes specific object instances or constants (e.g., UA flight 22 departs from LAX at 12:00PM and arrives Madrid at 9:20AM.). This is perhaps the easiest for users to specify. They can be effectively acquired through form-filling interfaces. In the extreme, some of this data acquisition can become extremely complex, since in some of the applications we have seen object instances that are composed of several hundred assertions. Model-based interfaces that declaratively represent object classes and the associated interfaces

have been used to acquire object instances of medium complexity [15]. The design of effective tools to acquire complex object instances remains largely an open research issue, in our view one that has more to do with the HCI aspects of the interface rather than providing more intelligent assistance.

- **Object classes** refer to general descriptions of object categories and the relations that exist among them (e.g., a flight has an origin and a destination). This is sometimes called an ontology. A variety of ontology editors have been developed over the years [1, 19, 6, 16, 9], many of them include facilities to enter data as well. Other tools have focused on related issues such the elicitation of attributes and differentiating features [7, 4], and the detection of inconsistencies in ontological descriptions [13]. Although many of these tools use graphical interfaces to show class hierarchies and relations, it appears that hypertext interfaces may be more practical for sizeable knowledge bases.
- **Constraints and preferences** specify how the user would like the system to make choices in cases when there are alternative options. This is perhaps the most difficult kind of knowledge to acquire, and has been the main focus of our work. Consider the example of renting a car only when using taxis is more expensive. This requires finding the distances between airport, hotel, and meeting locations to estimate the cost of taxis, figuring out how many days the car is rented to estimate the cost of the rental car, and comparing the two total amounts. This is problem solving (procedural) knowledge, which in this case is reasoning with the information provided (e.g., the meeting location) in order to derive more complex abstractions (e.g., the total cost of taking taxis). Problem solving knowledge is quite difficult to acquire, and is the end-user programming challenge as described in [5]. Because it uses information from the object classes and the data, it needs to be specified in a way that is consistent with those. Because it is generating intermediate abstractions and using them to generate more complex ones, it involves a number of steps and substeps that have to fit together precisely. We aimed to address these issues in our past work, especially the research on Interdependency Models [10, 8]. Alternative approaches learn from specific examples provided by users as they perform a task, and include programming by demonstration, learning apprentices, case-based reasoning, and feature-based induction.

Some acquisition interfaces have been developed to acquire some of the kinds of preferences and constraints that do not involve problem solving knowledge or very simple forms of it [14, 12]. These approaches do not address the acquisition of the knowledge involved in analyzing more complex constraints and tradeoffs that are ubiquitous in many decision making applications. In addition, although we have used the specification of preferences and constraints in planning tasks to motivate the need for supporting the acquisition of different kinds of knowledge we are interested in acquisition interfaces that use techniques that can be generally applied in many tasks and domains. In tool design there is often a tradeoff between generality and ease of use, in which a more general tool can be harder to use for particular applications than more specific tools designed for those applications. An integrated system such as ours can combine the benefits of powerful general tools with those of application-specific ones. This is because the more general components in the system can provide power to the more application-specific components, while the specific components provide better

contextual information to the general components. As we show in the next section, this is the case with the Acquisition Wizard, which makes use of application-specific task models, and the other components, which are general.

The acquisition environment that we have developed supports the acquisition of all the different types of knowledge described in this section. As we present its components in the next section, we will highlight the benefits of having an integrated system in terms of how the acquisition of different kinds of knowledge can be supported.

5. INTEGRATING KNOWLEDGE ACQUISITION TECHNIQUES

Figure 9 shows an overview of the different interface components that are integrated within our system and how they interact with each other. Many of the component tools become more powerful and easier to use in the integrated system because of the information that is shared between them.

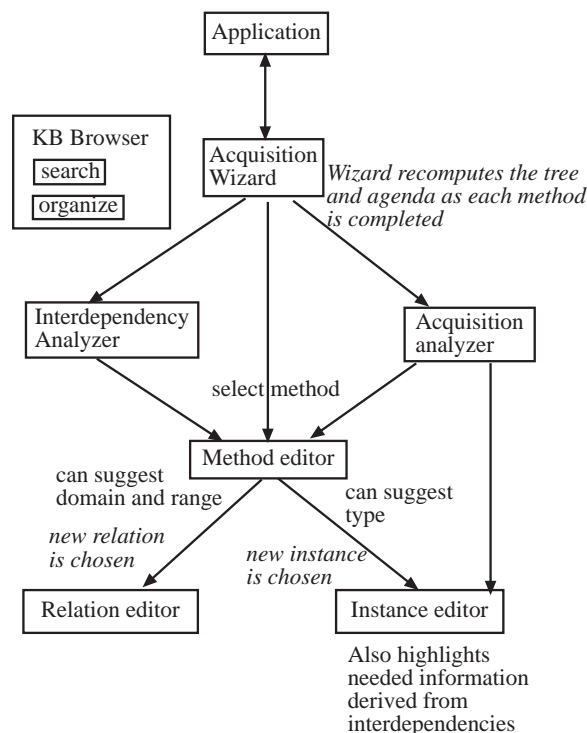


Figure 9: Overview of the integrated interface components and their interactions.

Acquisition Wizard

When the acquisition tool is invoked from the application, the **Acquisition Wizard** manages the initial interaction with the user shown in Figures 2 and 4. The wizard uses a general task model of plan evaluation that represents general classes of constraints and properties, and uses KA Scripts to organize the questions that it needs to ask the user in order to classify the new constraint appropriately. This task model includes an ontology that organizes different classes of plan evaluation criteria, shown in Figure 10. The task model also includes problem solving knowledge about how each class of criteria is evaluated.

Although the ontology shown in Figure 10 is not complete, and

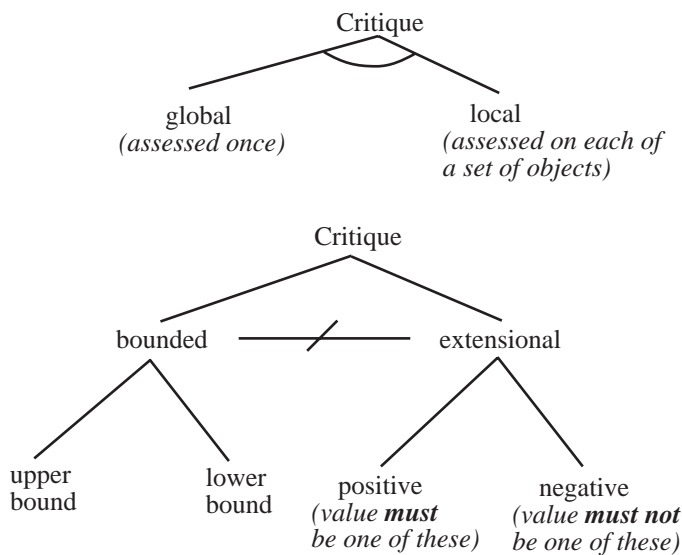


Figure 10: Part of the ontology of types of plan evaluation used by the Acquisition Wizard to classify and provide help for a new critique.

there will be cases where it does not help the user, it provides valuable guidance in many cases. A more inclusive ontology that includes planning resources can be found in [2], as well as more details about the techniques used. The ontology shown here divides the set of evaluations for which the wizard can provide help in two different ways. First, the evaluation can be either *global*, meaning that the property is computed once for the whole plan (e.g., the total cost) or it can be *local*, meaning that the property is computed for each one of a set of objects found in the plan (e.g., the length of each flight). An example of problem-solving knowledge associated with the ontology is that a local evaluation is made by iterating over the objects for which the property should be measured and recombining the individual results. Another piece of problem solving knowledge says that global evaluations are made by checking the presence or absence of that property in the overall plan. The second way that evaluations are classified in the ontology concerns how to check if the plan or object satisfies the evaluation. If an evaluation is *bounded*, then the value is checked against a threshold, either a maximum value, a minimum value or both. In our example, the hotel rate is a bounded evaluation. If an evaluation is *extensional*, then the value is checked against a set of values generated for the test. In the positive case the value must be in the set and in the negative case it must not be in the set. Either or both checks can be made for an evaluation. For example, expressing a constraint to rent only from Hertz is a positive extensional preference, while a constraint to avoid flying United is a negative extensional preference.

The wizard takes into account the answers already provided by the user to decide what additional questions it should ask. After the user finishes answering the first few questions, shown in Figure 2, the wizard classifies the kind of property being entered. In this case has a numeric value, so the wizard checks for a bounded evaluation by asking the user if the value can be too high and/or too low, as shown in Figure 4. Otherwise the wizard checks if the evaluation is extensional by asking the user if there are values that a plan should or should not have to satisfy the constraint. The wizard uses the problem solving knowledge specified in the task model to present the user with a template of the knowledge that needs to be speci-

fied, such as how to specify a maximum value (Figures 6 and 5). Although the wizard follows the structure of the ontology to generate its questions, the interaction with the user is done in a way that hides the complexity of the underlying task model.

Method Editor for Problem Solving Knowledge

The **Method Editor** is invoked to help the user add problem solving knowledge. It can be invoked by the Acquisition Wizard (as is done in our example for the windows in Figures 3, 5, and 6) which uses the general problem solving knowledge in the task model to create an initial template of the method. The method editor can be invoked in other contexts (from the Interdependency Analyzer and from the Acquisition Analyzer as we will describe below), and in all cases the editor uses that context to generate an initial template of the method. It can also be invoked by the user directly to update a method created previously. The method editor is always invoked with some initial template or specification of the problem solving knowledge to be added by the user.

At the top of the window, the method editor displays an automatically generated English description of that initial specification of the problem solving knowledge. The editor keeps track of what subexpressions in the formal language result in each substring of the English paraphrase (for example, $(r\text{-hotel } ?s)$ results in the hotel of the step). The paraphrase is mouse sensitive, but the user can only select meaningful portions of it. When the user has selected the part of the paraphrase that she wants to change, the method editor analyzes the formal subexpression that generated it and displays a tree view of the possible alternatives, in English, in the lower window. The alternatives are automatically generated by an algorithm that ensures that choosing any one would maintain the syntactic consistency of the method, as shown in [3].

In designing our integrated system, we were able to ensure that the method editor never starts with a blank slate, but instead always has some initial specification (even if generic or template-like) of the problem solving knowledge to be added. This is an important benefit, since the integrated system is able to provide a structured English editor for problem solving knowledge that would not be possible in other environments.

Relation (and Concept) Editor

When the alternatives presented by the method editor do not include what the user would like to specify, the user can select "other" and a **Relation Editor** is invoked. In our example, this was done in Figure 6 when the user found that contracts did not have maximum allowed hotel rates. Our integrated tool allows the user to switch seamlessly between adding problem-solving knowledge and extending the descriptions of objects in the knowledge base, which is often necessary.

An additional benefit of the integrated tool is that it can use the context (in this case the method being edited) to propose a default domain and range for the new relation (contract and number), further simplifying the user's task. Our integrated interface does not yet include a concept editor, although it would be integrated in a similar way.

Acquisition Analyzer

Throughout the interaction with the user, an **Acquisition Analyzer** keeps track of all the pending questions. Many of these questions result from analyzing the Interdependency Model derived by the system, as described in [10, 8]. The Acquisition Analyzer keeps track of the reason for each question to the user. For example, the Interdependency Model states what information about objects is needed to check the new constraint. In our example, it will notice

that the maximum allowed hotel rates of contracts is used in the constraint's definition, whereas the end dates or funding agencies of contracts are not. Based on this, the Acquisition Analyzer will create a question for the user about each existing contract.



Figure 11: The Acquisition Analyzer keeps track of all the pending questions and provides suggestions based on the types of questions.

The user can view these questions through an agenda-based interface, as shown in Figure 11. It shows an item "I need to know the maximum allowed hotel rate for 'expect contract' because...". The Acquisition Analyzer can reorganize the items in several ways: based on the type of question, based on the object, based on the kind of information requested, etc. For each item, a list of suggestions is attached. In this example, one suggestion is to add the information about the hotel rate for the EXPECT contract. When the user selects one of the options, the appropriate editor is automatically invoked for either problem-solving or object knowledge. In Figure 11, when the user selects the option "provide information about 'expect contract'", the system invokes the Instance Editor so that the user can add the information needed (its maximum allowed hotel rate). The integrated interface enables the user to relate editing activities to the system's questions seamlessly, and to select the appropriate editor based on the kind of question/agenda item being addressed.

Instance Editor

The **Instance Editor** is invoked when the user needs to enter information about particular objects, such as a new location or meeting. In our example, the Instance Editor is used to specify the maximum allowed hotel rate of existing contracts, as shown in Figure 7, and could be invoked from the Acquisition Analyzer's agenda.

The Instance Editor shows on the left hand side what information is needed about the particular object to check the constraints and properties defined by the user so far. On the right hand side it gives the user the option to specify additional things, but it notes that these are not currently needed. The editor makes this distinction by analyzing the Interdependency Model which is maintained by the Interdependency Analyzer as the system acquires new problem solving knowledge. This feature, enabled by our integrated system, is not provided by other instance editors.

Interdependency Analyzer

So far, the components of our acquisition interface that have been described are fairly easy to use, but the information they acquire is relatively simple. More advanced users may explore an additional component of the interface that enables them to enter more complex constraints.

To support the acquisition of more complex constraints, users need to add substantial amounts of problem solving knowledge as we discussed earlier. The **Interdependency Analyzer** guides the

user in entering problem solving knowledge for more complex constraints than those shown in our scenario. Consider, for example, that the user would like to enter the constraint discussed earlier of only renting a car if it is cheaper than using taxis. In this case, the Acquisition Wizard would guide the user through the same steps shown in our example. At some point, the system will invoke the Method Editor and ask the user to specify how to estimate the taxi cost for a trip. This involves the following steps and substeps:

- to estimate the taxi cost for a trip, add:
 - *estimate taxi cost for a trip from the airport to the hotel*
 - compute the taxi cost between the airport and the hotel
 - and multiply the result by 2
 - *estimate taxi cost for a trip from the hotel to the meeting*
 - compute the taxi cost between the hotel and the location of the meeting
 - and multiply the result by the duration of the meeting in days (the number of times to visit the meeting place)

To specify all these substeps, the user needs to add several pieces of problem solving knowledge. In order to guide the user in specifying these steps, the system needs to understand what pieces are missing at a given time, what pieces are related and how, and whether there are inconsistencies among them. Our acquisition interface does this by analyzing the Interdependency Model, specifically the portions of it that show how problem solving pieces are related as steps and substeps within a problem solving tree.

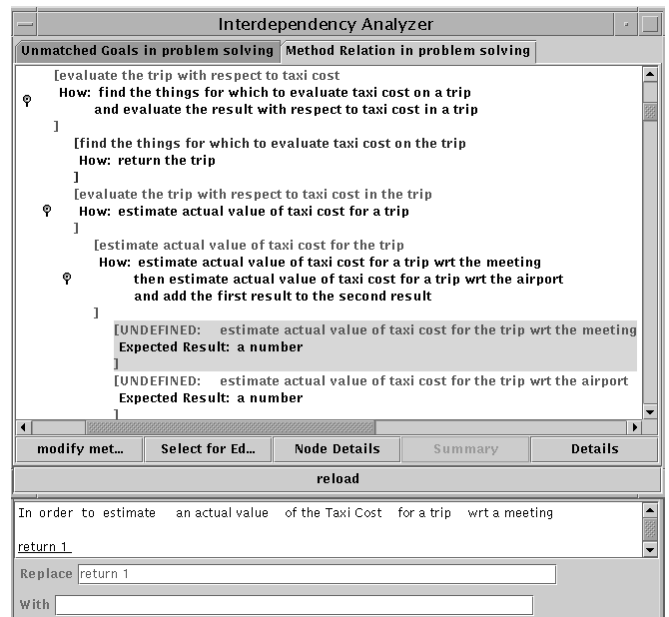


Figure 12: The Interdependency Analyzer helps the user to add problem solving knowledge by showing how the individual steps and substeps relate to each other.

The Interdependency Analyzer presents to the user the problem solving steps as shown in Figure 12. At this point, the user has provided a few problem solving pieces, but the system highlights the substeps that it does not yet know how to solve. In this case,

it indicates that it does not yet know how to estimate the taxi cost to and from the meeting. When the user clicks on that portion of the problem solving tree, the Interdependency Analyzer brings up the Method Editor, and creates an initial template for the method based on its place within the problem solving tree. In this case, the method needed is for estimating taxi cost trip from the hotel to the meeting, and it should return a number since the system knows that this result will be added to something else (in this case to the taxi cost from the airport to the hotel).

Many other kinds of help are provided through this component of the interface, described in more detail in [10]. The top levels of this problem solving tree are set up by the Acquisition Wizard based on the kind of constraint being defined. These top levels had to be set manually (and possibly out of context) in our previous implementations.

6. CONCLUSIONS

We have described an acquisition interface that integrates previously developed techniques to guide users in different aspects of knowledge acquisition. Compared with the individual techniques, it is able to make better use of the context of the user's actions, so the resulting interface provides stronger guidance to users. Some of the future functionality that we are planning to add includes more comprehensive ontology editors, example-based validation techniques, and semi-automatic tools to extract knowledge from on-line sources. Our previous experimental work has shown that end users who are not programmers can successfully use some of the individual components of this system [2, 11]. We intend to perform further experiments to investigate how well the integrated system supports end users.

7. ACKNOWLEDGMENTS

We gratefully acknowledge the support of DARPA with grant F30602-97-1-0195 as part of the DARPA High Performance Knowledge Bases program, and with grant F30602-00-2-0513 as part of the DARPA Active Templates program.

8. ADDITIONAL AUTHORS

9. REFERENCES

- [1] G. Abbreitt and M. Burstein. The kreme knowledge editing environment. *International Journal of Man-Machine Studies*, 27(2):103–126, August 1987.
- [2] J. Blythe. Extending the role-limiting approach: Supporting end users to acquire problem-solving knowledge. In *ECAI 2000 Workshop on Applications of Ontologies and Problem-Solving Methods*, 2000.
- [3] J. Blythe and S. Ramachandran. Knowledge acquisition using an english-based method editor. In *Proc. Twelfth Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, 1999.
- [4] J. H. Boose and J. M. Bradshaw. Expertise transfer and complex problems: Using aquinas as a knowledge acquisition workbench for knowledge-based systems. *International Journal of Man-Machine Studies*, 26, 1987.
- [5] A. Cypher. Bringing programming to end users. In *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.
- [6] A. Farquhar, R. Fikes, and J. Rice. The ontolingua server: A tool for collaborative ontology construction. In *Proc. Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, 1996.
- [7] B. R. Gaines and M. L. G. Shaw. Eliciting knowledge and transferring it effectively to a knowledge-based system. *IEEE Transactions on Knowledge and Data Engineering*, 5(1), 1993.
- [8] Y. Gil and E. Melz. Explicit representations of problem-solving strategies to support knowledge acquisition. In *Proc. Thirteenth National Conference on Artificial Intelligence*. AAAI Press, 1996.
- [9] P. D. Karp, V. K. Chaudhri, and S. M. Paley. A collaborative environment for authoring large knowledge bases. *Journal of Intelligent Information Systems*, 13:155–194, 1999.
- [10] J. Kim and Y. Gil. Deriving expectations to guide knowledge-base creation. In *Proc. Sixteenth National Conference on Artificial Intelligence*, pages 235–241. AAAI Press, 1999.
- [11] J. Kim and Y. Gil. Acquiring problem-solving knowledge from end users: Putting interdependency models to the test. In *Proc. Seventeenth National Conference on Artificial Intelligence*. AAAI Press, 2000.
- [12] G. Linden, S. Hanks, , and N. Lesh. Interactive assessment of user preference models: The automated travel assistant. *Sixth International Conference on User Modelling*, 1997.
- [13] D. L. McGuinness, R. Fikes, J. Rice, , and S. Wilder. The chimaera ontology environment. In *Proc. Seventeenth National Conference on Artificial Intelligence*. AAAI Press, 2000.
- [14] K. Myers. Strategic advice for hierarchical planners. In *Proceedings of the International Conference on Knowledge Representation*, 1996.
- [15] A. R. Puerta, J. W. Egar, S. Tu, and M. A. Musen. A multiple-method knowledge acquisition shell for the automatic generation of knowledge acquisition tools. *Knowledge Acquisition*, 4(2):171–196, 1992.
- [16] B. Swartout, R. Patil, K. Knight, and T. Russ. Towards distributed use of large-scale ontologies. In *Proc. Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, 1996.
- [17] W. R. Swartout and Y. Gil. Expect: Explicit representations for flexible acquisition. In *Proc. Ninth Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, 1995.
- [18] M. Tallis and Y. Gil. Designing scripts to guide users in modifying knowledge-based systems. In *Proc. Sixteenth National Conference on Artificial Intelligence*. AAAI Press, 1999.
- [19] L. G. Terveen and D. A. Wroblewski. A collaborative interface for browsing and editing large knowledge bases. In *Proc. Eighth National Conference on Artificial Intelligence*. AAAI Press, 1990.