From the Proceedings of the Ninth Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW'95) Banff, Canada, February 26-March 3, 1995

EXPECT: Explicit Representations for Flexible Acquisition

Bill Swartout
Yolanda Gil

USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

tel: (310) 822-1511
email: *swartout@isi.edu, gil@isi.edu*

Abstract: To create more powerful knowledge acquisition systems, we not only need better acquisition tools, but we need to change the architecture of the knowledge based systems we create so that their structure will provide better support for acquisition. Current acquisition tools permit users to modify factual knowledge but they provide limited support for modifying problem solving knowledge. In this paper, we argue that this limitation (and others) stem from the use of incomplete models of problem solving knowledge and inflexible specification of the interdependencies between problem solving and factual knowledge. We describe the EXPECT architecture which addresses these problems by providing an explicit representation for problem solving knowledge and intent. Using this more explicit representation, EXPECT can automatically derive the interdependencies between problem solving and factual knowledge. By deriving these interdependencies *from the structure of the knowledge-based system itself* EXPECT supports more flexible and powerful knowledge acquisition.

1. INTRODUCTION

EXPECT is a framework for knowledge based systems that we are developing to support knowledge acquisition and explanation. A central idea behind EXPECT has been the notion that more powerful acquisition and explanation tools can be constructed if acquisition and explanation concerns are reflected in the structure of the knowledge based systems we create. That is, rather than developing tools that operate on conventional knowledge based systems, we first modify the architecture of the target knowledge based systems so that they will be structured in a way that provides better support for knowledge acquisition and explanation. It is then possible to build tools that exploit this additional information to provide enhanced capabilities.

In prior work on the EES framework [Neches et al, 1985; Swartout et al., 1991] we explored the architectural modifications that are needed to support explanation. EXPECT extends the EES framework to support knowledge acquisition. In this paper, we discuss the architectural features that support knowledge acquisition. It is interesting to note that some of the features that are important for explanation also provide leverage for knowledge acquisition.

There are several knowledge acquisition capabilities that we seek to support with the EXPECT architecture:

1. *Users should be able to modify both factual knowledge and problem solving knowledge.* Although many acquisition systems provide good support for modifying factual knowledge,

support for modifying problem solving knowledge is more limited. In early acquisition systems (such as MORE [Eshelman 1988], SALT [Marcus and McDermott 1989], and ROGET [Bennett 1985]), a single problem solving strategy (e.g. heuristic classification) was built into the tool. As a result, when one selected a tool, one also determined the problem-solving strategy. However, many realistically-sized knowledge-based systems use several problem solving strategies, so a tool that only supports a single strategy won't be much help. Additionally, if the user changes his mind about which problem-solving strategy is appropriate, he may also have to change tools, potentially losing a lot of work in re-configuring the domain knowledge. Recent knowledge acquisition work [Klinker et al. 1991; Musen and Tu 1993] has partially addressed these problems by creating tools that can use multiple problem-solving methods. In PROTÉGÉ II [Musen and Tu, 1993], problem solving methods are composed of pre-encoded building blocks. PROTÉGÉ II permits modifications to problem solving by substituting one building block for another, however, users cannot modify the building blocks themselves. By constraining the user's options to just those that have been pre-encoded, he may not be able to make the modification he wants.

2. *Internal models (based on the knowledge based system being modified) should guide knowledge acquisition rather than external ones (based on the acquisition tool).* Current acquisition systems cannot allow much modification to problem solving knowledge because they use external models of problem solving that are built into the acquisition tool. By understanding what factual knowledge is needed to support problem solving the acquisition tool can form expectations about what additional knowledge is required when the user adds or modifies the system's knowledge base. In early acquisition tools, these expectations were built, by hand, into the tool itself. In more recent work, such as PROTÉGÉ II, the interdependencies between factual knowledge and problem solving building blocks are represented, but they are still entered by hand and interdependencies between factual knowledge and what happens inside a building block are not modeled.

   These limitations could be overcome and a wider range of modifications and problem solving methods could be supported if we could use an *internal* model for acquisition, that is, if the expectations for acquisition could be derived from the system itself. Furthermore, because the interdependencies would be derived from the system rather than entered by hand, they would change as the system changed, and would be more likely to be correct and consistent. Thus, we need architectures that allow us to derive the interdependencies between problem solving and factual knowledge from the knowledge structures themselves Such architectures will allow us to build knowledge acquisition tools that permit direct modification of problem solving methods and use their understanding of the program to guide the user.

3. *KA tools should support modifications at a semantic (or knowledge) level rather than just at a syntactic level.* KA tools should help ensure that the knowledge a user adds makes sense, that is, that it is coherent and consistent with the rest of the knowledge base—not just that it is syntactically correct. In addition, we want to facilitate the addition of new knowledge by reducing the distance between what the user understands and way the system represents them. The system must be able to represent and manipulate conceptual entities that are meaningful to users and that are used to describe the domain. To further facilitate acquisition, KA tool should allow users to make modifications that are local to these meaningful conceptual entities, and guide them in resolving the global implications of these changes. In addition, if the user enters knowledge using somewhat different terminology, the acquisition tools should be able to weave the new knowledge into the existing system based on the meaning of the terms. Providing assistance at the conceptual level and allowing greater flexibility in the use of terminology will free users to focus on what matters: getting the knowledge right.

We have introduced several features into the architecture of EXPECT that directly support the acquisition goals we outlined above. We have used the EXPECT framework to construct knowledge based systems in several domains. Our main application is for evaluating alternative military transportation plans for moving personnel and materiel from bases to crisis situations. The system we have been developing takes a high-level description of a possible plan, or course-of-action (COA), and produces an evaluation of the plan from a logistics perspective. The evaluation indicates, for example, how many logistic personnel will be required to execute it, and how long the plan will take (the closure date). Most of the examples in this paper are based on this domain, some are from another domain concerned with the diagnosis of faults with local area networks.

The next section presents a high level overview of the EXPECT architecture, and is followed by detailed discussions of its features. Section 3 is concerned with how we represent various kinds of knowledge in EXPECT. Section 3.1 deals with the need to separate different kinds of knowledge to make it easier to understand and modify. Section 3.2 describes how we represent the intent behind a problem solving method using a structured representation for goals and method capabilities. Section 4 then describes how we bring knowledge back together again to actually solve problems. Section 4.1 details how "loose coupling" between knowledge resources can facilitate knowledge acquisition and knowledge re-use, and describes how we achieve loose coupling through the use of semantic match and reformulation. Section 4.2 describes how we use partial evaluation to capture the interdependencies between problem solving and factual knowledge that are needed to guide knowledge acquisition.

2. EXPECT ARCHITECTURE OVERVIEW

A diagram of the overall EXPECT architecture appears in Figure 1. As we describe in the next section, EXPECT provides explicit and separate representations for different kinds of knowledge that go into a knowledge based system. EXPECT distinguishes domain facts, domain terminology and problem solving knowledge. These different sources of knowledge are integrated together by a program instantiator to produce a domain-specific knowledge based system. While the domain-specific system is being created, the program instantiator also creates a design history. The design history records the interdependencies among the different kinds of knowledge, such as what factual information is used by the problem solving methods. This information is used by the knowledge acquisition routines to form the expectations that guide the knowledge acquisition process (see [Gil, 1994; Gil and Paris, 1994] for a description of the knowledge acquisition tools). An interpreter executes the knowledge based system, and the user interacts with it through a user interaction manager.

3. REPRESENTING KNOWLEDGE IN EXPECT

In order to provide the capabilities that we discussed in the introduction, EXPECT represents different types of knowledge separately and explicitly. After describing the kinds of knowledge that are represented in EXPECT we present in detail our approach to representing goals. This explicit representation of goals provides EXPECT with a better understanding of the problem solving process.

3.1 Separation of Knowledge

It is well established that a major source of difficulties in understanding, modifying and augmenting first generation knowledge based systems stemmed from the use of low-level knowledge representations that failed to distinguish different kinds of knowledge (see [Chandrasekaran and Mittal, 1982; Clancey, 1983b; Swartout, 1981; Swartout, 1983]). In a first generation system, domain facts, problem solving knowledge, and terminological definitions were all expressed in rules. A single rule might mix together clauses concerned with the user interface, the system's problem solving strategy and internal record-keeping. Because none of these different concerns were distinguished, it was often difficult to understand exactly what the rule was supposed to do, and

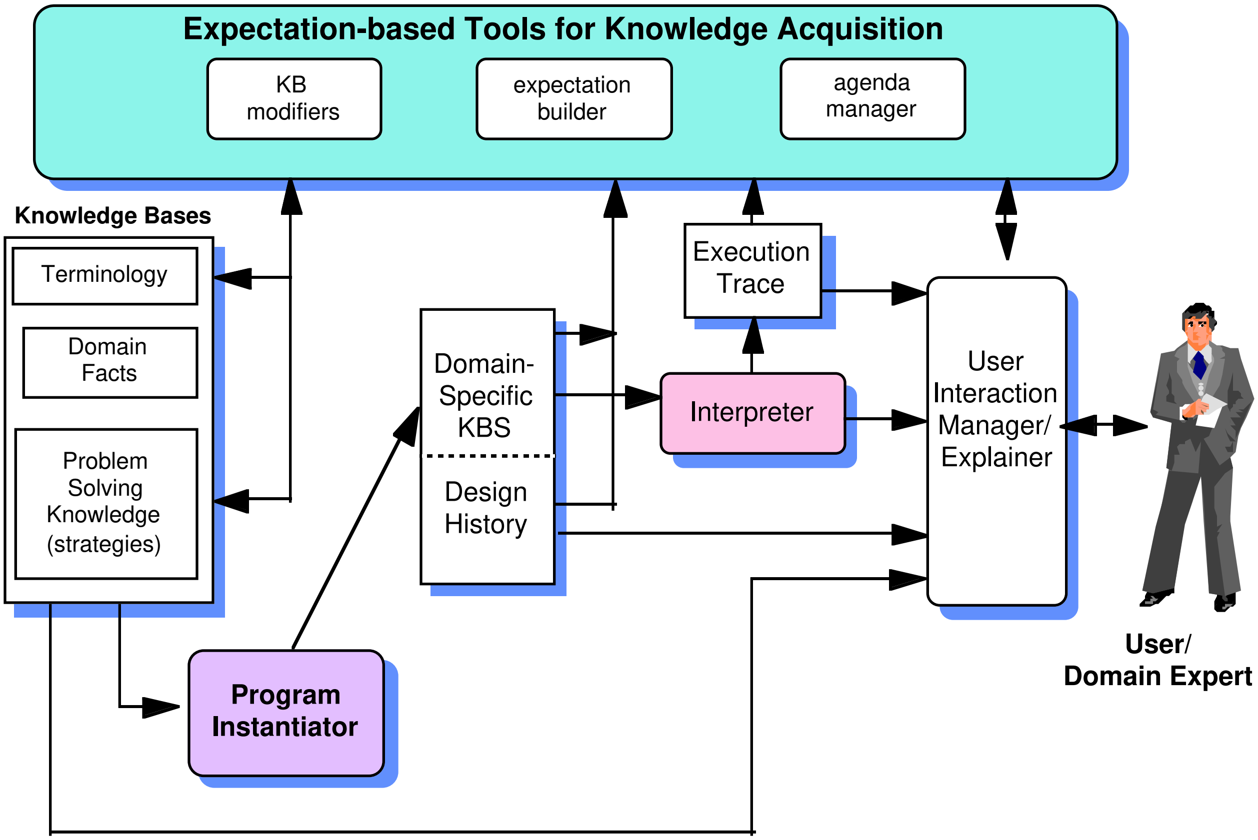

Figure 1. EXPECT Architecture

when modifying a rule, it was difficult to see what the effect of the modification might be. Although early critiques of these representations focused on their failure to provide good explanations, explanation and knowledge acquisition can be viewed as inverse processes, and these architectural flaws create problems for both.

A number of second generation expert system frameworks that support better explanations have emerged in recent years (see [Chandrasekaran, 1986; Clancey, 1983a; Hasling et al., 1984; Neches et al, 1985; Swartout, 1983; Swartout et al., 1991; Wielinga and Breuker, 1986]). A common theme among these frameworks is that they encourage a more abstract representation of domain knowledge and problem solving knowledge that makes distinctions between different kinds of knowledge explicit.

By moving toward architectures that allow a system builder to distinguish different kinds of knowledge and represent them separately and more abstractly, second generation frameworks increased the modularity of an expert system. It has been shown that the use of these higher-level representations helps make second generation systems easier to understand. These frameworks can also make it easier to extend and maintain an expert system, and enhance the possibilities for knowledge re-use across systems [Bachant and Soloway, 1989].

The XPLAIN [Swartout, 1983] and EES [Neches et al, 1985; Swartout et al., 1991] frameworks, which were precursors to EXPECT, were among the first to explicitly distinguish the different kinds of knowledge in a knowledge based system. In EXPECT, we distinguish three different kinds of knowledge: *domain facts*, *domain terminology*, and *problem solving knowledge.*

Domain facts are the relevant facts about a system's domain. For example, the domain facts in a system for transportation planning might include the fact that the naval port of Los Angeles is Long Beach and the fact that the maximum depth of Long Beach berths is 50 feet.

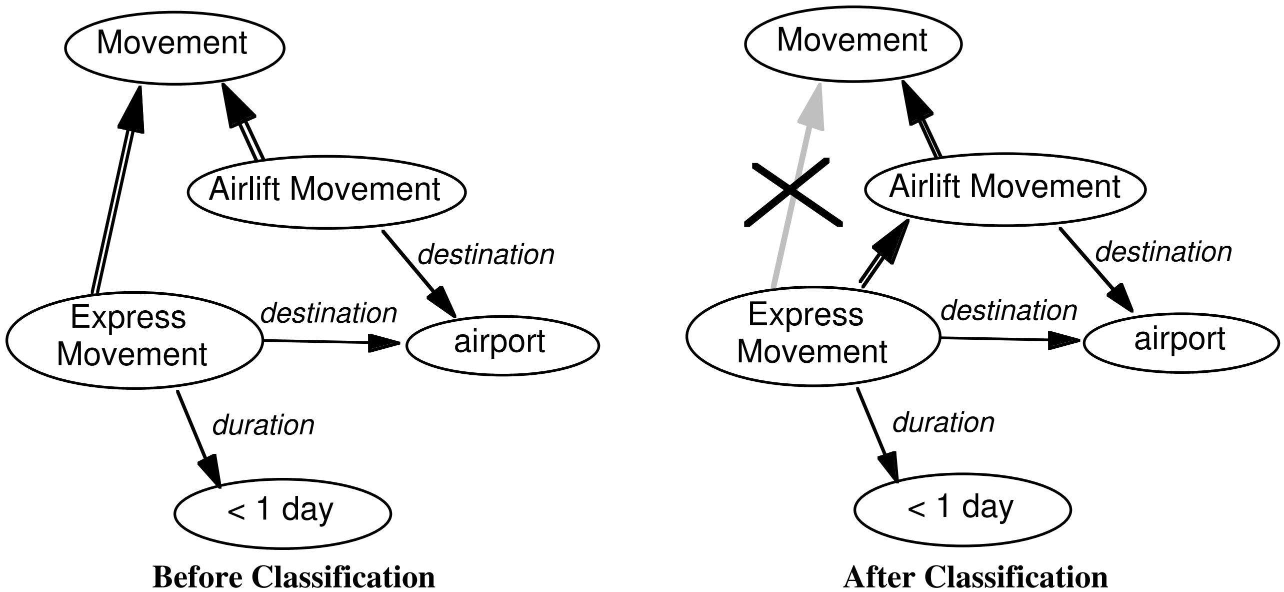

Figure 2. Loom Classifier Organizes Concept Hierarchy

The domain terminology (or *ontology*) provides the conceptual structures that are used to describe a domain. In a transportation planning domain, the domain terminology would include concepts for various kinds of ports, such as airports and seaports, and concepts for describing the various kinds of movements[1] and materiel to be moved, among other things. Concepts can be defined in terms of other concepts. For example, an airlift-movement is a kind of movement whose destination is an airport. Domain terminology and domain facts may seem to be quite similar, but they are different. Domain terminology provides a set of terms, or concepts, that can be used to describe some situation. The existence of a concept does *not* imply that the situation it describes actually exists—concepts are just descriptions that may or may not apply in any given situation. Domain facts, on the other hand, use domain terminology to describe the state of the world at some point in time.

To represent both domain facts and terminology, EXPECT uses Loom [MacGregor 1988, 1990], a knowledge representation system in the KL-ONE family. Loom provides a descriptive logic representation language, and provides a classifier for inference. Facts are represented as Loom *instances*, while terminology is represented using Loom *concepts.* Both instances and concepts are structured, frame-based representations with slots that indicate relations in which the object is involved. Instances denote objects that actually exist in the domain, while concepts are descriptions.

Like other description logic representations, Loom provides a tool, called the *classifier*, that has unique advantages for knowledge acquisition. Given a set of defined concepts, the classifier can automatically organize them into an AKO (or subsumption) hierarchy by analyzing their definitions. For example, if we defined an express-movement as a movement whose duration is less than a day and whose destination is an airport, the classifier would figure out that an express-movement is also a kind of airlift-movement, given the definition of airlift movement above (see Figure 2). The classifier helps support the acquisition of new concepts, because it ensures that the concept hierarchy is maintained consistently and it can find subsumption relations that a user might overlook. Furthermore, if a new concept is added to the knowledge base and the classifier places it in a bizarre place, that usually indicates a modeling error of some sort. For knowledge acquisition these

[1]In transportation planning, a "movement" specifies what is to be moved from an origin to a destination at a particular time using some set of vehicles.

capabilities are important, because they mean that a user can add knowledge to a knowledge base and maintain its consistency without needing to be aware of all the other concepts in the system. Concepts from the domain terminology are used as the building blocks for EXPECT's two other kinds of knowledge: domain facts and problem solving knowledge.

As Doyle and Patil [Doyle and Patil, 1991] have pointed out, the advantages of automatic classification are vitiated somewhat if primitive concepts are used. Primitive concepts lack a definition, and the knowledge base builder explicitly indicates where they should be placed in the concept hierarchy, instead of having the concept's place inferred by the classifier reasoning about its definition. Primitive concepts are used when the model builder either cannot or chooses not to provide an explicit concept definition. Doyle and Patil's critique of NIKL (an early description logic representation) stated that in realistically sized applications, most of the concepts are primitive. We have not found this to be the case in our work, perhaps because Loom provides a more expressive language for defining concepts, making it easier to define concepts. Regardless, it is important to point out that the most important capabilities of our architecture, such as capturing the interdependencies between domain and problem solving knowledge, can be provided whether or not primitive concepts are used. The use of primitive concepts reduces our framework's ability to verify the consistency of a knowledge base, and to automatically find new uses for knowledge.

Loom benefits us by providing a declarative foundation for representing concepts, relations and facts. However, to be able to analyze the interdependencies between the conceptual structure of a system and its problem solving methods we need a highly declarative representation for problem solving knowledge as well. In the next section, we describe how we have built on Loom to create a declarative representation for goals and methods that captures their intent and promotes their analysis.

3.2 Capturing Intent with Structured Goal Representation

In the introduction, two of our goals were to allow users to be able to make modifications to problem solving as well as factual knowledge, and to make modifications locally, so that a problem solving method could be modified without having to understand how other methods in the system are implemented. Problem solving knowledge in EXPECT is represented as strategies (called *methods*) for achieving particular goals. Each method has a *capability description* associated with it, which states what the method can do (e.g. "diagnose a component"), and a *method body* which is a sequence of steps that attempt to achieve the advertised capability of the method. The steps can post further subgoals for the system to achieve. The language used in the method bodies allows conditional expressions and iteration.

One of EXPECT's architectural features that helps users make modifications to problem solving knowledge is a rich representation for goals and method capabilities that provides a more explicit representation of intent than is available in most systems. This representation both makes it easier for people to understand what a method is supposed to be doing, and it makes it easier for EXPECT to analyze a system's structure and hence provide guidance to a user in making modifications.

In most programming languages, function names have no real semantics associated with them. A good programmer may indicate what a function is supposed to do by giving it an appropriate name (e.g. `clear-screen`) but as far as the system is concerned, the name is just a string of characters. Furthermore, the relationship between the function name (what is to be done) and the functional parameters (the things that will be involved in doing it) is completely implicit. This is one reason why conventional programs are so difficult to modify: before one can make a modification to a function, one needs to understand not only the function itself, but also how the function is called and, in turn, how the functions it calls work. Only then will the information that is needed for modification but not represented be available. AI systems such as planners provide a more explicit representation of what is to be done. Usually the goal is represented as some sort of state expressed in some form of predicate logic. A problem with this representation that affects both acquisition and explanation is that it seems to be a bit removed from the way people think and talk about what they are doing, since protocols of people solving problems show that they use verb clauses to describe what they are doing rather than state descriptions (see for example, [Anzai and Simon 1979]).

Thus, to address these difficulties and decrease the distance between EXPECT'S internal representation and how people talk about what they do, EXPECT's representation is based on a vocabulary of *verbs.* In EXPECT, we represent both goals (what is to be done) and method capability descriptions (what a method can do) using a verb clause representation based on case grammar. Each verb clause consists of a verb and a set of slots (or "cases") that are the parameters. For example, the goal of evaluating a particular COA is represented as a verb clause where the main verb is "evaluate" and its (direct) object slot is filled by the COA instance (e.g., "coa-3") as follows:

```
(evaluate (obj (instance coa-3)))
```

The capability descriptions associated with methods are represented similarly, except that they may contain variables. For example, the following capability description would be associated with a method that has the capability to evaluate a course of action for force deployment:

```
(evaluate (obj (?c is (instance-of deployment-coa))))
```

where:

```
deployment-coa is a coa
                  that has force-movements
```

For matching, the goals and capability descriptions are translated into corresponding Loom concepts. The goal above would match this capability description if `coa-3` had movements of forces, since `coa-3` would then be a kind of `deployment-coa`.

This approach gives us a rich representation for what is intended by a goal and what the capabilities of a method are. Unlike the state-based representation, goals and capabilities can be easily paraphrased into natural language. Furthermore, goals can be expressed at an abstract level without forcing over-commitment.

Figure 3 shows a simple method from our system for evaluating transportation plans. It finds the unloading time of a movement by calculating the unloading time of the set of available vehicles of the COA movement. There are two things to notice about this method. First, it is relatively straightforward to paraphrase the representations of the capability description and the method body into understandable natural language, because their structure[2] mirrors natural language. Second, this method illustrates the use of two general kinds of parameters that can be passed in EXPECT. They distinguish between the kind of data that will be provided to the method and the kind of task to be accomplished by the method. They are indicated by `instance-of` or `specialization-of` in the capability description. These keywords indicate how the matcher should match these slots against the corresponding slots in goals.

[2]One weakness of our current representation for goals and capabilities is that some slots that are attached directly to the main verb should really be embedded within other parameters. Figure 3 illustrates that problem, since the "of" slot in the capability description of the method should be attached to unloading-time rather than directly to the verb. We are currently developing a more structured representation to support this embedding.

```
capability: (find
              (obj (?t is (specialization-of unloading-time)))
              (of (?m is (instance-of movement))))

result-type: (inst-of time-value)
method: (calculate
             (obj ?t)
             (of (available-vehicles ?m)))
```

Figure 3. A method to calculate the unloading time of a movement

`Instance-of` indicates that the slot is a *data parameter* and will match an instance of the indicated type. Thus, `(instance-of movement)` will match a movement instance or an instance that is more specialized. (Recall that in Loom, instances are representations of objects that actually exist, whereas concepts are just descriptions.) Instance-of slots work much like function parameters in conventional programming languages: they supply the data that the function manipulates. However, in EXPECT, slots on goals do more than just provide data; they can also further specify the task to be done. *Task parameters* are indicated by `specialization-of` slots and match against *concepts* that appear in corresponding slots in the goal. For example, `(specialization-of unloading-time)` will match `unloading-time` or any of its specializations. The capability description of the method in Figure 3 will match a goal such as:

```
(find (obj (specialization-of unloading-time))
      (of (instance movement-23)))
```

or it will match the goal:

```
(find (obj (specialization-of emergency-unloading-time))
      (of (instance movement-23)))
```

Notice the “obj” slot in both cases does not *supply* data but instead it *specifies* the sort of information that is supposed to be found by the method. `Specialization-of` slots allow us to re-use the same method in several different contexts, and they are one of the ways we achieve “loose-coupling” in EXPECT, which we will discuss in detail in Section 4.2. In the example in Figure 3, `?t` is bound to the concept in the goal that matches `(specialization-of unloading-time)`. The variable `?t` is then used in the method body to pass the goal context on to subgoals so that the method body will compute for example the `emergency-unloading-time` rather than the `unloading-time` if emergency time was specified in the original goal. Like other variables `?t` can be used in conditional expressions so that methods can be written that branch depending on the nature of the task specified.

In most systems, the only parameters passed are data objects that will be used during the execution of the method. Distinguishing task parameters adds an additional dimension for method abstraction, so that methods can be abstracted not only in terms of the data that they will operate on, but also in terms of the *classes* of tasks they can address. This allows us to create a common method that abstracts both the instances it operates on and the classes of tasks it can be used for and provide a single, common skeletal representation that captures both. This helps both method re-use and knowledge acquisition by providing an additional way of abstracting methods.

The key features of our goal representation are: 1) it is structured, 2) the structure is based on a verb clause representation, and 3) in addition to data parameters, task parameters are explicitly distinguished. Our representation captures more of the intent behind a goal than conventional representations. Thus it provides more of the information that acquisition tools need to analyze the knowledge base and capture interdependencies. Our representation also structures goals and method capabilities in a way that is quite close to the way people express them in natural language, making it easier to build tools that explain methods and help users modify them.

4. BRINGING KNOWLEDGE BACK TOGETHER: LOOSE COUPLING IN EXPECT

We have argued that by separating different kinds of knowledge and providing explicit representations for them knowledge based systems created within the EXPECT framework can be easier to understand, and because the separation provides increased modularity, they are easier to augment and modify. In this section, we describe how the program instantiator works to match up and integrate different knowledge resources. We refer to the matching process we use as *semantic match* because resources are matched up based on their *meaning* as opposed to their syntax. We argue that this loosens the coupling between resources, which can have distinct advantages for knowledge re-use and acquisition.

4.1 Resolving Goals: Semantic Matching And Goal Reformulation

In many systems, matching of goals and methods is done on a fairly syntactic basis. Lexemes in goals must match those in methods, and variables are matched by position. In EXPECT, we have tried to move toward a matching process that is based on the semantics of the goals rather than their syntax, and one in which reformulation can be used to achieve a match when a more direct match is not possible. In EXPECT, the ability to provide looser coupling between goals and method capabilities depends on the way that they are represented. As we described in Section 3.2, both goals and method capabilities are represented as verb clauses. A main verb states what is to be done, using a number of slots that act as “cases” (as in case grammar). For matching, both goals and method capabilities are translated into Loom concepts that mirror their structure.

Semantic Match One of the mechanisms EXPECT provides to achieve looser coupling is based on Loom’s classifier (one of the reasoners that Loom provides). Given an existing hierarchy of Loom concepts (and their definitions) organized according to the subsumption relations (AKO) between them, the classifier is capable of figuring out where in the hierarchy a new concept belongs, based solely on the definitions of the concepts. To find possible methods for accomplishing a goal, the Loom classifier is used to find those methods whose capability descriptions subsume the goal. The classifier provides a form of semantic match, because match is based on the meaning of concepts, not on their syntactic form. Section 3.2 described how goals are represented as Loom concepts and showed some simple examples of how the semantic matcher works.

The semantic matcher finds methods whose capabilities subsume the posted goal not only when it is given the class name but also when it is given a description of the type of object that needs to be matched. An example of this kind of matching occurs in our network diagnosis domain. EXPECT’s terminological knowledge contains the definition:

```
lanbridge-23 is a COMPONENT
                    that is CONNECTED-TO 2 networks

CONNECTED-COMPONENT is a COMPONENT
                    that is CONNECTED-TO some NETWORK
```

When EXPECT is reasoning with its problem solving knowledge about how to achieve the goal:

```
(diagnose (obj (instance lanbridge-23)))
```

It will be able to match that goal with a method with the capability description:

```
(diagnose (obj (?c is (instance-of CONNECTED-COMPONENT))))
```

because the classifier can figure out that `lanbridge-23` is a kind of component which is connected to a network, based on the definitions of `lanbridge-23` and `CONNECTED-COMPONENT` above. This kind of subsumption matching allows EXPECT to reason about the semantics of a goal in terms of the *meaning* of its parameters.

Reformulations The second mechanism that EXPECT provides for looser coupling of goals and methods is reformulation. Usually if a method cannot be found to achieve a goal using semantic match, the system will attempt to reformulate the goal and then look for methods to achieve the resulting goal(s). Goal reformulation involves decomposing a goal and then assembling a new goal (or goals) by transforming pieces of the original goal based on their meaning. To be able to perform goal reformulation, one needs an explicit, decomposable representation for the goal, definitions for the terms the goal is constructed from, and domain facts to drive the reformulation process.

In EXPECT, we provide two general types of reformulations, *conjunctive* and *disjunctive.* A conjunctive reformulation involves transforming some goal into a set of goals, where *each of the goals* in the set must be performed to achieve the intent of the initial goal. Thus, a conjunctive reformulation is a form of divide-and-conquer: it splits a problem up into subpieces that together achieve the original goal. As in divide-and-conquer, the system must find a way of recombining the results of each of the subproblems back into an appropriate result for the original goal for a conjunctive reformulation to be successful. A disjunctive reformulation may also reformulate an initial goal into several goals, but at runtime, *only one of the goals* needs to be executed to achieve the intent of the original goal.

EXPECT provides three types of conjunctive reformulations: covering, individualization, and set reformulations.

A *covering* reformulation occurs when a goal can be transformed into several new goals that together "cover" the intent of the initial goal. Suppose that the following goal is posted to estimate how many support personnel are required for a COA:

```
(estimate (obj (specification-of support-personnel)
          (for (instance-of coa)))
```

Using subsumption matching, the system would try to find methods for achieving this goal. Suppose none were found, because the system had no general method for estimating the support personnel needed for a COA. Suppose, however, that the system did have methods for estimating particular types of support personnel needed for a COA. How could these methods be found? When the system failed initially to find a method, it would then try to reformulate the goal into new goals. If the domain model contained the fact:

```
support-personnel is partitioned by
     airport-support-personnel and seaport-support-personnel
```

the system could reformulate the original goal into two new goals:

```
(estimate (obj (specification-of airport-support-personnel)
          (for (instance-of coa)))
```

and

```
(estimate (obj (specification-of seaport-support-personnel)
          (for (instance-of coa)))
```

The system would then be able to find the two methods for estimating particular types of support personnel needed for a COA. For conjunctive reformulations, part of the reformulation process involves finding a function for re-combining the results of each of the reformulations to form the result for the original goal. The appropriate function for combining results is determined by the type of the goal. In this example, the system adds the estimates together.

This sort of reformulation process reduces the need for different parts of the knowledge base to match up exactly, which enhances the possibilities for knowledge re-use across systems. Also, because the system explicitly reasons through the reformulation process, more of the design can be captured to support knowledge acquisition. In this case, if the user added a new type of support personnel to the knowledge base, for example `unloading-support-personnel`, then the system would use its record of this reformulation to determine that it would be necessary to perform the reformulation over again to capture the new type of support personnel. EXPECT could detect that the user needs to provide a method for estimating the law enforcement personnel needed for a COA.

Individualization reformulations are similar to covering reformulations, except that they decompose a goal over a set of objects into a set of goals over individual objects (i.e., instances). For example, given the goal of calculating the employment personnel of the force modules in a COA:

```
(calculate (obj (specification-of employment-personnel))
           (of (force-modules coa-2)))
```

and the domain fact that:

```
force-modules of coa-2 are the instances:
      3rd-ACR 57th-IMF CVN71-ACN
```

the system could transform the original goal into three goals:

```
(calculate (obj (specification-of employment-personnel))
           (of (instance 3rd-ACR)))

(calculate (obj (specification-of employment-personnel))
           (of (instance 57th-IMF)))

(calculate (obj (specification-of employment-personnel))
           (of (instance CVN71-ACN)))
```

where each of these goals corresponds to one of the instances force modules of coa-2.

Individualization reformulations are used when the set of objects denoted by a concept is known at program instantiation time. Sometimes, however, it may be known that a set of objects will be passed to a method, but the elements of that set may not be known at program instantiation time, that is, they may only be known later, at runtime. To deal with that situation, EXPECT provides a third kind of conjunctive reformulation, the *set* reformulation. When no method is found to achieve a goal that has a set of objects in its parameters, EXPECT tries to solve the goal, at runtime, for each element of the set in turn. For example, suppose that the following goal is posted to calculate the closure date[3] for several movements:

```
(calculate (obj (specification-of closure-date))
           (of (set-of (instance-of movement))))
```

and there are no methods that operate on a set of movements. EXPECT reformulates this goal over a set into a goal over an individual movement:

```
(calculate (obj (specification-of closure-date))
           (of (instance-of movement)))
```

The matcher will return the method for calculating the closure date of a movement. At execution time, the system will loop over each of the movements in the set and calculate one by one the closure date of the movements that are included in the set.

Finally, EXPECT provides the *input* reformulation, which is a form of *disjunctive* reformulation. It occurs when no method can be found to handle one of the inputs to a goal, but several methods can be found that together will cover the range of possible inputs that will occur at runtime. For example, given the goal:

```
(calculate (obj (specification-of closure-date))
           (of (instance-of movement)))
```

suppose that the method library contained no method for calculating the closure date of a movement in general, but there were methods for calculating the closure date of particular types of movements and there was a domain fact that told the system that:

```
movement is partitioned by airlift-movement and
                            sealift-movement
```

then the system could create the goals:

```
(calculate (obj (specification-of closure-date))
           (of (instance-of airlift-movement)))

(calculate (obj (specification-of closure-date))
           (of (instance-of sealift-movement)))
```

the system would expand methods for each of these goals, and then create dispatching code that would select which method to use at runtime, based on the type of the instance of movement that was actually passed in. Note that unlike a covering reformulation, only one of the branches of a disjunctive reformulation needs to be executed.

In summary, the loose coupling that EXPECT provides through semantic match and reformulations is crucial to our approach to knowledge acquisition. First, by moving away from syntactic matching, users can add knowledge to a system without being as concerned with issues of form. This opens up

[3]The closure date is the date when all the material to be shipped has arrived at the destination.

the possibility for greater knowledge re-use and eases collaborative work on knowledge bases. The second benefit is that by having the program instantiator reason extensively about the process of matching up goals and methods, more of the design of the knowledge based system and the interdependencies between parts of the system can be captured (and hence, used to form expectations for knowledge acquisition). In reformulating goals, the program instantiator develops a rationale for how a high-level goal can be achieved in terms of lower level goals. This sort of processing is often exactly the sort of reasoning that one wants to explain and use as a basis for knowledge acquisition.

4.2 The Program Instantiator: Capturing Interdependencies

The EXPECT program instantiator works in a refinement driven fashion. Initially, the program instantiator starts with a high level goal that specifies what the expert system is supposed to do. For example, to create an expert system for evaluating a particular COA, the system would be given the following goal:

```
(evaluate (obj (instance-of deployment-coa)))
```

This high level goal determines the scope of the knowledge based system that EXPECT will create. The goal above would create a knowledge based system that could evaluate a deployment-COA—but nothing else. On the other hand, a goal such as:

```
(evaluate (obj (instance-of coa)))
```

would create a knowledge based system that could evaluate any COA that was in EXPECT's knowledge base (assuming appropriate problem solving knowledge was also available). Thus, a single EXPECT knowledge base can be used to create a variety of knowledge based systems, each scoped to cover a different (or possibly overlapping) set of problems.

References to instances that appear in goals during the program instantiation phase act as "place holders" for the actual data object that will appear when the program is executed. The use of a more general or abstract instance results in the creation of a system that can handle a wide range of inputs, i.e., all the instances of that type.

When a goal is posted, the program instantiator searches its problem solving knowledge to find a method whose capability description matches the goal. This matching of goals and methods is a critical step in the program instantiator's reasoning and was the main topic of the previous section. How it is done, and the representations that are used, have a direct effect on how maintainable and reusable the knowledge base will be and EXPECT's ability to acquire new knowledge.

Once a method is selected to achieve the posted goal, the variables in the method's capability description are bound to corresponding instances and concepts in the goal. The body of the method is expanded by plugging in the bindings for the variables in the body and then posting its subgoals. During this process, if any of the slots of an instance are accessed by the method, those accesses are recorded in the design history. This record of the interdependencies between factual knowledge (instances) and problem solving knowledge is later used to form the expectations that guide knowledge acquisition. For example, if the program instantiator notes that it uses a method that requires information about the berths of a port, then when a new port is entered, the knowledge acquisition routines will know that information about the berths of that new port needs to be added.

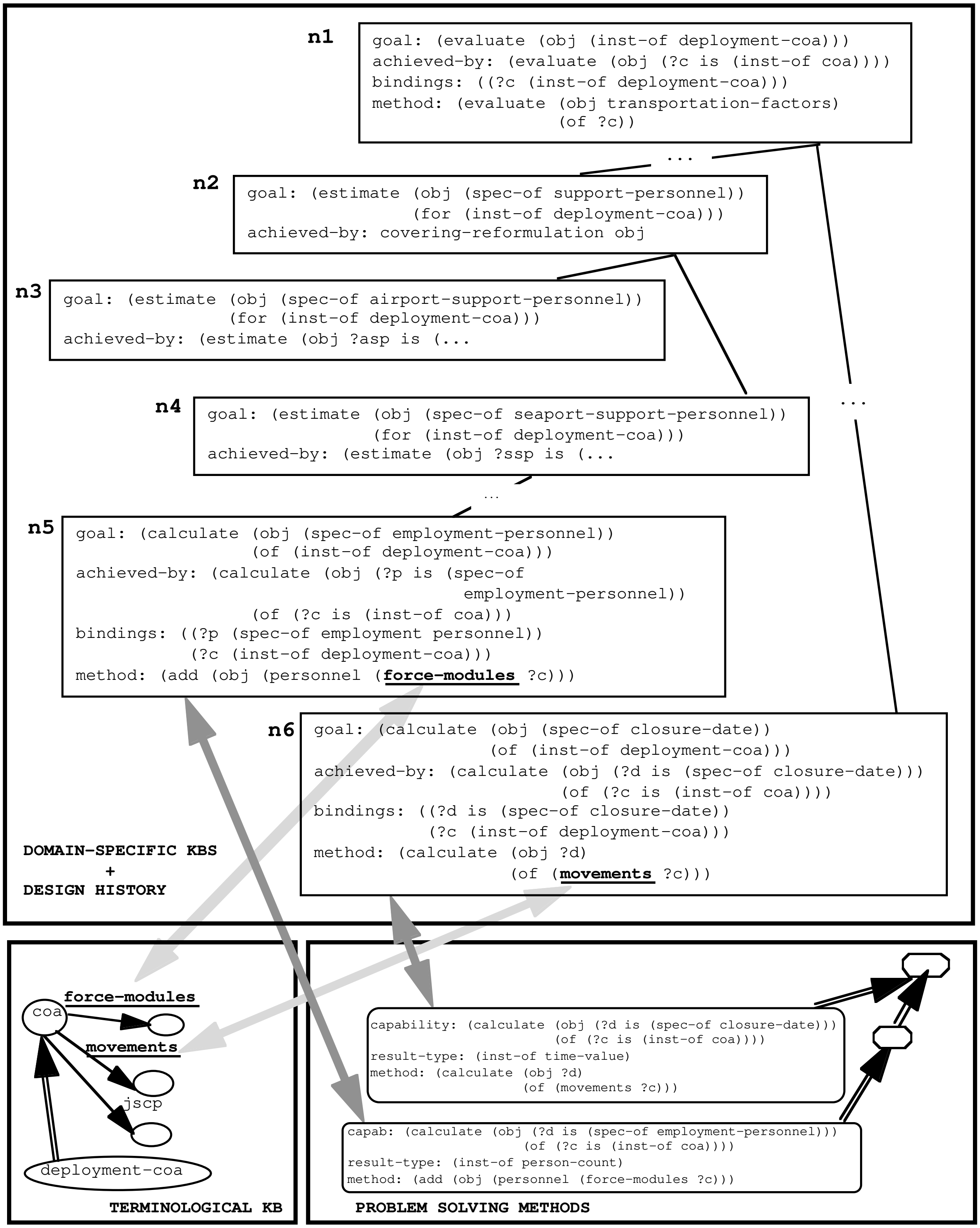

Figure 4. During problem solving, the program instantiator integrates the different types of knowledge that the diffferent knowledge bases contain and keeps track of how they are used (use of problem solving methods is in dark grey, use of aspects of domain entities is in light grey). These interdependencies, dynamically generated by EXPECT based on the current available knowledge, are the basis for guiding knowledge acquisition.

A key advantage for knowledge acquisition of EXPECT's approach is that the program instantiator explores *all* the possible execution paths through the knowledge based systems it creates. It thus captures all the interdependencies between factual and problem solving knowledge, something which is not possible to do by analyzing execution traces, for example, since each analysis will only cover one execution path through the system.

Figure 4 shows a partial view of how the program instantiator expands goals. The top-level goal given to the system is to evaluate an instance of a deployment course of action. This is the goal posted in node n1. The matcher finds a method whose capability can achieve this goal, and the method's capability, the bindings, and the method's body are recorded in the node. After the bindings are substituted in the method body, the subgoal of evaluating the transportation factors of the COA would arise, and successive goal expansions would produce the goals in nodes n2 and n6.

Notice that even though the method used to achieve the goal in n1 can be used for any kind of COA, the bindings specify that is used for deployment COAs and the system propagates this more specific type as it expands the subgoals. When no method is found to match a goal, EXPECT tries to reformulate the goal and try matching again. For example, the goal in node n2 is achieved by a covering reformulation of the object parameter of the goal.

During the process of expanding the goals, the program instantiator also keeps track of the interdependencies between the different components of the knowledge bases. Let us look more closely at node n6 in Figure 4. The goal of calculating the closure date of a deployment COA is achieved with a method whose body indicates that the system must calculate the closure date of all the movements of the COA. Since `movements` is a slot (or *role* in Loom terminology) of the concept `COA`, the system annotates that COA movements are used by the method in this node. The factual domain knowledge is effectively being linked to the problem-solving knowledge that uses it. This is shown with thick gray lines in the figure. Furthermore, the bindings in the node indicate that movements are used (and thus needed) for deployment COAs, but not for other types of COAs.

5. FLEXIBLE KNOWLEDGE ACQUISITION

By representing separately and explicitly knowledge of different types, EXPECT allows users to make changes to the knowledge bases in terms that are meaningful in the domain. By deriving the interdependencies between the different types of knowledge as they are used for problem solving, EXPECT can provide guidance for knowledge acquisition that is dynamically generated from the current content of the knowledge bases. By representing explicitly problem-solving methods, EXPECT can reason about their components and support users in modifying any component of the methods. This section describes very briefly how EXPECT's knowledge acquisition tool takes advantage of the architectural features described in this paper, see [Gil, 1994; Gil and Paris, 1994] for a more detailed description of the knowledge acquisition tools.

Using Problem-Solving Knowledge to Guide the Acquisition of Factual Domain Knowledge

EXPECT's knowledge acquisition tool supports users in entering factual domain knowledge by automatically generating a dialogue that requires the information needed for problem-solving as indicated by the interdependencies captured by the program instantiator. Let us go back to the example in Figure 4. For example, when a user wants to define an instance of a new COA EXPECT will examine the interdependencies derived by the Program Instantiator. EXPECT realizes that only deployment COAs are evaluated (according to the top-level goal in node n1), and that other kinds of COAs (such as employment COAs) are not evaluated. Based on that information, EXPECT will first request the user to be more specific about the COA. To provide maximum support to the user in providing this information, EXPECT generates a menu of options that correspond to the different kinds of COAs that are known to the system. The system continues to request the user to be more specific for as long as the subtypes of the currently specified type are used differently by the system.

Next, EXPECT examines the interdependencies to request the data needed about a deployment for problem solving. It notices that the roles used in the methods are force-modules and movements. According to the knowledge that is currently available to the system, the JSCP (Joint Strategic Capabilities Plan) information is not used for COA evaluation. (JSCP information specifies high level directives that are irrelevant for COA evaluation.) Thus, EXPECT requests the user to enter the force modules and movements of the COA and makes the JSCP information optional. Again, to support the user as much as possible, EXPECT generates a menu with the force modules and the movements that are currently defined in the system as suggestions for possible fillers.

If the user adds a step to a problem-solving method that uses the JSCP of a COA, EXPECT will automatically detect this new interdependency and ask the user to provide this information for any COAs.

Using Factual Domain Knowledge to Guide the Acquisition of Problem-Solving Knowledge

Suppose now that the domain knowledge changed and a new subclass of support personnel was added, e.g., `unloading-support-personnel`. EXPECT would then detect that to estimate the support personnel for a COA in node n2 it now generates three subgoals in the covering reformulation instead of two, and as a result it needs to have a problem-solving method for estimating the `unloading-support-personnel` needed for a COA. EXPECT guides the user in specifying the new method by reusing one of the other two as a rough initial version for the new method that the user can correct by changing any of its components.

In addition to adding new problem-solving methods, EXPECT's knowledge acquisition tool supports users in modifying existing methods. For example, a user may add a new step in a method to use the priority of a movement in a COA to decide whether or not to use more resources for transportation. Suppose that the priority of a movement is not defined, and that priorities are defined as relations that only apply to mission objectives. Based on this domain knowledge, EXPECT notifies the user that the priority of a movement is not defined and thus cannot be used by a method. EXPECT also presents to the user with a menu of all the roles that are defined for movements as options to be used in the method.

As always, the knowledge acquisition dialogue is updated if the underlying interdependencies change. In this case, if the definition of the role "priority" is changed so that it is defined for a COA then the system would allow the user to use the priority of a COA.

Discussion

EXPECT's knowledge acquisition tool can:

1) derive the interdependencies between domain and problem-solving knowledge
2) use these interdependencies to guide knowledge acquisition
3) provide suggestions to the user about how to resolve an acquisition problem based on the nature of the interdependencies
4) justify, when asked to do so, the reasons for requesting the user's intervention based on the derivation of the interdependencies
5) support users in adding new problem solving methods and in changing the internal steps of existing ones because they are explicitly represented and their relation with domain knowledge is automatically derived.

All these features are important for the flexibility of EXPECT's knowledge acquisition tool, and are possible because of its architectural features as they are described in this paper.

Other knowledge acquisition systems, such as BLIP [Morik, 1989] and KREME [Abrett and Burstein, 1987] have used description logics. However, their goals have been complementary to ours. BLIP has shown how concept definitions can be inferred from domain facts using machine learning technology, and KREME provides a variety of tools that primarily support a user in building a domain model and domain facts. Our concern, on the other hand, has been to capture how a domain model is constrained by the way it is used in solving problems, and to use those constraints to guide knowledge acquisition.

KSSn [Gaines, 1994] is very close in spirit to EXPECT. Like EXPECT, KSSn uses a description logic representation to capture domain concepts, facts and relations. In our view, a major difference between the two frameworks is the way in which the description logic inference mechanisms (such as the classifier) are used. In KSSn, the inference mechanisms are used at runtime to help derive a solution. EXPECT uses the inference mechanisms at a *meta level*. That is, the inference mechanisms are used to put together a set of program modules that will solve the problem, but the inference mechanisms are not used in the problem solving effort itself. One of the reasons we chose this approach is that we wanted the option of supporting kinds of reasoning (such as reasoning with uncertainty) that are not yet supported by Loom.

6. SUMMARY

Second generation expert system frameworks have advanced over first generation frameworks by providing ways of representing knowledge that separates different kinds of knowledge and makes explicit the distinctions among them. This has resulted in better tools for knowledge acquisition. In this paper, we have argued that additional advances in acquisition can be made by structuring problem solving knowledge so that intent is more explicitly represented and by the use of automated tools for recombining the different kinds of knowledge. We argued that the process of combining knowledge together to solve problems should be done by the machine so that the interdependencies between problem solving and factual knowledge that guide knowledge acquisition can be derived automatically. We presented a verb clause based representation for goals and method capabilities that captures the interrelationships between the action to be performed and the parameters involved. We argued that the architecture should support loose-coupling of different knowledge sources so that users adding knowledge to a system can focus on the meaning of what they are adding, rather than particular terms or syntax. Finally, we described the major mechanisms that EXPECT uses for combining knowledge together, semantic match and reformulation. These architectural features make explicit the knowledge needed to support more flexible and powerful knowledge acquisition tools.

ACKNOWLEDGMENTS

The authors would like to thank the current and former members of the EXPECT and EES projects, especially Pedro Gonzalez, Bing Leng, Vibhu Mittal, Johanna Moore, Robert Neches, Cecile Paris, Ramesh Patil, and Marcelo Tallis for their hard work and thoughtful insights that contributed greatly to this research.

The work described here has been supported by Rome Laboratory and the Advanced Research Projects Agency under contract DABT63-91-C-0025, and under NASA Ames cooperative agreement number NCC 2-520.

REFERENCES

[Abbrett and Burstein, 1987] G. Abbrett and Mark Burstein. The KREME knowledge editing environment. In International Journal of Man-Machine Studies (1987) 27, pp. 103-126.

[Anzai and Simon 1979] Y. Anzai and H. A. Simon. The Theory of Learning by Doing. In Psychological Review, 86(2) 124-140, 1979.

[Bachant and Soloway, 1989] J. Bachant and E. Soloway. The Engineering of XCON in Communications of the ACM, 32(3) 311-317, March 1989.

[Chandrasekaran and Mittal, 1982] B. Chandrasekaran and S. Mittal. Deep versus compiled knowledge approaches to diagnostic problem solving. In *Proceedings of the National Conference on Artificial Intelligence. AAAI, 1992.*

[Chandrasekaran, 1986] B. Chandrasekaran. Generic tasks in knowledge-based reasoning. *IEEE Expert* , 1(3):23-30, 1986.

[Clancey, 1983a] W.J. Clancey. The advantages of abstract control knowledge in expert system design. In *Proceedings of the National Conference on Artificial Intelligence,* pages 74-78, Washington, DC, 1983. AAAI.

[Doyle and Patil, 1991] J. Doyle and R.S. Patil. Two theses of knowledge representation: language restrictions, taxonomic classification, and the utility of representation services. *Artificial Intelligence* 48(3) 261-297.

[Clancey, 1983b] W.J. Clancey. The epistemology of a rule-based expert system: A framework for explanation. *Artificial Intelligence,* 20(3):215-251, 1983.

[Gaines 1994] Gaines, B.R. A situated classification solution of a resource allocation task represented in a visual language. *International Journal of Human-Computer Studies* 40(2) 243–271.

[Gil 1994] Gil, Y. Knowledge Refinement in a Reflective Architecture. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-94)*, 1994.

[Gil and Paris 1994] Gil, Y., and Paris, C.L. Towards Method-Independent Knowledge Acquisition. In *Knowledge Acquisition*, 6 (2), pp. 163-178, 1994.

[Hasling et al., 1984] D.W. Hasling, W.J. Clancey, and G. Rennels. Strategic explanations for a diagnostic consultation system. *International Journal of Man-Machine Studies,* 20(1), 1984.

[Neches et al., 1985] Robert Neches, William Swartout, and Johanna D. Moore. Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development. *IEEE Transactions on Software Engineering,* SE-11(11):1337-1351 November 1985.

[MacGregor 1988] MacGregor, R. A Deductive Pattern Matcher. In *Proceedings of the Seventh National Conference on Artificial Intelligence*. St. Paul, MN, August 1988.

[MacGregor 1990] MacGregor, R. The Evolving Technology of Classification-Based Knowledge Representation Systems. In John Sowa (ed.), *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, 1990.

[Morik, 1989] K. Morik. Sloppy Modeling. In *Knowledge Representation and Organization in Machine Learning.* Morik, K. (ed), Springer-Verlag, 1989.

[Musen and Tu 1993] Musen, M. A., and Tu, S. W. Problem-solving models for generation of task-specific knowledge acquisition tools. In J. Cuena (Ed.), *Knowledge-Oriented Software Design*, Elsevier, Amsterdam, 1993.

[Swartout et al., 1991] William R. Swartout, Cecile L. Paris, and Johanna D. Moore. Design for explainable expert systems. *IEEE Expert,* 6(3):58-64, June 1991

[Swartout, 1981] W. Swartout. Explaining and justifying expert consulting programs. *In Proceedings of the Seventh International Joint Conference on Artificial Intelligence.* 1981.

[Swartout, 1983] W.R. Swartout, Xplain : A system for creating and explaining expert consulting systems. *Artificial Intelligence,* 21(3):285-325, September 1983. Also available as ISI/RS-83-4.

[Swartout and Moore 1993] Swartout, W. R., and Moore, J. D. Explanation in second-generation expert systems. In J.-M. David, J.-P. Krivine, and R. Simmons (Eds.), *Second Generation Expert Systems*, Springer-Verlag, 1993.

[Wielinga and Breuker, 1986] B.J.Wielinga and J.A. Breuker, Models of Expertise, *ECAI* 1986, 497-509.