On Analyzing Planning Applications

Yolanda Gil
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
gil@isi.edu

Marc Linster
Workplace Integration Technologies
Digital Equipment Corporation
200 Forest Road
Marlboro, MA 01752
linster@guess.enet.dec.com

Abstract

It is hard to evaluate in current planning applications what aspects of the approach address each of the complexities of the problem. This results from the fact that the planning community is lacking a vocabulary to describe planning tasks and applications. This work is an effort towards descriptions of planning applications in terms that are useful 1) to extract conclusions from particular implementations, 2) to facilitate cross-comparisons among different planners applied to the same problem, and 3) to facilitate comparisons among different tasks. We analyze the Sisyphus experience, a 3-year old and still ongoing effort in the knowledge acquisition community to enable a cross-comparison of their application systems as they implement a common pre-stated problem description. Based on this experience, we propose a set of dimensions to describe applications that distinghish between descriptions of the properties of the architecture, the type of problem, and the data sets. We show how they can be used to produce useful distictions in the context of the first Sisyphus task, which was an office assignment problem. Our hope is that the same dimensions will be useful to other researchers in describing and characterizing their applications, as well as a useful point of comparison for future Sisyphus efforts.

Introduction

Planning applications should be an invaluable experimental source of challenges for planning research. The real world always stretches the limitations of our planning systems. These limitations point towards new research themes in all areas: knowledge representation, computational constraints, adaptability, instructability, etc. Although the planning community has developed formalisms to describe planning architectures, we are still lacking a formal language to describe applications and problems that provides an understanding of implementations. We argue that this is a major deficiency that stops feedback from applications back to research and that keeps us away from cross-comparisons of planning systems in the same task. As Figure 1 illustrates, the world is a complex place and in the process of implementing an application, the designer continually makes choices based on (1) the baseline architecture used to implement the application, (2) the characteristics of the problem itself, or (3) arbitrary decisions to simplify the problem. All these decisions are intertwined in the resulting application, and as a result, it is not easy to abstract a description of it in terms of architectural limitations. At the same time, we would like to base our science on real-world applications that are subject to analyses that allow making predictions and to controllable experiments whose parameters can be modified to obtain experimental results of our programs' behavior (Hanks, Pollack, & Cohen 1993). But our applications rarely facilitate this task. Abstracting the design issues that lie behind the behavior of our programs is a hard thing to do, yet it is crucial that we aim towards this goal. At the core of this matter is how to achieve in practice Newell's distinction between descriptions of systems at the knowledge level, abstracting from implementation concerns and focusing on reasoning-related issues, and descriptions of systems at the level of the symbols they use (Newell 1982).

The expert systems community is an experienced one when confronted with real-world applications. There were efforts from the beginning to understand the applications of knowledge-based systems and abstract from the details the techniques used to address conceptual issues. (Stefik *et al.* 1982) describes alternative organizations of expert system architectures. Each organization is appropriate for a class of problems. Different classes of problems require different organizations, depending on the complexity of the task and the type of knowledge required. For example, if the data and knowledge are reliable, the data are static, and the search space is large but factorable, then generate-and-test is an appropriate method. Although many applications may be complex and require a combination of these methods, this analysis makes useful distinctions between the techniques that are necessary (and sufficient) to address the complexity of classes of problems.

Figure 1: Applications are shaped by the architecture used, the characteristics of problems they automate, and assumptions about the problem made by the designers.

The knowledge acquisition community, continuing this quest for automating expertise into real-world applications has taken an alternative and perhaps more empirical approach towards the same goal of understanding application systems. In 1990, during one of their meetings (Wielinga *et al.* 1990), they decided to publish several problem statements and ask research groups from around the world to use their tools, methods, and approaches to provide solutions for the problems. The results would be presented at an annual conference baptised Sisyphus. Their goal was to use the different solutions to a same problem as a basis for developing a common vocabulary for the description and differentiation of knowledge acquisition approaches. Various previous camparisons (Karbach & Voß 1991) had pointed out that the vocabulary used by different research groups varied in terms and in meaning. Using a common problem statement would hopefully help clarify issues such as: what is the difference between what different systems call actions, schemas, knowledge sources, inference actions. Intuitively, there are bound to be commonalities between their definitions, but each approach describes its building blocks at a different grain size and using varying terminology. Additionally, relating systems implemented in different languages and using different algorithms forced the KA community to use knowledge-level abstractions. Sisyphus allowed the KA community to develop a better understanding of the different tools, approaches, and representation schemes. We draw on their effort and propose a set of dimensions that can be used to describe planning applications in a way that facilitates identifying the architectural issues behind the implementation.

First, we will give a brief review of the Sisyphus experiment, the problem statement, and the solutions submitted by different groups in the knowledge acquisition community. We will describe a set of dimensions that were used in (Linster 1993b) to evaluate the first two rounds of the Sisyphus experiment and some of the conclusions that we drew using those dimensions. Then, we enumerate a set of distinctions that we deem useful to compare applications of planning architectures to the same problem.

Drawing from the Sisyphus Experience

The goal of Sisyphus is to compare different methodological approaches to the engineering of knowledge-based systems, and to provide a set of meaningful data-points to enable further analysis. Each year, a Sisyphus conference is held organized around a problems that should each be solved by the different research teams. Each participating team develops a solution using its own methodology or tool. The solutions are presented to foster comparison, discussion, and to serve as data-points to relate different approaches. In 1991, two problems were proposed: one was concerned with the analysis of textual material related to the rigging of sailboats, and the other was an assignment problem of people to offices. In 1992, a slightly modified version of the latter was used. In 1993, a complex elevator configuration task was chosen (Sandra Marcus 1988; Marcus & McDermott 1989; Yost 1993). We focus our discussion on the office assignment problem, because 1) it is the most manageable in size, 2) it is the one that has been implemented by most groups to date, and 3) it has already produced some comparative studies. The problem is described briefly in the Appendix. We now summarize the dimensions for comparing systems in one of the studies mentioned in which we based the discussion presented in the rest of the paper.

Dimensions for comparison

Linster (Linster 1993b) compares several Sisyphus solutions along different dimensions to address the following questions:

1. What are the building blocks of the model?
2. Which components are generic, which are non-reusable?
3. What is the purpose of the tool/methodology (e.g., code generation, visualization, conceptualization, elicitation, knowledge management)?
4. Where in the development cycle is the tool or methodology most useful?
5. Who is the user of the tool or methodology?

These dimensions reflect an engineering point of view on knowledge modeling, instead of a cognition-oriented one. For example the analysis does not consider the adequacy of the models nor the efficiency of the approaches in capturing the *right kind* of knowledge (see (Burton *et al.* 1990) for such a study). We do not look at the methodological aspects of the approaches either, that is, we do not study how much guidance they provide for the practitioner developing a model (see (Allemang & Rothenfluh 1992) for such a study).

What are the building blocks used to model knowledge? Building blocks were defined as discrete and identifiable constructs that the framework provides to describe the knowledge that goes into an application. Note that according to this definition interpreters or other predefined and invariable elements of a system are not considered building blocks. Rules, objects, knowledge sources, agents, classes, are examples of building blocks. The focus was on declarative elements of the knowledge representation. This appears to be the only clear boundary between general-purpose programming languages, such as Lisp or HyperTalk and other means of representation that are more commonly referred to as knowledge representation. We will distinguish three kinds of representational primitives: (1) domain-knowledge representation facilities; (2) method elements, that is the elementary generic problem-solving building blocks, their aggregation principles and the control description that combines building blocks into problem-solving methods; and (3) primitives to connect method definitions with domain knowledge.

What is the purpose of the tool/methodology? Four categories of usage were employed to discuss the different approaches: *knowledge conceptualization*, *visualization*, *implementation*, and *knowledge management*. A tool helps in the conceptualization phase if it allows to represent observations and interpretations of observations previous to their formalisation, and if it supports the user in the transition from informal to formal knowledge. We refer to a tool as a knowledge visualisation tool if its interface emphasizes graphical communication of knowledge. For a tool to be categorized as an implementation tool it must provide directly operational formalisms or strong methodological support to transform pre-operational knowledge into an operational representation. Simply attaching an editor for Lisp code doesn't do it. Moreover, delivering running systems must be the intention of the tool developers, as opposed to tools built to deliver executable specifications or feasability studies. Knowledge management refers to repository, dictionary, and browsing capabilities for the tool's knowledge representation primitives. An environment provides active knowledge acquisition support if it derives guidance for the ongoing acquisition from the current contents of its knowledge base.

Where in the development cycle is the tool most useful? The study used a description of system development as a cyclic process consisting of a set of distinct activities. These activities are organized in a logical sequence, not to be confused with a waterfall-oriented series of phases; we see them as being the central activities of a spiral development cycle, so that all phases can be repeated and previous results can be refined and even undone if indicated by the ongoing elaboration process. A more detailed description of this view on the development cycle is given in (Linster 1993a).

1. **Initial knowledge acquisition.** The knowledge engineer goes through initial interviews with the domain expert, records first protocols, and if possible she uses techniques such as the knowledge acquisition grid (LaFrance 1987) to obtain initial structures and to get a first overview of the application task.
2. **Data interpretation and knowledge structuring.** The knowledge engineer identifies recurring and potentially more abstract structures in the domain. These structures can be of different kinds.
   (a) **Identification of domain structures.** The knowledge engineer develops a structured terminology for the domain, for example she can define a T-box in KL-ONE–like approaches (Brachman & Schmolze 1985) or a set of classes in an object-oriented approach.
   (b) **Identification of inferences and roles that knowledge elements play in the problem-solving process.** The knowledge engineer defines the actions, goals, and decision criteria to give an abstract (possibly knowledge level) description of the problem-solver.
   (c) **Identification of other structures,** such as, task sharing, task decomposition, data flow, or modality.
   (d) **Integration and mapping of the different structures into a coherent model of the task.** The different knowledge structures, identified in the previous phases, must be merged into a coherent model. This phase is most important, as it represents the creative interaction between the different points of view represented by the different knowledge structures.
3. **Acquisition of the detailed knowledge in the framework defined by the task model.** The model of the task provides structures that are now stuffed with detailed knowledge about the application.
4. **Knowledge implementation.** The task model is transformed into an operational system[1].
5. **Testing and debugging.**
6. **Knowledge maintenance after system delivery.** We distinguish two types of knowledge maintenance:
   (a) **Maintenance of the structures that constitute the framework of the task model.** If structures are modified, then this changes the task model. Such modifications require re-engineering

[1] Obviously this phase is obsolete if the task model is defined using operational primitives.

of the task model, and if necessary, restructuring of the detailed knowledge.

(b) **Maintenance of the detailed knowledge in the structures of the task model.** Within the framework of the task model, maintenance of the detailed knowledge is similar to the acquisition of the detailed knowledge.

This distinction depends strongly on the implementation phase. It can only be drawn if the implementation maintains the distinction between structures of the task model and the detailed knowledge.

Who is the user? The study differentiated between four classes of users: (1) domain experts, that is, users with a lot of knowledge of the area that the tool will be used in, but without systems analysis skills and little or no programming knowledge; (2) knowledge engineers, that is, people with good systems analysis and programming skills; (3) analysts, such as workplace analysts, who do not necessarily have programming knowledge; and (4) teams consisting of domain experts working under the guidance of knowledge engineers or analysts.

Some conclusions

Many different approaches were used to solve the Sisyphus tasks, including configuration design, situated classification, constraint satisfaction, and genetic algorithms (Linster 1994). After categorizing the Sisyphus contributions in this framework, we drew conclusions such as the following ones:

> *If one wants to support the creative aspects of model building in the early phases of knowledge acquisition, then having independent languages for method and domain modeling appears to be crucial.*
>
> *If one wants support for initial knowledge acquisition and bottom-up structure development, then one is bound to get little help for the acquisition and maintenance of the detailed knowledge.*
>
> *If one wants to have good support for the implementation phase, the acquisition and maintenance of the detailed knowledge, then one should use a shell.*

Given the wide variation in the approaches taken by the different participants, it is clear that any of the approaches could successfully implement the Sisyphus problem statements. The question is not whether we can implement an application using a certain kind of approach, since this seems to have a clearly positive answer. Rather, the question is whether our implementations can support the additional functionalities that real applications demand. Can these systems be corrected if the designers had any misconceptions or misunderstandings about the task they were automating? Can these systems support additional functionalities that applications may require, such as explanation, adaptation, and scaling up? We believe that the next step is to analyze each contribution along the lines by which they would be evaluated to determine their success as office assignment tools. To do so, we propose a set of dimensions that can make useful distinctions regarding this kind of analysis. Building on these observations drawn from the Sisyphus experience, we hope that they will be useful to analyze planning domains and planning applications.

Comparing Architectures, Applications and Problems

In order to produce useful descriptions of planning applications, it is important to make distinctions between the planning architecture, the problem that the application is trying to address, and the application itself.

A *planner* is any system that implements a planning algorithm. SNLP (McAllister & Rosenblitt 1991) is an example of a planner. An *architecture* is a domain-independent system that combines different aspects of reasoning. SOAR (Laird, Rosenbloom, & Newell 1986) is an example of an architecture. For planning applications, we are interested in architectures that have some kind of planning capability.

A *problem* states the constraints posed by a set of situations in a domain and the requirements of desired solutions. Typically a solution is found for particular instances of a problem, which we will call *data sets*. Examples of problems are Blocks World and the STRIPS robot problem (Fikes & Nilsson 1971). A *task* is a description that encompasses a class of problems. For example, robot navigation tasks.

An *application* is the system that results from applying a problem-solving architecture to a problem. Notice that the same architecture applied to the same problem may result in a different application if the architecture offers different alternatives for modeling the same problem.

The dimensions presented in the previous section are useful to define which aspects of a problem an application addresses. However, they are more focused on issues that are directly relevant to work on knowledge acquisition, such as the type of user and tool support during the development of the application. Planning applications should address these issues to some degree, since these are issues that arise with any real application. But in order to understand the planning architecture itself, we need to abstract additional things from the applications that are more directly relevant to planning research. The following subsections present some dimensions that we believe are useful to describe planning architectures, problems, and applications. Our aim is not to be comprehensive, but rather to present what we believe are useful factors to analyze applications.

Describing an Architecture

1. Scope of Architecture

- task coverage — types of tasks that the architecture can be used for
- capabilities — types of reasoning that the system is able to do: uncertainty, default reasoning, learning, etc.

2. Representations
   - state descriptions — what formalism is used to represent static descriptions of objects? is there a representation of time?
   - procedural knowledge — can the system represent deduction rules? can action changes be represented?
   - decisions — can preferences be expressed explicitely? can constraints be represented?
3. Finding solutions
   - search strategy — how does the system find solutions?
   - search control — are there domain-independent search-control mechanisms? how is domain-dependent search-control knowledge formulated?
   - abstraction — is there an abstraction mechanism in the system?
4. Building blocks used to construe the problem and phrase the solution
   - generic building blocks
   - grain size of building blocks
   - knowledge-level vs. symbol-level building blocks
   - formal vs. operational vs. informal
   - process vs. structure oriented building blocks
5. Activities supported in the development cycle of an application
   - elicitation
   - knowledge structuring
   - acquisition of detailed knowledge
   - knowledge implementation
   - testing
   - maintenance
6. Adaptability — can the system evolve as its environment requires?
   - execution — can the system handle unexpected events during plan execution?
   - learning — does the system adapt autonomously, or does it require manual adaptation?
   - factual knowledge — how can objects and facts be added/deleted/updated? can the same factual knowledge be used for a different task?
   - task knowledge — can the definition of the task be changed? can the goal be changed and in which ways?
7. Prototyping
   - fast prototyping — how much effort is required to produce a working prototype?
   - prototype complexity — how much complexity of representation is required for simple problems?
8. User-system communication
   - User intervention — Does the system share the task with the user? This is a dimension spawned by the following extreme points: (1) the user keys in an instance of the problem statement; the system comes back with a ready-made solution; and (2) the user and the system cooperate in solving problems.
   - inspection — can the user understand how the system is solving a problem?
   - explanation — can the system justify its behavior?

Describing a Problem

1. Entities
   - types of objects
   - properties of individual objects
   - properties of classes of objects
   - properties that relate objects
2. Actions and Change
   - temporal considerations — reasoning about events and durations
   - state transformations — transitions and action models
3. Restrictions — can be preferences or hard-set restrictions
   - restrictions on objects
   - restrictions on actions
4. Solutions
   - are there solutions to all the instances of the problem?
   - criteria satisfaction — does the problem require satisficing solutions or are partially satisficing solutions sufficient?
   - solution quality — is there a notion of better quality solutions? can the user specify a range for the quality of the resulting solution?

A problem does not necessarily need to be modelled in these terms. Our intention in enumerating the above set is rather to provide a list of issues that need to be addressed in modelling the problem.

Data sets instantiate the problem statement in all the respects listed above. Data sets may consist of a description of all the objects of each type and their properties.

Describing an Application System

1. Representation — the knowledge necessary for solving problems
   - object types — how many different types of objects?
   - constraints — are there restrictions in the way the objects interact?
   - relations — are there relations among objects?
   - actions — what changes and transitions are possible?
   - time — does the system do some type of temporal reasoning?

2. Search Space — is the system exploring a search space in finding a solution?
   - branching factor — what is the branching factor?
   - search depth — what is the depth of the search tree?
   - goal interactions — how can the interactions between goals be characterized? (independent, serializable vs non-serializable, positive vs negative)
   - backtracking — does the system ever need to backtrack?
   - solution density — what is the average solution density when comparing the size of the search tree to the number of solutions?
3. Efficiency — how does the system address the combinatorial complexity of the problem?
   - quantitative measures — empirical results on the system efficiency
   - qualitative measures — a description of the mechanisms used to make the system efficient
4. Delivery — what happens when the system is put to use?
   - users — who are the users of the application? does the system achieve a useful task according to users?
   - environment — how does it fit in the overall work environment?

The description of the application can also include a more detailed account of the dimensions for describing an architecture.

The Sisyphus Problem Revisited

The Sisyphus problem is an office assignment task. The task consists on assigning a set of people into a set of offices satisfying some given restrictions. An office assignment task is a kind of assignment task.

As far as the particular office assignment problem posed, it can be described as follows:

- types of objects: rooms, people, projects, job types (group head, project head, manager, secretary, project member)
- properties of individual objects: person x is a smoker or a non-smoker, person x is a hacker or a staff member, room x is double or single size
- properties that relate objects: person x has job of type y, person x works with person y, person x works with person y.
- restrictions
  1. single rooms accomodate one person, double rooms accomodate two.
  2. there is only one group head
  3. there is only one manager
  4. there is only a project head per project
- preferences
  1. offices of the group head and staff should be close.
  2. group heads should be given a double-size office.
  3. project heads should be close to the group head.
  4. twin offices should be shared by people in different projects.
  5. researchers are not eligible for single offices.
  6. a non-smoker cannot share a room with a smoker.
- criteria satisfaction — solutions that satisfy all the constraints are preferred if they exist. Otherwise, solutions that satisfy most of the criteria are acceptable.

The data set provided made the problem relatively simple. For example, there were a lot of double-size and single-size offices compared to how many people needed to be assigned. There were only 4 smokers. There were 8 projects, each of 1-3 people. These data made the constraints relatively easy to satisfy. The participants in the Sisyphus conferences describe their systems with a focus on how this particular problem and data were implemented with their architecture. While it is interesting to see that an architecture can represent and solve this problem with the data proposed, it may be more useful to focus on how the architecture would implement alternative definitions of this problem. This was tried out when the conference organizers issued a second statement of this Sisyphus problem (described at the end of the Appendix) is a variant of the first data set but with one more smoker. This small change produced a data set that had no satisficing solution, which made some architectures fail to return any solution at all. This sort of exercise is very useful to understand the strengths and weaknesses of an architecture. But it is nonetheless a small variation along one single dimension. It would be interesting to learn how the different alternative architectures compare in other dimensions. For example, how would the system's performace be affected if:

- More rooms, more people, more projects (this would test scaling up issues)
- More (and more problematic) smokers (as in second Sisyphus task, but more than one)
- More managers
- More people in each project
- Absolutely no assignments with smoker/non-smoker
- Hackers cannot share rooms with non-hackers
- New types of workers (e.g., visiting researchers)
- Required to respect officemate assignments as they are currently stand in the building before the group moves to the castle
- The user has strong preferences on which constraints to violate

Notice that the data determines whether a constraint in the problem definition is making the problem hard to solve. In this case, the fact that the number of smokers is even makes this data set simpler to solve with regard to the non-smoking constraint, compared to data sets that include many smokers. And in the extreme, if there are no smokers at all then the constraint becomes nonrelevant. Similarly, if there are many more rooms than people, then this constraint does not really affect the complexity of the problem.

It is noticeable that the planning and knowledge acquisition communities usually base their analyses in almost mutually disjoint items from the list above. The planning community focuses mostly on computational efficiency, while the knowledge acquisition groups concentrate on modeling, knowledge representation, and system development. Are our planning systems concerned only with execution issues? Are our knowledge acquisition tools so submerged in the real world that that there is not much room for computational issues? We would expect the answers to be negative in both cases. Our analyses and evaluations in the two areas should make an effort to- cover both grounds.

Appendix: The Office Assignment Problems of Sisyphus '91 and '92

The task is described in detail in (Linster 1994). In essence, the members of a research group need to be assigned to new offices. Their characteristics are as follows:

The offices in the floor plan are numbered counterclockwise. In addition, the problem statement included some restrictions such as:

- offices of the group head and staff should be close.
- group heads should be given a double-size office.
- project heads should be close to the group head.
- twin offices should be shared by people in different projects.
- researchers are not eligible for single offfices.
- a non-smoker cannot share a room with a smoker.

The problem statement contains additional information that we do not include here because of lack of space, including a sample protocol of an expert solving the problem.

The second version of the problem stated that Katharina N. left the group and Christian I. joined as a researcher on the MLT project that smokes and hacks.

References

Allemang, D., and Rothenfluh, T. 1992. Acquiring knowledge of knowledge acquisition: A self-study of Generic Tasks. In Wetter, T.; Althoff, K.-D.; Boose, J.; Gaines, B.; Linster, M.; and Schmalhofer, F., eds., *Current developments in knowledge acquisition–EKAW92*, volume 599 of *LNAI*, 353 – 372. Heidelberg: GI, ECCAI.

Brachman, R., and Schmolze, G. 1985. An overwiev of the KL-ONE knowledge representation system. *Cognitive Science* 9(11):216–260.

Burton, A. M.; Shadbolt, N. R.; Rugg, G.; and Hedgecock, A. 1990. The efficacy of knowledge elicitation techniques: A comparison across domains and levels of expertise. *Knowledge Acquisition* 2(2):167 – 178.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3-4):189–208.

Hanks, S.; Pollack, M. E.; and Cohen, P. R. 1993. Benchmarks, test beds, controlled experimentation, and the design of agent architectures. *AI Magazine* 14(4).

Karbach, W., and Voß, A. 1991. Glueing together small solutions: An office planning model in MODEL-K. In Linster, M., ed., *Sisyphus'91: Models of problem solving*, volume 663 of *Technical Report*. St. Augustin: GMD.

LaFrance, M. 1987. The knowledge acquisition grid: A method for training knowledge engineers. *Int. Journal of Man-Machine Studies* 26:245–255.

Laird, J. E.; Rosenbloom, P. S.; and Newell, A. 1986. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning* 1(1):11–46.

Linster, M. 1993a. Explicit and operational models as a basis for second generation knowledge-acquisition tools. In David, J.-M.; Krivine, J.-P.; and Simmons, R., eds., *Second generation expert systems*. Springer Verlag. 477 – 506.

Linster, M. 1993b. A review of Sisyphus 91 & 92: Models of problem-solving knowledge. In Aussenac, N.; Boy, G.; Gaines, B.; Linster, M.; Ganascia, J.-G.; and Kodratoff, Y., eds., *Knowledge Acquisition for Knowledge-Based Systems*, volume 723. Toulouse: Springer Verlag.

Linster, M. 1994. Special Issue on Sisyphus: Models of Problem Solving. *Int. Journal of Human-Computer Studies* M. Linster (Ed.) 40 (2).

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

McAllister, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proceedings of the 1991 National Conference on Artificial Intelligence*.

Newell, A. 1982. The knowledge level. *Artificial Intelligence* 18:87 – 127.

Sandra Marcus, Jeffrey Stout, J. M. 1988. VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine* 9(1).

Stefik, M.; Aikins, J.; Balzer, R.; Benoit, J.; Birnbaum, L.; Hayes-Roth, F.; and Sacerdoti, E. 1982. The organization of expert systems: A tutorial. *Artificial Intelligence* 18(2).

Wielinga, B.; Boose, J.; Gaines, B.; Schreiber, G.; and Someren, M. V., eds. 1990. *Current Trends in Knowledge Acquisition; Proceedings of EKAW90*, Amsterdam: ECCAI.

Yost, G. R. 1993. Sisyphus 1993: Configuring elevator systems. Technical report, Workplace Integration Technologies Group, Digital Equipment Corporation, 111 Locke Drive (LM02/K11), Marlboro, MA 01752. Unpublished.