Experiences with SRL: An Analysis of a Frame-based Knowledge Representation

Mark S. Fox, J. Mark Wright, and David Adam

CMU-RI-TR-85-10

Intelligent Systems Laboratory
The Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

July 1985

This research was sponsored in part by Digital Equipment Corporation, and the CMU Robotics Institute.

Table of Contents

List of Figures

Abstract

The goal of this paper is to examine a single representation language, SRL, and its applications to determine utility of its ideas. Post mortems have been performed before but have the appearance of a massive "weeding" due to the plethora of ideas included in the initial version of the language. What distinguishes SRL is its *evolution* from a research engine to a "production level" language. Its evolution has been hastened by its application to "real" problems, and its transition to industrial use.

1. Introduction

During the latter half of the 70's the field of AI experienced a proliferation of semantic network and frame-based knowledge representation languages: Concepts (Lenat 1976), FRL (Roberts & Goldstein 1977), KLONE (Brachman 1977), NETL (Fahlman 1977), Scripts (Schank & Abelson 1977), Units (Stefik 1979), and SRL (Fox 1979). With the advent of AI techniques as marketable products, we are beginning to see a similar surge of vendor supported knowledge representation languages in the market place: KEE (Intelligenetics 1984), LOOPS (Bobrow & Stefik 1983), ART (Williams 1983).

One would think that before an idea is "productized," a clear understanding of it and its use would have emerged. Yet the majority of the applications of knowledge representation languages have been experimental, and have yet to move into production use. A survey of systems in field test or production use are either rule-based, e.g., R1 (McDermott 1980), ACE (Stolfo 1982), XSEL (McDermott 1983), and CATS-1 (GE 1983), or utilize an ad hoc representation. In the case of knowledge representation languages, though the size of the intersection of frame-based languages has grown larger, no clear subset has yet to emerge; and the field continues to evolve as new ideas are explored, e.g., RLL-1 (Greiner 1980), and MRS (Genesereth 1980).

In this paper, the SRL system and its applications are described, followed by a description of our experiences and what may be concluded from them.

2. What is SRL

2.1. Language Overview

SRL is a frame-based language with the "schema" as its primitive. A schema is a symbolic representation of a concept. Its definition is the summation of its slots and values. Slots are used to represent attributive, structural and relational information about a concept. A schema is composed of a schema name (printed in the bold font), a set of slots (printed in small caps) and the slot's values (Lisp printing conventions are observed). Values can be any Lisp expression and reference schemata when they are strings. When printed, a schema is always enclosed by double braces with the schema name appearing at the top. The **h1-spec** schema (figure 2-1) contains six slots, each of which contains a value.

```
{{ h1-spec
        IS-A: "engineering-activity"
        SUB-ACTIVITY-OF: "develop-board-h1"
        INITIAL-ACTIVITY-OF: "develop-board-h1"
        ENABLED-BY: "TRUE"
        CAUSE: "h1-spec-complete"
        DESCRIPTION: "Develop specifications for the cpu board" }}
```

Figure 2-1: h1-spec Schema

Many of the ideas found in other representation systems have been incorporated into SRL. These include meta-information, demons, restrictions on legal slot value and a context facility.

Meta-information may be associated with schemata, their slots, and values in the slots. It is represented by another schema, called a meta-schema, that is attached to the schema, slot, or value. Representing meta-information as schemata provides a uniform approach to representation. The user is provided with access functions for retrieving meta-schemata. Once retrieved, they are manipulated just as any other schema. The meta-information is printed in italics beneath schema, slot or value to which it is attached.

```
{{ h1-spec
      Creator: "mark fox"
      To-Create:: schemac

      IS-A: "engineering-activity"
      SUB-ACTIVITY-OF: "develop-board-h1"
           range: (type "instance" "activity")
      INITIAL-ACTIVITY-OF: "develop-board-h1"
      ENABLED-BY: "TRUE"
      CAUSE: "h1-spec-complete"
      DESCRIPTION: "Develop specifications for the cpu board" }}
```

Figure 2-2: h1-spec Schema

Any slot may have *facets* associated with it. Four facets are defined in SRL: DEMON, DOMAIN, RANGE, and CARDINALITY. The DEMON facet allows Lisp procedures to be associated with a slot. The execution of demons is keyed to particular SRL access functions, such as filling or retrieving the value of a slot. RANGE and DOMAIN facets are used to restrict the values that may fill a slot and the schemata in which a slot may be placed, respectively. The CARDINALITY is used to restrict the number of values that a slot may contain. Values for each facet may be inherited from slots in other schemata.

As in other representation languages, a standard set of relations are provided to the user to form taxonomic and part hierarchies. Slots and values may be inherited automatically between schemata along these relations. One of the novel representational ideas introduced by SRL is *user-defined inheritance relations* (Fox 1979). In most other knowledge representation systems, several relations for inheriting slots and values are defined as part of the representation (e.g., AKO, is-a, virtual-copy). In contrast, SRL offers a facility by which users can define their own inheritance relations, allowing only slots and values of the user's choice to be inherited. In addition, slot structures can be elaborated between schemata, and slots and their values mapped arbitrarily between schemata, as need demands. Inheritance relations are represented by additional slots in a schema. A *dependency mechanism* is integrated into the inheritance facility that notes as meta-information the source of inherited slots and values. Here again, the user can define the dependency relations that are put into place.

Another novel feature provided by SRL is a means of *controlling the search* performed by the inheritance process. Any query of the model may optionally use a *path* to restrict which relations may be traversed while searching for a suitable value to inherit. Paths may also be used to specify the *transitivity* properties of relations. For example, a PART-OF hierarchy for describing a car might represent the **battery** as PART-OF the **electrical system**, and the **electrical system** is PART-OF the **car**. The implicit notion that the **battery** is PART-OF the **car** (i.e., that PART-OF is transitive with itself) is represented using paths.

Contexts in SRL act as virtual copies of databases in which schemata are stored. In the copy, schemata can be created, modified and destroyed without altering the original context. Contexts are structured as trees where each context may inherit the schemata present in its parent context. Hence, only schemata that are used in a context need be explicitly represented there. This avoids copying schemata that will never be used in the context. The context provides for version management and alternate worlds reasoning with SRL models.

Error handling is also schema based. An instance of the **error** schema is created to describe each error encountered by the system. **error-spec** schemata may be defined that specify how to recover from each kind of error.

In order to support large applications, a database system is integrated into SRL. Schemata are stored in a database until they are accessed, at which time they are brought into Lisp. A *cache* of the most recently accessed schemata are kept in Lisp for quick access. When the cache becomes too large, schemata are swapped back to the database using a recency algorithm.

2.2. Extensions to the Language

SRL serves as the core of a knowledge engineering environment called Islisp (ISL 1984). It offers a number of inference tools that operate on schemata: HSRL, PSRL, OSRL, ESRL, and KBS. HSRL (Allen & Wright 1983) takes HCPRVR(Chester 1980), a *logic program interpreter*, and alters it to use SRL models as its axioms. The system combines the *modus ponens* inference of logic programming systems with the representation power of SRL. In addition, the inheritance mechanism provides default reasoning, not available in logic programming environments.

Similarly, PSRL is a *production rule interpreter* that operates on SRL models (Rychener 1985). Production rules and their parts are represented by schemata. A subset of PSRL provides the form and execution pattern of OPS5 rules (Forgy 81). OSRL provides a schema-based object programming facility similar to Flavors (Weinreb & Moon 1981). ESRL (ISL 1984) provides an event mechanism which enables the user to schedule events to occur either in a simulated or normal operating mode. KBS, a knowledge-based simulation system (Reddy & Fox 1982) uses ESRL to perform discrete simulations of systems modeled in SRL. Simulation objects are represented as schemata. An object's associated events and behaviors are represented as slots and values in the schema. An object's event behavior may be inherited along relations which link it to other schemata.

In addition to inference tools, system building tools are provided. RETINAS (Greenberg 1983) is a schema based *window system*. Schemata for windows, displays, and canvases are instantiated to build an interface. Default specifications for windows, etc., may be inherited from the prototype schemata. KBCI (ISL 1984) is a schema based *command system*. Again, the **command** schema is instantiated to create commands. A command interface is defined by a collection of command schemata organized in a SUB-COMMAND-OF hierarchy. CPAK (ISL 1984) is a 2-D graphics package based on the CORE definition. A business graphics facility is provided on top of CPAK.

2.3. Applications

Each of the following applications are supported by one or more corporations with the goal of transferring the technology for internal use. Each system uses SRL as its modeling language and makes extensive use of the RETINAS, KBCI, and graphics package for user interfacing.

- **Callisto**: A project management system which focuses on the semantic representation of activities and product configurations (Sathi et al. 1985a; 1985b). Callisto makes extensive use of the SRL's meta-information, search specifications, user-defined relations, and context. In addition, it uses PSRL for representing managerial project management heuristics, and ESRL for project scheduling. Portions of Callisto are in field test.

- **INET^{tm}**: A corporate distribution analysis system which models and simulates a corporation's manufacturing, distribution, and sales organization (Reddy & Fox 1983). INET uses SRL's meta-information and context mechanism. OSRL is the simulation vehicle, and PSRL is used to represent post-analysis heuristics. INET is now being transferred to the sponsor.

- **ISIS**: A production management system which models, schedules, and monitors activities (Fox 1983; Fox & Smith 1984). ISIS uses all of SRL's facilities, with the majority of the search algorithm implemented in Lisp. ISIS is now being transferred to the sponsor.

- **PDS**: A rule-based architecture for the sensor-based diagnosis of physical processes (Fox et al. 1983). PDS uses the basic schema representation only. PDS is in production use.

- **Rome**: A quantitative reasoning system for long range planning (Kosy et al. 1983; Kosy & Wise 1984). Rome uses SRL's meta-information, context mechanism, and user-defined relations. HSRL is a primary inference mechanism.

What are some of the characteristics of the applications to which SRL has been applied?

- **Size**: The number of schemata in a system are large enough to exceed their practical storage directly in memory.

- **Complexity**: The complexity of decision making required by an application requires the incorporation of many of the types of semantic primitives that have evolved in the field, including time, causality, states, actions, etc., and corresponding inference techniques.

- **Efficiency**: The efficiency of the language is important. Response must be provided in a reasonable amount of time, whether for realtime control or interactive support.

3. Experiences

This section discusses the experiences we have had building knowledge based systems in SRL. Our results have been mixed. Some facilities have proven surprisingly useful, while others remain almost entirely unused. The discussion is organized by facility.

3.1. User-Defined Relations

Definition. User defined relations allow the user to tailor the inheritance definition of their relations to the needs of their application. Each relation is represented by a schema. The inheritance semantics of a relation are specified using inheritance specs. There are five kinds of inheritance specs that allow the user to finely tailor the inheritance of their relations.

inclusion	Specifies slots and values that should be inherited unchanged.
exclusion	Specifies slots and values that are specifically excluded.
elaboration	Specifies a one to many mapping of slots. Values may not be inherited along an elaboration.
map	Specifies one-to-one mappings of slots and values
introduction	Specifies slots and values that are introduced when the relation is created.

Relations may also specify their inverse, which is used to perform automatic inverse linking.

The **previous-activity** relation embodies some of this functionality.

```
{{ previous-activity
    IS-A: "relation"
    DOMAIN: (type "is-a" "activity")
    RANGE: (type "is-a" "activity")
    MAP: "previous-activity-map"
    INCLUSION: "previous-activity-inclusion"
    INVERSE: "next-activity"
    TRANSITIVITY: (repeat (step "previous-activity" all) 1 inf) }}

{{ previous-activity-map
    comment: "the finish-time slot in the range schema of the relation is
              mapped onto the start-time slot of the domain schema.
              Hence, the finish time of the preceding activity
              becomes the start time of the following activity."
    IS-A: "map-spec"
    DOMAIN: "start-time"
    RANGE: "finish-time" }}

{{ previous-activity-inclusion
    IS-A: "inclusion-spec"
    SLOT: "sub-activity-of"
        comment: "the slot which may be inherited from the range
                  of the relation to the domain"
    VALUE: all }}
```

Figure 3-1: Previous-Activity Schema

The **previous-activity** relation allows two kinds of inheritance. First, it maps the previous activity's finish time to the next-activity's start time. Second, it allows the inheritance of the SUB-ACTIVITY-OF slot and its values along the relation.

Reflections. User defined relations have proven to be one of the most extensively used features of SRL. They have been exploited by most of the applications yet built using the language. We have several theories as to their usefulness. First, their use has enabled more inference to take place automatically in the systems. In many applications, relations peculiar to a domain (e.g., next-activity, child-of, etc.) will be used often. Inheritance along these relations could not be supported by other languages since only a few relations (e.g., is-a, instance, part-of) would be provided. To overcome this deficiency, the user would have to provide code in their inference engine to deduce what information could have been inherited. But in SRL, the user may define their own relations and their inheritance semantics, and use them where needed.

Second, they allow the terminology of the models to resemble more closely that of the model builder. Separate relations might be constructed for SUB-CLASS, IS-A and KIND-OF which have the same inheritance properties to make models more understandable.

A third point is perspicuity. A relation incapsulates all the information required to use it, including restrictions on its domain and range of use, its inheritance semantics, and its transitivity. Even local overides to its general definition are defined in the schema (e.g., a platypus is-a mammal but does not lay eggs).

Making the user defined relations work with reasonable speed took a number of iterations. In the first implementation, inheritance specs could be inherited along the relation type hierarchy (i.e., relations could form type hierarchies of arbitrary depth). This was far too slow. The second implementation restricted the definition of relations to avoid excessive searching. That is, a new relation had to be related directly to the "relation" schema via an "is-a" relation. Speed was obtained, but the restriction on the definition of the facility was too great. The third implementation introduced a compiler for relations. This allowed a return to the more general definition of relations. Compiling relations combines the best of both worlds. It has the speed of the limited representation, and the power of the general representation. The only sacrifice is that relations cannot be altered dynamically. This compromise yields a powerful and useable system.

3.2. Demons

Definition. Demons provide a facility for reactive processing within SRL. They may be placed in any slot's meta-schema and are executed based on the SRL function used to access the slot. Demons may be inherited from other schemata in a manner similar to that of values. Each demon specifies what slot access functions causes it to fire. Each demon has an action slot that contains any number of Lisp functions. They are executed either before or after the slot access is performed. There are three kinds of demons. First, the "side-effect" demon has no direct effect on the slot access. Second, the "alter-value" demon alters the values that the access function is using. Third the "block" demon stops the slot access function from executing. They are only valid before the function is performed. The ACCESSOR, ACCESS-VALUE, and CURRENT-VALUE hold information about the call for use by the ACTION functions. The demon schema is defined as follows.

```
{{ demon
    ACCESS: <access>+
        range: (type "is-a" "SRL-access-fn")
    ACCESSOR:
    ACCESS-VALUE:
    CURRENT-VALUE:
    WHEN:
        range: (or before after)
    ACTION:
        range: <Must be a function definition>
    EFFECT:
        range: (or alter-value block side-effect)  }}
```

Figure 3-2: Demon Schema

Reflections. Demons have fallen into disuse because they are very expensive. When the facility is enabled, SRL must attempt to inherit demons on every slot access. *This slows the system down by an* *order of magnitude*, as often many slot access functions are performed internally, for each call to SRL.

All attempts to use demons have used them sparsely. The user found a way to avoid demons eventually, to speed up their program. There are two reasonable approaches to using demons within SRL. The first is to limit their functionality. This would entail restricting inheritance of demons, or the SRL functions that check for demons. If only a subset of the slot access functions of SRL checked for demons, then the system might run at a reasonable speed. The second approach is to use demons extensively. For instance, if most slots had demons for most slot access functions, then the user would not be paying for a facility they were not using.

3.3. Restrictions

Definition. SRL provides a mechanism, for restricting the domain and range of a slot. It is possible to restrict the domain of the slot, the range of the slot, and the number of values in the range. Domain, range and cardinality restrictions are placed in the meta-schema associated with a slot. Also like demons, the values of the various restrictions may be inherited along a meta-schema's relational network.

Reflections. Automatic restriction checking is not used for the same reasons that demons are not used. On every attempt to alter the contents of a slot, SRL must attempt to inherit each facet used for restriction. Restrictions do not merit the associated cost. Restriction checking slows the system down by an order of magnitude.

A facility for manually checking restrictions is used, particularly to check user input. Manual restriction checking gives users the benefit of restriction checking when they need it, but avoids the excessive overhead. Full restriction testing is usually turned on during the debugging phase of a system only, much like array bound checking is provided in a compiler.

3.4. Paths for Transitivity

Definition. Transitivities are an important part of SRL, as they allow the user to test if two schemata are related by a particular relation. For example, the transitivity for the **instance** relation is:

```
(list (step "instance" all) (repeat (step "is-a" all) 0 inf)) .
```

This path specifies to step one INSTANCE relation, and any number of IS-A relations. Using this path, it is possible to determine if one schema is an instance of another. It is also possible to find all the schemata which are related to a particular schema by a relation.

Reflections. Transitivities are used by all SRL applications, some extensively. Both types of transitivities are used. Two factors combine to make transitivities an important addition to SRL. First, they are a very expressive and powerful for model definition. Second, they do not add any fixed cost to other SRL accesses. Therefore transitivities are expressive and economical.

3.5. Paths for Search

Definition. Paths are of the same form as transitivity paths, and are used as an added parameter to slot-value access functions in SRL. A path specification can be used to restrict the relations along which inheritance is to be performed for the particular slot access.

Reflections. Search paths have been gradually introduced to most projects. There are two reasons for restricting the search. The first is selective inheritance. For instance if one path for inheritance is correct at the current state of the user's program, this may be specified by a path argument. Consider the situation where a "dog" schema is related to both "pet" and "guard" via an "is-a" relation.

```
{{ dog
    IS-A: "pet" "guard" }}

{{ pet
    DISPOSITION: docile }}

{{ guard
    DISPOSITION: mean  }}
```

Figure 3-3: Search Paths

Depending on what role the dog is playing, the value of its DISPOSITION slot differs. Search paths enable the user to specify along which relations inheritance is to be performed.

The second reason for focusing the search is to avoid searching branches which the user knows are irrelevant. This is used to improve performance by avoiding an exhaustive search. The user community views paths first as a method for improving efficiency, and second as a tool for selective inheritance. Only one project has ever used search paths for selective inheritance, while most projects use them to speed up their programs.

3.6. Meta-information

Definition. Each schema, slot and value may have a schema attached to it in which "meta-information" is placed. These schemata are manipulated in the same manner as other schemata. Meta-schemata enable the user to embed a wide variety of information in a model. Using meta-information, it is easy for a user to associate semantics with the elements of a model. Meta-information is used to maintain dependencies of slots and values, when they are inherited. It is also used to define facets like DEMON and RANGE.

Reflections. Meta-information is used by all applications, some more extensively than others. Usage falls into three categories: restrictions, documentation, and dependencies. Meta-schemata attached to slots provide information restricting the domain and range of the slot (see section 3.3). Meta-schemata also document who created the schema, slot or value, when and why. Meta-schemata attached to values usually provide dependency information, which describes how the value was derived. The BRUTUS facility (Adam et al. 1984), which was just implemented, uses dependencies to provide both truth (Doyle 1979) and belief (van Melle 1980) maintenance at the meta-level.

There have been divided opinions on the efficiency of meta-information. Using it adds a fixed cost to some kinds of inheritance, but it adds a great deal of power to SRL. The result is that automatic generation of meta-information has been separated from maintaining dependencies. This means that users can now use meta-information without increasing the cost of inheritance. This compromise will make meta-information cheap to use, as there is no overhead unless a user wants to maintain dependencies for inherited information.

3.7. Contexts

Definition. SRL has a context facility, that allows the user to have different data spaces for schemata. Contexts are defined in 2.1

Reflections. The primary use of contexts has been to support version management of knowledge bases, and "what-if" reasoning. In the former, new contexts are sprouted, in a hierarchical fashion, as alternative or successive versions of the knowledge base are created. This has been quite useful during model building and testing in INET and KBS, in general. Other systems such as ROME use it to support reasoning about alternative scenarios. In this role, the use of contexts is limited, since there does not exist the ability to relate schemata in two different contexts.

3.8. Database Interaction

Definition. SRL uses a database in order to deal with very large knowledge bases. This allows models which are larger than the memory available to LISP. It also provides a convenient facility for saving knowledge bases. SRL uses a cache for fast access of the most recently used schemata. The database system greatly extends the upper limit on the size of a knowledge base.

Reflections. There are two performance problems with using a database. First, schemata must be copied in and out of the database. This is a reasonably expensive operation. In addition to copying schemata, there is added expense to determine that a schema is not in the knowledge base, as the database must be checked. This was a problem when determining whether a slot was a relation involved looking at the possibly nonexistent schema which represents the slot. Second; users can not have pointers to schemata, because not all schemata are resident in memory. This means that a user's reference to a schema must be converted into a schema every time the user calls SRL. Never-the-less, without the database, the large applications to which SRL has been applied would not be "doable."

3.9. Efficiency

Definition. Efficiency, as defined by the speed with which information may be created and accessed, has become increasingly important as the complexity of the models in SRL increased.

Reflections. As soon as people started writing real programs in SRL, speed became a constant issue. Some projects push SRL to be as fast as possible. Many design decisions balance efficiency versus functionality. To increase the speed of SRL, the decision was made to compile relations, and make many of SRL's features selectable via user switches. For example, value caching, restriction checking, demon execution, meta-information creation, dependency maintenance and other facilities are user selectable. This has provided a good balance between those who require speed and those who require power.

4. Conclusion

A number of features have proven useful in most of our applications. In particular, user-defined relations for adapting the representation to the user's domain, meta-knowledge such as dependencies and facets, relational path specifications for both transitivity checking and search restrictions, contexts for knowledge-base version control, and the caching system for managing large schema bases.

Efficiency has been the overriding concern governing the acceptability of a particular feature in SRL. Both demons and restriction checking have fallen into disuse (except the latter for debugging) because they "overload" schema access functions. While such concerns may be ignored in lieu of faster machines, the inherent complexity of relational search (when information is non-local) in large knowledge bases invalidates such approaches. Two solutions present themselves. The first is an interim solution. Current technology enables the creation of an "SRL machine." It would be a micro-programmed, multi-processor database machine which performs schema accesses and search. The longer term solution lies in the work of connection machines as proposed by Fahlman et al. (1983) and Hillis (1981).

5 References

Adam, D., B. Allen, M. Fox, and P. Spirtes. 1984. "Brutus: A System for Dependency and Belief Maintenance." Technical report, Robotics Institute, Carnegie-Mellon University. In preparation.

Allen, B. P., and J. M. Wright. 1983. "Integrating Logic Programs and Schemata." *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany.

Bartlett, F. C. 1932. *Remembering*. Cambridge: Cambridge University Press.

Bobrow, D. B., and M. Stefik. 1983. "The LOOPS Manual." Xerox PARC, Palo Alto, California.

Bobrow, D., and T. Winograd. 1977. "KRL: Knowledge Representation Language." *Cognitive Science* 1, no. 1.

Bobrow, D., and T. Winograd. 1977. "Experience with KRL-0, One Cycle of a Knowledge Representation Language." *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 213-222. Cambridge, Mass.

Brachman, R. J. 1977. "A Structural Paradigm for Representing Knowledge." Ph.D. thesis, Harvard University.

Chester, D. 1980. "HCPRVR: An Interpreter for Logic Programs." *Proceedings of the National Conference on Artificial Intelligence*.

Fahlman, S. E. 1977. "A System for Representing and Using Real-World Knowledge." Ph.D. thesis, Artificial Intelligence Laboratory, MIT, AI-TR-450.

Fahlman, S. E., G. E. Hinton, and T. J. Sejnowski. 1983. "Massively Parallel Architectures for AI: NETL, Thistle, and Boltzmann Machines." *Proceedings of AAAI-83* 109-113.

Forgy, C. L. 1981. "OPS5 User's Manual." Department of Computer Science, Carnegie-Mellon University.

Fox, M. S. 1979. "On Inheritance in Knowledge Representation." *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* 282-284. Tokyo, Japan.

Fox, M. S. 1983. "Constraint-Directed Search: A Case Study of Job-Shop Scheduling." (Ph.D. thesis.) Technical report, Robotics Institute, Carnegie-Mellon University.

Fox, M. S., S. Lowenfeld, and P. Kleinosky. 1983. "Techniques for Sensor-Based Diagnosis." *Proceedings of the International Joint Conference on Artificial Intelligence*. Karlsruhe, West Germany.

Fox, M., and S. Smith. 1984. "ISIS: A Knowledge-Based System for Factory Scheduling." *International Journal of Expert Systems* 1, no. 1.

General Electric. 1983. "Delta/CATS-1." *Artificial Intelligence Report.*

Genesereth, M. R., R. Greiner, and D. Smith. 1980. "MRS Manual." Computer Science Department, Stanford University.

Greenberg, M. 1983. "RETINAS User's Manual." Internal report. Robotics Institute, Carnegie-Mellon University.

Greiner, R. 1980. "RLL-1: A Representation Language Language." HPP-80-0, Computer Science Department, Stanford University.

Hillis, W. D. 1981. "The Connection Machine." Technical report 646, MIT AI Lab., Cambridge, Mass.

IntelliGenetics. 1984. "KEE^{tm} User's Manual." Third edition. Palo Alto, Cal.: IntelliGenetics, Inc.

ISL. 1984. "Intelligent Systems Laboratory Software Systems Manual." Internal report, Robotics Institute, Carnegie-Mellon University.

Kosy, D., and V. S. Dhar. 1983. "Knowledge-Based Support System for Long Range Planning." Technical report, Robotics Institute, Carnegie-Mellon University.

Kosy, D., and B. Wise. 1984. "Self-Explanatory Financial Planning Models." *Proceedings of the American Association for Artificial Intelligence.* Austin, Texas.

Lenat, D. 1976. "AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search." Ph.D. thesis, Computer Science Department, Stanford University.

McDermott, J. 1980. "R1: An Expert in the Computer Systems Domain." *Proceedings of the First Annual National Conference on Artificial Intelligence* 269-271. Stanford University.

Minsky, M. 1975. "A Framework for Representing Knowledge." *Psychology of Computer Vision,* P. Winston (Ed.). New York: McGraw-Hill.

Reddy, Y. V., and M. S. Fox. 1982. "KBS: An Artificial Intelligence Approach to Flexible Simulation." CMU-RI-TR-82-1, Robotics Institute, Carnegie-Mellon University.

Reddy, Y. V., and M. S. Fox. 1983. "INET: A Knowledge-Based Simulation Approach to Distribution Analysis." *Proceedings of the IEEE Computer Society Trends and Applications.* National Bureau of Standards, Washington, D.C.

Roberts, R. B., and I. P. Goldstein. 1977. "The FRL Manual." MIT AI Lab. Memo 409, MIT, Cambridge.

Rychener, M. 1984. "PSRL User's Manual." Technical report, Robotics Institute, Carnegie-Mellon University. Internal report.

Sathi, A., M. Fox, M. Greenberg, and T. Morton. 1985a. "Callisto: An Intelligent Project Management System." Technical report, Robotics Institute, Carnegie-Mellon University.

Sathi, A., M. Fox, and M Greenberg. 1985b. "The Application of Knowledge Representation Techniques to Project Management." *Transactions on Pattern Analysis and Machine Intelligence.*

Schank, R., and R. Abelson. 1977. *Scripts, Plans and Understanding*. Hillsdale, NJ: Lawrence Erlbam Assoc., Inc.

Stefik, M. 1979. "An Examination of a Frame-Structured Representation System." *Proceedings of the Sixth International Joint Conference on Artificial Intelligence.*

Stolfo, A. 1982. "ACE: An Expert System Supporting Analysis and Management Decision Making." Technical report, Computer Science Dept., Columbia University.

van Melle, W. 1980. "A Domain Independent System that Aids in Constructing Knowledge-based Consultation Programs." Ph.D. thesis, STAN-CS-80-820, Computer Science Dept., Stanford University.

Weinreb, D., and D. Moon. 1981. "Lisp Machine Manual." Fourth edition. Cambridge: Symbolics, Inc.

William, C. 1983. "Advanced Reasoning Tool: Conceptual Overview." Inference Corp., Los Angeles, Cal.

Wright, J. M., M. S. Fox, and D. Adam. 1984. "SRL/1.5 Users Manual." Technical report, Robotics Institute, Carnegie-Mellon University.