Knowledge Acquisition for Configuration Tasks : The EXPECT Approach

Surya Ramachandran Yolanda Gil
USC Information Sciences Institute
4676 Admiralty Way, Suite 1001.
Marina del Rey, California 90292
{rama, gil}@isi.edu

Abstract

Configuration systems often use large and complex knowledge bases that need to be maintained and extended over time. The explicit representation of problem-solving knowledge and factual knowledge can greatly enhance the role of a knowledge acquisition tool by deriving from the current knowledge base, the knowledge gaps that must be resolved. This paper details EXPECT's approach to knowledge acquisition in the configuration domain using the propose-and-revise strategy as an example. EXPECT supports users in a variety of KA tasks like filling knowledge roles, making modifications to the knowledge base including entering new components, classes and even adapting problem-solving strategies for new tasks. EXPECT's guidance changes as the knowledge base changes, providing a more flexible approach to knowledge acquisition. The paper also examines the possible use of EXPECT as a KA tool in the complex and real world domain of computer configuration.

Introduction

Knowledge Acquisition is an integral part of any configuration system. Changes and modifications need to be continuously made with respect to changes in markets. With the emergence of new product-lines and the discontinuation of old ones there is a need not only for good configuration systems, but also for knowledge acquisition tools that will help to keep knowledge bases current. Further, with the changes in the business needs of customers more sophisticated tools that help change configuration constraints and parameters are needed. These would be a very useful capabilities, since product knowledge changes at a high rate (40-50%/year) is reported for configuration systems such as R1 (McDermott 82) and PROSE (Wright et al. 93).

EXPECT (Swartout and Gil 95; Gil 94; Gil and Paris 94) is a flexible KA tool that has been used for a variety of tasks and domains including configuration. The problem-solving strategy is represented explicitly, and the knowledge acquisition tool reasons about it and dynamically derives the knowledge roles that must be filled out, as well as any other information needed for problem solving. Because the problem-solving strategy is explicitly represented, it can be modified, and as a result, the KA tool changes its interaction with the user to acquire knowledge for the new strategy. EXPECT provides greater flexibility in adapting problem-solving strategies because their representations can be changed as much as needed.

The paper begins by describing propose-and-revise and its use in a role-limiting tool for knowledge acquisition. Then we summarize the work described in (Gil and Melz 96) to illustrates how EXPECT's knowledge acquisition tool works when the system is using a specific problem-solving strategy. EXPECT not only supports users in filling out knowledge roles, but extends the support to acquire additional knowledge needed for problem-solving. We use the propose-and-revise paradigm in small domain for U-Haul rentals (Gennari et al. 93). We then look at a possible implementation and usefulness of a KA tool for the computer configuration domain, a domain that is considerably more complex and real world. Though not implemented, it serves as an example of the power of EXPECT as a KA tool. We describe the types of knowledge that need to be acquired for configuration tasks and show how EXPECT could support users in implementing them.

Solving Configuration Design Tasks with Propose-and-Revise

Propose-and-revise is a problem-solving strategy for configuration design tasks. A configuration problem is described as a set of input and output *parameters* (or variables), a set of *constraints*, and a set of *fixes* to resolve constraint violations. A solution consists of a value assignment to the output parameters that does not violate any constraint.

Propose-and-revise constructs a solution by iteratively extending and revising partial solutions. The *extension* phase consists of assigning values to parameters. In the *revision* phase, constraints are checked to verify whether they are violated by the current solution and if so, the solution is revised to resolve the violation. Violated constraints are resolved by applying fixes to the solution. A fix produces a revision of the solution by changing the value of one of the parameters that are causing the constraint violation.

Propose-and-revise was first defined as a problem-solving method for configuration in VT (Marcus 88) for designing elevator systems. Input parameters for VT included features of the building where the elevator was to be installed. Output parameters included the equipment selected and its layout in the hoistway. An example of a constraint is that a model 18 machine can only be used with a 15, 20, or 25 horsepower motor. An example of a fix for a violation of this constraint is to upgrade the motor if the current configuration was using one without enough horsepower.

SALT (Marcus and McDermott 89) which was used to build VT, is a knowledge acquisition tool for propose-and-revise using a role-limiting approach. In this problem-solving strategy, there are three types of knowledge roles; procedures to assign a value to a parameter which would result in a design extension, constraints that could be violated in a design extension and fixes for a constraint violation. Consequently, the user could enter one of the three types of knowledge. For each type of knowledge, a fixed menu is presented to the user to be filled out. SALT does not provide support in updating or maintaining the knowledge about elevator components.

Explicit Representations in EXPECT

In EXPECT, both factual knowledge and problem-solving knowledge are represented explicitly. This means that the system can access and reason about the representations of factual and problem-solving knowledge and about their interactions. Factual knowledge is represented in LOOM (MacGregor 91), a knowledge representation system based on description logic. Factual knowledge includes concepts, instances, and the relations among them. Problem-solving knowledge is represented in a procedural-style language that is tightly integrated with the LOOM representations. Subgoals that arise during problem solving are solved by methods. Each method description specifies: 1) the goal that the method can achieve, 2) the type of result that the method returns, and 3) the method body that contains the procedure that must be followed in order to achieve the method's goal. A method body can contain nested expressions, including subgoal expressions that need to be resolved by other methods; control expressions such as conditional statements and some forms of iteration; and relational expressions to retrieve the fillers of a relation over a concept. Some method bodies are calls to Lisp functions that are executed without further subgoaling.

We first look at an example of EXPECT's representations using propose-and-revise as a strategy for solving the following type of problems in the U-Haul domain: Given the total volume that the client needs to move, the system recommends which piece of equipment (e.g., a truck, a trailer, etc.) the client should rent. Figure 1 graphically shows parts of the factual domain model for propose-and-revise and for the U-Haul domain. [1]The upper part of the picture shows factual knowledge that is domain independent and can be reused for any domain. In the configuration process, there is an explicit representation of state variables (which denote a configuration) and constraints. The state variables can be associated with components the make up the configuration. Constraints are associated with valid sets of instantiations for the state variables. The lower part of the picture shows factual knowledge that is relevant to the U-Haul domain. Here state variables denote things like pieces of equipment and constraints contain information about capacity restrictions, etc.

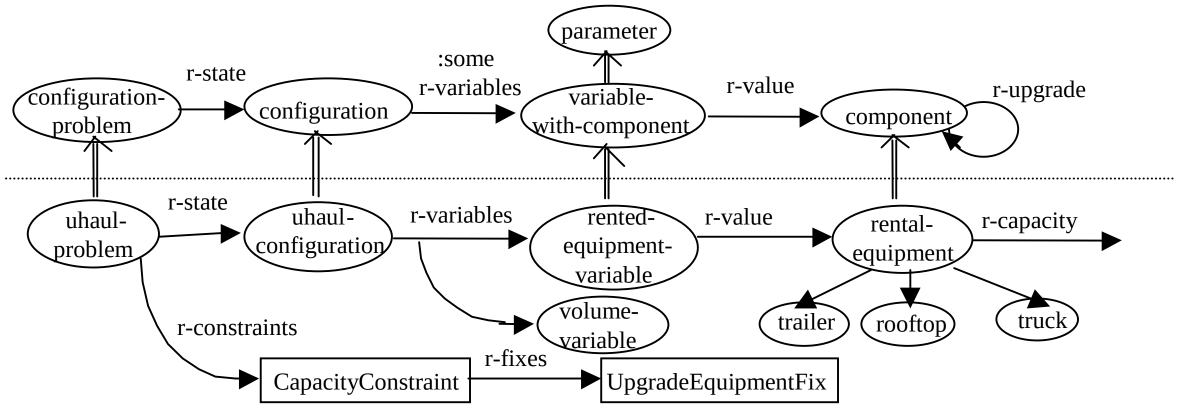


Figure1: EXPECT's representation of some of the factual knowledge needed for propose and revise problems, for configuration problems and the U-Haul domain.

There is a continuum between the representation of domain-dependent and domain-independent factual knowledge in EXPECT. They are represented in the same language, yet they can be defined and maintained separately. Once a U-Haul problem is specified as a kind of configuration problem, it inherits the fact that it has constraints and fixes. Trucks are not defined as having upgrades, since having upgrades is a way to look at components from the point of view of configuration problems. Instead, they are defined as configuration components, which have upgrades.

```
(defmethod REVISE-CS-STATE
   "To revise a CS state, apply the fixes found for the
   constraints violated in the state."
   :goal (revise (obj (?state is (inst-of cs-state))))
   :result (inst-of cs-state)
```

1 By convention, we denote relations with the prefix r-.

```
   :body (apply (obj (find (obj (set-of (spec-of fix)))
              (for (find
                      (obj
                        (set-of (spec-of violated-constraint)))
                 (in
                   ?state)))))
              (to ?state)))

(defmethod CHECK-CAPACITY-CONSTRAINT
   "To check the Capacity Constraint of a U-Haul
   configuration, check if the capacity of the rented
   equipment is smaller than the volume to move."
   :goal (check (obj CapacityConstraint)
                    (in     (?c     is
   (inst-of uhaul-configuration))))
   :result (inst-of boolean)
   :body (is-greater-or-equal
                                 (of(r-capacity
                          (r-rented-equipment ?c)))
                                 (than (r-volume-to-move ?c))))
```

Figure 2: Problem-solving knowledge in EXPECT.

Figure 2 shows two different problem-solving methods. REVISE-CS-STATE is one of the methods that specifies how propose-and-revise works. The CHECK-CAPACITY-CONSTRAINT specifies that the capacity of the equipment rented must at least be equal to the volume of the client's needs.

Knowledge Acquisition in EXPECT

EXPECT's problem-solver is designed to detect errors and to report them to the KA tool (see table 1) together with detailed information about how they were detected. The KA tool uses this information to support the user in fixing them. Other modules that can detect and report errors are the parser (which detects syntax errors and undefined terms), the method analyzer (which detects errors within a problem-solving method), and the instance analyzer (which detects missing information about instances).

EXPECT's problem-solver can analyze how the different pieces of knowledge in the knowledge-based system interact. For this analysis, it takes a generic top-level goal representing the kinds of goals that the system will be given for execution. In the U-Haul example, the top-level generic goal would be (solve (obj (inst-of uhaul-problem))), and a specific goal for execution would be (solve (obj jones-uhaul-problem)).

EXPECT analyzes how to achieve this goal with the available knowledge. EXPECT expands the given top-level goal by matching it with a method and then expanding the subgoals in the method body. This process is iterated for each of the subgoals and is recorded as a search tree. Throughout this process, EXPECT propagates the types of the arguments of the top-level goal, performing an elaborate form of partial evaluation supported by LOOM's reasoning capabilities. During this process, EXPECT derives the interdependencies between the different components of its knowledge bases. This analysis is done every time the knowledge base changes, so that EXPECT can re-derive these interdependencies.

The EXPECT's problem solver is designed to detect goals that do not match any methods, and to detect relations that try to retrieve information about a type of instance that is not defined in the knowledge base. In addition to detecting an error, each module is able to recover from the error if possible, and to report the error's type and the context in which it occurred. It would also report this error to the knowledge acquisition module, together with some context information and a pointer to the part of the problem-solving trace where the subgoal was left unsolved.

Once the errors are detected, EXPECT can help users to fix them as follows. EXPECT has an explicit representation of types of errors, together with the kinds of corrections to the knowledge base that users can make in order to solve them. This representation is based on typical error situations that we identified by hand. Table 1 shows some of the errors that can currently be detected by two of the modules: the problem solver (e1 through e3) and the instance analyzer (e5).

Knowledge Acquisition for Propose-and-Revise in EXPECT

Previously, we pointed out some of SALT's limitations in terms of its lack of flexibility as a knowledge acquisition tool. In this section, we illustrate how EXPECT's explicit representations support a more flexible approach to knowledge acquisition.

Code	Error/Potential Problem	Suggested Corrections
e1	no method found to achieve goal G in method body M	modify method body modify another method's goal add a new method modify instance, concept, relation
e2	role R undefined for type C in method M	modify method M add relation R to C
e3	expression E in method M has invalid arguments	modify method M modify another method's goal modify instance, concept, relation
e5	missing filler of role R of instance I needed in method M	add information about instance modify method body delete instance

Table1: Some of the potential problems in the knowledge bases detected by EXPECT.

Acquiring Domain-Specific Knowledge

Suppose that U-Haul decided to begin renting a new kind of truck called MightyMover. The user would add a new subclass of truck, and EXPECT would immediately request the following:

E1---I need to know the capacity of a MightyMover.

The reason for this request is that EXPECT has detected that the capacity of rental equipment is a role that is used during the course of problem solving, specifically while achieving the goal of checking the CapacityConstraint with the method shown in Figure 2.

This corresponds to errors of type e5 in Table 1. EXPECT will only request the information that is needed by the problem-solving methods.

Acquiring New Constraints and Fixes

Instead of needing the definitions of schemas to enter constraints and fixes, EXPECT requests them as constraints and fixes that are to defined by the user. Suppose for example that the user wants to add a new constraint that restricts the rental of trailers to clients with cars made after 1990 only. The user would add a new instance of constraint: TrailersForNewCarsOnly. EXPECT would analyze the implications of this change in its knowledge base and signal the following problem:

E2---I do not know how to achieve the goal
(check (obj TrailersForNewCarsOnly) (in (inst-of uhaul-configuration)))

This is because during problem solving EXPECT calls a method that tries to find the violated constraints of a configuration by checking each of the instances of constraint of U-Haul problems. This is a case of an error of type e1. Before defining this new instance of constraint, the only subgoal posted was (check (obj CapacityConstraint) (in (inst-of uhaul-configuration))) and now it also posts the subgoal (check (obj TrailersFor NewCarsOnly) (in (inst-of uhaul-configuration))). There is a method to achieve the former subgoal (shown in Figure 2), but there is no method to achieve the latter.

To resolve E2, the user chooses the third suggestion for errors of type e1 and defines the following method to check
the constraint: Once this method is defined, E2 is no longer a problem and disappears from the agenda. EXPECT's error detection mechanism also notices possible problems in the formula to check the constraint. For example, if r-year had not been defined EXPECT would signal the following problem (of type e2):

E3---I do not know what is the year of a car.

When the user defines the role r-year for the concept car this error will go away. EXPECT can also detect other types of errors in the formulas to check constraints. For example, if r-year was defined to have a string as a range, then EXPECT would detect a problem. It would notice that there is no method to check if a string ìs greater than a number, because the parameters of the method for calculating is-greater must be numbers. EXPECT would then tell the user:

E4---I do not know how to achieve the goal
(is-greater (obj (inst-of string)) (than 1990))

Like E2, E4 is an error of type e1. But in this case the user chooses a different way of resolving the error, namely to modify the definition of the relation r-year. If the user defined a fix for the new constraint, then EXPECT would follow a similar reasoning and signal the need to define a method to apply the new fix.

EXPECT changes its requests for factual information according to changes in the problem-solving methods. This can be illustrated in this example of adding a new constraint. An effect of the fact that the user defined the new method to check the constraint is that new factual knowledge about the domain is needed. In particular, EXPECT detects that it is now important to know the year of the car that the client is using (and that is part of the configuration), because it is used in this new method. The following request will be generated for any client that, like in this case Mr. Jones, needs to rent U-Haul equipment:

E5---I need to know the year of the car of Jones.

This is really requiring that the information that is input to the system is complete in the sense that configuration problems can be solved. In EXPECT, the requirements for inputs change as the knowledge base is modified.

Changing the Propose-and-Revise Strategy

Suppose that the user wants to change the revision process of propose-and-revise to introduce priorities on what constraint violations should be resolved first. The priorities will be based on which variable is associated with each constraint.

The user would need to identify which of the problem-solving methods that express propose-and-revise in EXPECT needs to be modified. The change involves adding a new step in the method to the revise state in the propose-and-revise methodology. The new step is a subgoal to select a constraint from the set of violated constraints. EXPECT would signal the following request:

E6---I do not know how to achieve the goal
(select (obj (spec-of constraint)) (from (set-of (inst-of violated-constraint))))

This is an error of type e5, and it indicates that the user has not completed the modification. The user needs to create a new method to achieve this goal. The user may also need to define a new method for the take subgoal.

With these modifications to the knowledge base, the propose-and-revise strategy that EXPECT will follow has changed. Because the representation of the new strategy is explicit, EXPECT can reason about it and detect new knowledge gaps in its knowledge base. As a result of the modification just made, there is additional factual information needed including new information about an existing knowledge role and a new kind of knowledge role. EXPECT would then signal the following requests (both of type e5):

E7---I need to know the constrained variable of TrailersForNewCarsOnly.

E8---I need to know the preference of equipment-variable.

E7 and E8 illustrate that EXPECT has noticed that the change in the problem-solving strategy requires the user to provide new kinds of information about the factual knowledge used by the strategy. This shows that in EXPECT the acquisition of problem-solving knowledge affects the acquisition of factual knowledge. Recall that E2 illustrated the converse.

Knowledge Acquisition for Computer Configuration

In this section we shall describe how the explicit representation of knowledge approach used in the EXPECT architecture can aid knowledge acquisition in the computer configuration domain. This is not an implemented domain but serves as an example to show that the approach taken in the U-Haul domain can be applied to more complex real world domains. In EXPECT the separation of different pieces of knowledge can help the user (or developer) in acquiring different forms of knowledge. In the computer configuration domain we shall look at problem solving methods (PSMs), constraints, class hierarchies and the actual instances of data that populate these classes as examples of knowledge.

For lucidity in the explanation of the benefits of a KA tool in the computer configuration domain, let us look at a possible way of representing constraints and component specifications. As the reader is aware the knowledge representation framework used by EXPECT is LOOM a description logic based KR system (detailed in the section on explicit representations in EXPECT). Frame based semantics that provide for the definition of Concepts (or frames), Instances of these concepts and Roles which provide a way to relate instances. LOOM allows for class hierarchy descriptions of components. A hierarchy of constraints can then be defined that closely relates to the class hierarchy.

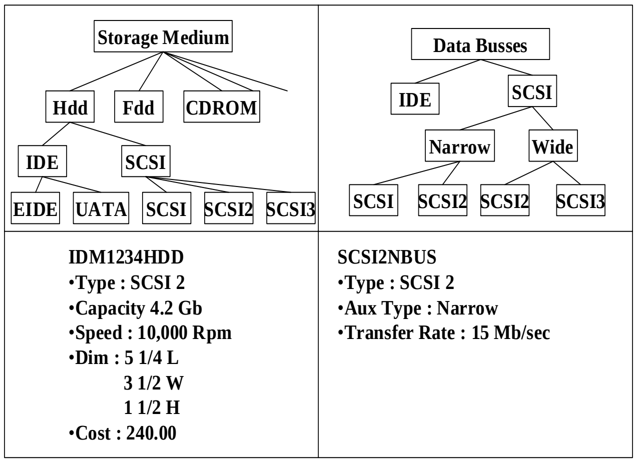
The advantages of using a description logics based system as described in the PROSE system (Wright, et al 93) which uses C-Classic (Weixelbaum 91) are applicable.

- Classification. The ability to find all descriptions applicable to an object; finding all descriptions that are more general or more specific than it (subsumption architecture).
- Completion or propagation of logical consequences. including but not limited to inheritance.
- Contradiction detection where a particular instantiation of features does not represent a legal combination.
- Dependency maintenance. A truth (or falsity) preservation over the entire set of assertions.

Figure3 depicts, on the left, a possible representation of the class hierarchy for the general class of storage mediums that can be further decomposed into *hdd* (hard disk drives), *fdd* (floppy disk drives), *CDROMs* and so on. On the right, a similar hierarchy is shown describing a common data bus architectures found in the computer domain today.

An example of what kinds of attributes an actual instance would have are also depicted. Instances here are actual components that are manufactured and that make up the final configuration.

Generally, constraints can be defined as the rules or heuristics that govern the binding of values to a set of variables for a given problem specification. A constraint limits the possible values that can be assigned to these variables. The constraint satisfaction problem (CSP) can thus be defined as a consistent set of variable assignments such that the resultant solution does not violate any of the given constraints. Configuration can be classified as a type CSP where the final solution is a list of instances (or variable instantiations) from the domain of products that do not violate any constraints (which is represented in the top half of figure 1). Constraints in the computer configuration domain can be represented either as roles (or relations) defined on the component class hierarchy at the LOOM level or as explicit problem solving methods in EXPECT.



IDM1234HDD
•Type : SCSI 2
•Capacity 4.2 Gb
•Speed : 10,000 Rpm
•Dim : 5 1/4 L
3 1/2 W
1 1/2 H
•Cost : 240.00

SCSI2NBUS
•Type : SCSI 2
•Aux Type : Narrow
•Transfer Rate : 15 Mb/sec

Figure3 : Hierarchical representation of classes and instances.

Let us now look at some examples of constraints. A typical constraint on the class of storage mediums may try to assign a data bus to connect the device to. This constraint may in turn be composed of two disjunctive sub-constraints. One trying to assign an existing data bus checking for bandwidth availability. Failing which (OR), the assignment of some component to be included in the configuration that will provide the correct bus type and enough bandwidth. But the focus of this section is not on the formulation of constraints or the actual configuration process but rather on enumerating the clear benefits of using an architecture like that of EXPECT for the ease of knowledge acquisition. We shall now describe the benefits that a KA tool provides at various levels of the hierarchy and aspects of the configuration process. We have identified four main levels that a KA tool can provide support and aid the user/developer.

The addition of a new component.

By representing explicitly the information used in the constraints and problem solving methods, the KA tool can identify which attributes of a class are needed in the configuration task. Thus the KA tool can be used to guide the users (or data entry operators) to enter only relevant information for configuration. The benefits here are that not only do data entry operators spend less time entering information on new components but the automation of this process will lead to the configuration system co-existing with other applications in the enterprise by sharing information from different databases. An example of the extraction of component information from databases may include; a database that maintains product sheets for user information, a database that has stock and warehouse information and another that has current pricing information. This is similar to E1 for knowledge acquisition in the U-Haul domain.

The addition of a new component class.

By exploiting the inter-dependencies between the various constraints and classes (i.e. the attributes used in a constraint must be defined in at least one class in the constraint hierarchy) the KA tool can suggest the possible list of attributes that would be needed for that class by comparing the list of attributes that are mentioned in the constraints and the inherited attributes at that level of the component hierarchy. The KA tools can further help in defining the value, range, and domain if such information is exploited in the constraints. Thus the configuration system can be modified with relative ease and without the domain knowledge of either the component manufacturer or the component hierarchy knowledge. This resembles E1 for KA in the U-Haul domain.

The addition of new constraints.

By identifying the level of applicability of a constraint with respect to it's position in the constraint hierarchy and the class of components it affects, the KA tool can guide the user in effectively using all parts of the KB that are available at that level (often times with complex class hierarchies and inheritance patterns, the user may not be completely aware of what attributes are available to him while writing a constraint). Another area where a KA tool can help the user write constraints is by looking for similar constraints that may already exist, and the modification of which would not only lead to greater uniformity and less mistakes in the knowledge bases but would also be a conservation of time and effort. This is similar to E2 through E5 for KA in the U-Haul domain.

The modification/extension of PSMs.

By explicitly representing the problem solving methods used by the configuration system and having access to a library of PSMs, the KA tool can help the user make the appropriate change to the PSMs for a new application in a domain requiring different inferential capabilities. For example, the choice of which part of a disjunctive constraint to evaluate first may be based on some criteria like the amount of computation needed to explicitly make a choice at to which branch to explore first. The primary benefit of having a KA tool actually guiding the user throughout this process is the fact that its usage will lead to a more robust and efficient configuration system. This is similar to E6 through E8 for KA in the U-Haul domain.

Discussion and Conclusion

Throughout the paper, we have referred to a generic user wanting to make changes to the knowledge base. This is not necessarily one user, and not necessarily the end user or domain expert. For example, the end user may only enter knowledge about clients and new parts. A more technical user would be able to modify propose-and-revise. A domain expert who does not want to change the problem-solving methods can still use EXPECT to fill up knowledge roles and populate the domain-dependent factual knowledge base. Supporting a range of users would require adding a mechanism that associates with each type of user the kinds of changes that they can make to the knowledge base and limiting the users to make only those changes. The important point is that all the changes, no matter who ends up making them, are supported by the same core knowledge acquisition tool. EXPECT's explicit representations of problem-solving strategies can be used to support flexible approaches to knowledge acquisition. Our goal is to apply this approach to support users to maintain and extend configuration systems.

Acknowledgments

The authors are grateful to all the members of the EXPECT group, past and present. Especially to Eric Melz for his work on the U-Haul Domain. We are grateful for the support of DARPA with contract DABT63-95-C-0059 as part of the DARPA/Rome Laboratory Planning Initiative, and with grant F30602-97-1-0195 as part of the DARPA High Performance Knowledge Bases Program.

References

Gennari, J. H., Tu, S. W., Rothenfluh, T. E., and Musen, M. A. Mapping methods in support of reuse. In Proc. of the 8th Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, 1994.

Gil, Y. Knowledge refinement in a reflective architecture. In Proc. of the 12th National Conference on AI, Seattle, WA, 1994.

Gil, Y., and Paris, C. Towards method-independent knowledge acquisition. Knowledge Acquisition, 6(2):163--178, 1994.

Gil, Y. and Melz, E. Explicit Representations of Problem-Solving Strategies to Support Knowledge Acquisition. In Proc. of 13th National Conference on AI, AAAI 96. 469-476, 1996.

McDermott, J. R1: A rule-based configurer of computer systems. Artificial Intelligence 19:39--88, 1982.

MacGregor, R. The evolving technology of classification-based knowledge representation systems. In J. Sowa, editor, Principles of Semantic Networks: Explorations in the Representation of Knowledge. Morgan Kaufmann, San Mateo, CA, 1991.

Marcus, S., and McDermott, J. SALT: A knowledge acquisition language for propose-and-revise systems. Artificial Intelligence, 39(1):1--37, May 1989.

Marcus, S., Stout, J., and McDermott, J. VT: An expert elevator designer that uses knowledge-based backtracking. AI Magazine 9(1):95--112, 1988.

Swartout, W. R., and Gil, Y. EXPECT: Explicit Representations for Flexible Acquisition. In Proc. of the 9th Knowledge

Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, 1995.

Weixelbaum, E. C-CLASSIC Reference Manual, Release 1.0, Technical Memorandum 59620-910731-07M, AT&T Bell Laboratories, Murray Hill, NJ. 1991.

Wright, J. R., Thompson, E. S., Vesonder, G. T., Brown, K. E., Palmer, S. R., Berman, J., and Moore, H. A knowledge-based configurator that supports sales, engineering, and manufacturing at AT&T Network Systems. AI Magazine 14(3):69--80, 1993.