The Design Space of Frame Knowledge Representation Systems

Peter D. Karp
Artificial Intelligence Center
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
pkarp@ai.sri.com

May 5, 1993

SRI AI Center Technical Note #520

Key words: Knowledge representation, frame, semantic network, description logic, terminologic reasoner, design principles, design space

Contents

ABSTRACT

In the past 20 years, AI researchers in knowledge representation (KR) have implemented over 50 frame knowledge representation systems (FRSs). KR researchers have explored a large space of alternative FRS designs. This paper surveys the FRS design space in search of design principles for FRSs. The FRS design space is defined by the set of alternative features and capabilities — such as the representational constructs — that an FRS designer might choose to include in a particular FRS, as well as the alternative implementations that might exist for a particular feature. The paper surveys the architectural variations explored by different system designers for the frame, the slot, the knowledge base, for access-oriented programming, and for object-oriented programming. We find that few design principles exist to guide an FRS designer as to how particular design decisions will affect qualities of the resulting FRS, such as its worst-case and average-case theoretical complexity, its actual performance on real-world problems, the expressiveness and succinctness of the representation language, the runtime flexibility of the FRS, the modularity of the FRS, and the effort required to implement the FRS.

INTRODUCTION

In the past 20 years, AI researchers in knowledge representation (KR) have implemented over 50 frame knowledge representation systems (FRSs). KR researchers have explored a large space of alternative FRS designs. The central goal of this paper is to present and elucidate design principles for FRSs, and to note where such principles are lacking, i.e., to identify open problems in KR. The FRS design space is defined by the set of alternative features and capabilities — such as the representational constructs — that an FRS designer might choose to include in a particular FRS. The design space also includes the alternative implementations that might exist for a particular feature.

The foremost principle in any area of design is an understanding of what the design space is. This paper provides that understanding by surveying the FRS design space, and by providing a road map to the FRS literature. As well as elucidating what the FRS design space is, this survey should help to decrease the frequent duplication of effort where different researchers rediscover the same new points in the design space. In addition, an understanding of the current range of FRS behaviors is of crucial importance to the standardization efforts now under way in the KR community [63]. Ideally, an "interlingua" for knowledge representation would be able to represent any knowledge that can be represented in any existing FRS. More realistically, those representational constructs that are excluded should be excluded intentionally, rather than due to ignorance of their existence. Additional principles should aid FRS designers in choosing the optimal FRS design for a given class of applications.

The FRS design space is quite large, therefore this paper does not attempt to cover all of it. We do not consider FRS features such as graphical user interfaces, context mechanisms, truth maintenance systems, production-rule systems, or declarative query languages. Furthermore, FRSs form a subset of all KR systems. This paper is not concerned with other types of KR systems such as theorem provers, nonmonotonic reasoners, or temporal reasoners. It is concerned with substantial FRS implementations that have been employed in complex applications such as natural language understanding and medical diagnosis.

FRSs are known by a variety of names, including semantic networks, frame systems, description logics, structural inheritance networks, conceptual graphs, and terminologic reasoners. Some researchers may object to my lumping all these types of KR systems together in one survey. Although differences do exist between different subclasses of FRSs, and the plethora of names for these systems are not completely synonymous, past authors have interchanged these names enough that we would be hard pressed to provide precise, consistent definitions for all of these different terms. Terminology aside, it is productive to survey these systems together because they were developed by closely related communities of researchers with similar problem-solving goals, because they have been tested in real-world application domains, because their design spaces overlap to a large degree, and because they share many of the same design principles. These properties do not apply to other KR systems, and a survey of all KR research would not fit in a single paper, therefore my aggregation stops before that of Schubert, who sees a convergence in all the major KR schemes [82].

The intended audience for this paper is quite wide, ranging from experts in KR, to re-

searchers in other areas of AI, to database researchers. This paper should be of great interest to database researchers because of the high degree of similarity between FRSs and object-oriented databases since the two classes of systems are different variations of essentially the same data model. The paper will also be valuable to FRS users who wish to select the existing FRS whose capabilities best meet the requirements of a problem at hand.

The paper begins with an overview of FRSs in a tutorial style for readers who are not well acquainted with these systems (Section 1). Section 2 discusses in more detail what design principles for FRSs should tell us, and what design principles have thus far been elucidated. Section 3 provides a terse road map to the FRS literature from a historical perspective by listing the major families of FRSs, and by listing the general literature citations for each FRS. This organization puts most FRS citations in one place and avoids the need to repeatedly list the same citations later in the paper. Section 4 begins the in-depth exploration for the FRS design space by considering the diverse models of the frame that different researchers have explored. Section 5 considers alternative slot designs, and Section 6 considers alternative models of inheritance. Section 7 considers mechanisms for providing persistent knowledge base storage, and Section 8 discusses object-oriented and access-oriented programming systems within FRSs. Finally, Section 9 discusses the classification operation and its role within FRSs.

As a result of researching this survey I identified a number of methodological problems in KR research. To focus the subject of this paper, and because of space limitations, they will be discussed in a separate publication.

1 OVERVIEW OF FRAME REPRESENTATION

This section presents a very brief, simple introduction to frame representation. It begins by describing the general classes of tasks that FRSs are used for. Next it presents the structure of frames. We then consider the services that FRSs provide to users and to application programs. This section simplifies the notion of a frame considerably to set the stage for the detailed analysis that follows in the remainder of the paper. Other introductions to frame representation can be found in [27, 30, 11, 26].

It is important to note what FRSs are not: FRSs are not equivalent to "knowledge-based systems". This term is very general and encompasses a large number of artificial-intelligence techniques. FRSs are a subset of knowledge-representation systems, which in turn are used to build knowledge-based systems.

1.1 The Structure of Frames

A frame is a data structure that is typically used to represent a single object, or a class of related objects, or a general concept (or predicate). Researchers have used a number of words synonymously for the word *frame*, including *memory unit*, and *unit*. Whereas some systems define only a single type of frame, other systems distinguish two or more types,

such as *class* frames and *instance* frames. The former represent classes or sets of things (such as the class of all computers in the VAX-11 family or the class of all VAX-11/780s) and the latter represent particular instances of things (such as a particular VAX-11/780).

Frames are typically arranged in a *taxonomic hierarchy*[1] in which each frame is *linked* to one (or in some systems, more than one) *parent* frame. A parent of a frame A represents a more general concept than does A (a superset of the set represented by A), and a child of A represents a more specific concept than does A. A collection of frames in one or more inheritance hierarchies is a *knowledge base* (KB).

Frames have components called *slots*. The slots of a frame describe attributes or properties of the thing represented by that frame, and can also describe binary relations between that frame and another frame. In addition to storing values, slots also contain restrictions on their allowable values. We might stipulate that the `Word_Size` slot defined in the `COMPUTER` frame must be an integer between 1 and 100. Slot definitions often have other components in addition to the slot name, value, and value restriction, such as the name of a procedure than can be used to compute the value of the slot, and a justification (in the truth-maintenance sense) of how a slot value was computed. These different components of a slot are called its *facets.*

Inheritance causes slot definitions to propagate down the taxonomic hierarchy. For example, when we create a frame called `VAX-8000_Family` as a child of the `VAX_Family` frame, `VAX-8000_Family` automatically acquires the slot `Word_Size,` and the value of this slot automatically becomes `32`. Thus `VAX-8000_Family` has inherited its `Word_Size` slot from `VAX_Family`. Similarly, `VAX_Family` might have inherited the `Word_Size` slot from the frame `DIGITAL_Computer,` but we would not have given the `Word_Size` slot the value `32` in `DIGITAL_Computer` because not all computers manufactured by DIGITAL have a word size of 32. Thus, the value `32` was defined *locally* in `VAX_Family`.

Inheritance is a tremendously useful tool for engineering complex knowledge bases. When a user creates a new frame and that frame inherits slots from its parent, the inherited slots form a template that guides the user in filling in knowledge about the new concept. Because all slot and facet information is available at run time (in contrast to object-oriented programming languages such as C++), it is accessible to a program such as a user interface that guides the user in entering new knowledge. For example, the user interface can directly determine the slot datatype and value restrictions. Inheritance also facilitates systematic changes to complex knowledge. If we discover that all Cray computers can run UNIX in addition to CrayOS, we can encode this information in one place: by altering the value of the `Operating_Systems` slot in the `Cray_Computer` frame.

Some FRSs compute a relation between class frames called *subsumption* that allows the FRS to automatically determine the correct position of a class in a taxonomic hierarchy (to *classify* the class). Frame A subsumes Frame B if A defines a more general concept than does B, meaning that every instance of the concept B is an instance of A. For example, because the value of the `Manufacturer` slot of `DIGITAL_Computer` is `DIGITAL`,

[1] Synonyms: generalization–specialization hierarchy, is–a hierarchy, class–subclass hierarchy, AKO (a kind of) hierarchy, and inheritance hierarchy.

whereas the `Manufacturer` slot of `Computer` is constrained only to be an instance of the class `Corporation`, a FRS could infer that `DIGITAL_Computer` is subsumed by `Computer`.

1.2 A Functional View of Frame Representation Systems

This section discusses FRSs from a functional perspective by examining two classes of operations that these systems provide to problem-solving programs: direct storage and retrieval of knowledge, and inferential problem solving.

1.2.1 Storage and Retrieval of Knowledge

Most FRSs provide a library of (usually LISP) functions that application programs can call to perform such actions as: adding a new value to a slot, deleting one or more values of a slot, retrieving the current value of a slot, creating a new frame with specified parents, changing the parents of a frame, deleting a frame, renaming a frame, adding a new slot to a class, and adding a facet to a slot.

In addition to the function-call library, some systems allow the user to accomplish these same functions with a graphical user interface. For example, KEE, STROBE, CYCL, and KREME [2] allow the user to create graphical displays on a workstation of both the taxonomic hierarchy of a knowledge base, and of the slots within a given frame. Items within these displays are mouse sensitive, and can be used to call up menus from which the operations described in the previous paragraph can be selected.

In an approach pioneered by KRYPTON, some FRSs provide a declarative language that users can employ to both query a KB, and to assert new facts into a KB. In PROTEUS, for example, the query `(Manufacturer ?X:Computers DIGITAL)` would return a list of assertions describing all children of the `Computers` frame whose `Manufacturer` slot contained the value `DIGITAL`. Little research has been performed on optimizing the evaluation of FRS queries.

1.2.2 Problem Solving and Inference

Application programs that interact with a FRS typically employ either production rules or classification to perform inference based on knowledge stored in the FRS. Most FRSs provide either production rules or classification, whereas LOOM and CLASSIC support both. In KEE and CLASS [80], each production rule is itself encoded as a single frame. In KEE and PROTEUS, queries such as those described in the previous section can invoke a backward-chaining production-rule interpreter to derive the queried slot value. Similarly, THEO users can attach PROLOG rules to slots to cause THEO to backward chain to derive queried slot values. KEE and PROTEUS can also invoke forward chaining when new slot values are asserted.

Classification is used to support inference in two different ways. First, the very act of classification can be a problem-solving action, for example, if a system can recognize a description of a patient as an instance of a disease class, it has computed a diagnosis. Second, in the KL-ONE family of FRSs, classification is a key component of the query processor that allows the system to reason about relationships among terms used in a query and terms in a knowledge base. These systems answer a query by translating the query into a concept description, and then classifying that concept to determine its placement in the taxonomic hierarchy. All concepts below the query concept in the hierarchy are subsumed by the query, and thus comprise the answer to the query.

The principal principle of KR that has thus far been discovered concerns the complexity of computing subsumption (and therefore, classification). Researchers have compared the cost of computing subsumption in a number of different FRS representation languages, and have found that the more expressive the language, the higher the cost of computing subsumption within that language. This result is called the expressiveness–tractability tradeoff, and will be discussed in more detail in Section 9.1.

Some FRSs leverage their inference capabilities by combining them with context mechanisms and truth-maintenance systems [20]. These facilities are valuable for investigating alternative problem solutions in parallel, and for tracking the dependence of problem solutions on underlying assumptions. Context mechanisms exist in THEO, KEE [29], STROBE, CYCL [36], SRL, LOOM, and CRL; truth-maintenance systems are present in THEO, KEE, CYCL, LOOM, CLASSIC, KL-TWO, and PROTEUS.

2 DESIGN PRINCIPLES FOR KNOWLEDGE REPRESENTATION SYSTEMS

The large size of the FRS design space implies that the designer of an FRS must make many decisions. For example, we will see that she must decide what model of the frame and the slot to utilize, what inheritance mechanism(s) to use, whether to employ classification, and what subsumption algorithm to use if classification is employed. A comprehensive set of principles of KR should guide FRS designers — and users — through a complex web of choices. FRS users need to know what combination of representational constructs will allow them to quickly build an application that has acceptable performance. They need to know what representational constructs can encode the knowledge in their domain most naturally and succinctly, to yield an application that can be maintained easily as it evolves. Users need to know the theoretical costs and benefits of FRS features, and they need to know how a particular FRS will perform under the demands of their application. KR principles should guide users in choosing the optimal FRS for a particular problem — the one with the maximum benefits and the minimum costs.

When designing an FRS to solve one or more classes of application problems, implementors must address a superset of these issues. As well as anticipating what combination of representational constructs will yield sufficient expressiveness and performance for the

applications, the implementors must make a number of engineering decisions. For example, implementors must decide among alternative implementation strategies for the representational constructs they have chosen. And although we might hope that the implementations of every FRS feature are independent, they often interact to yield an FRS that is not modular and is therefore difficult to develop, debug, maintain, and improve.

I claim that comprehensive FRS design principles are largely lacking. The main principle of KR that has thus far been elucidated is the expressiveness–tractability tradeoff that relates the expressiveness of a representation language with the cost of computing classification within that language. This principle is clearly valuable since it helps users and implementors understand the expressive benefits and the worst-case computational costs of several representation languages. However, this principle describes only the worst-case theoretical impact of one class of representational constructs (concept-definition constructs) on one type of FRS operation (classification). Many additional principles are needed to cover other representational constructs, other FRS operations, other theoretical performance besides worst case, actual performance in addition to theoretical performance, and other criteria besides performance and expressiveness. More specifically:

- Classification is only one of many operations that FRSs compute. Some FRSs do not even compute classification. We must know the impact of different representational constructs on other operations such as computing inheritance, storing and retrieving slot values, and production-rule inference within an FRS.
- Expressiveness–tractability analyses have not considered representational constructs such as metaclasses, facets, and inheritance across multiple links. Although not all of these constructs will affect the classification operation, they will certainly impact some FRS operations. In general we should know the effects of every representational construct on the performance of every FRS operation.
- Worst-case theoretical results are not always representative of the average case, and theoretical results are not always constraining in practice. Theoretical principles concerning average-case behavior of various FRS operations are generally lacking. Engineering principles concerning choices of data structures and algorithms are even fewer.
- Performance and expressiveness are not the only factors to consider when choosing a representation. Two representations might have equal expressive power, but one might be much more succinct for a particular application. In addition, there is a tradeoff between the run-time flexibility of the FRS (the degree to which knowledge that the FRS maintains can be altered at run time as opposed to the time of definition), and the performance of the FRS. Also, the modularity of an FRS implementation is affected by the choice of representational constructs and the implementation of those constructs. Other factors concern the effort involved in implementing a particular feature, and the frequency with which that feature is used in different applications; a feature that is difficult to implement but that is hardly ever used should probably be disregarded. We need much more knowledge about the costs and benefits of different FRS features with respect to all of these factors.

3 THE FAMILIES OF FRAME KNOWLEDGE REPRESENTATION SYSTEMS

This section provides a very brief overview of implemented frame representation systems of the past and present. An ideal treatment of the evolutionary relationships between FRSs would differ from the treatment presented here in several ways. Ideally we would like to know the general characteristics of each family, as well as how and why a given system differs from its parent(s): what shortcomings did an author discern in the parent system, what differences did the author introduce in the child FRS to remedy these shortcomings, were these modifications successful, and what were the costs of these changes in terms of the metrics listed in Section 2? Unfortunately, space restrictions preclude a full treatment of these issues, and, more importantly, authors rarely document these aspects of their work systematically. This practice not only obscures the intellectual history of frame representation, but it makes principles of FRSs more difficult to derive.

Therefore this section lists the one or two most influential parents of a number of FRSs, either as identified by the FRS author, or as ascertained by the author of this paper with a high degree of certainty. I have been unable to make this determination for many systems. General literature references for particular FRSs are presented in this section of the paper only; later sections provide some references to support specific points.

Figure 1 shows several FRS families. The UNITS family originated at Stanford University in the late 1970s. Its members include the UNIT Package [89, 93], STROBE [88, 90, 91], CLASS, RLL [35, 34], CYCL [47, 49, 48], ARLO [37], THEO [57], JOSIE [60], OPUS [28], and the commercial systems KEE [42] and KAPPA.

The KL-ONE family originated at Harvard University in the early 1970s. Its members include KL-ONE [15], NIKL [41, 6, 76], KANDOR [65], KL-TWO [98], K-REP [52], KREME [2], BACK [67, 99], MUNIN, SPHINX, KRIS [4], MESON, SB-ONE [43], KRYPTON [13], LOOM [101, 40, 51], and CLASSIC [12, 14, 69]. See [76, 51, ?] for historical overviews of the KL-ONE family.

The SRL family originated at Carnegie-Mellon University in the early 1980s. Its members include SRL [31], FRAMEKIT [64], PARMENIDES [86], and the commercial system KNOWLEDGECRAFT.

The FRL family originated at MIT in the mid 1970s. Its members include FRL [71, 70], HPRL [44, 72], and GOLDWORKS.

Several other FRSs do not exist within a larger family. They include PROTEUS [73, 68], FROBS [58], OZONE [45], KRL [7, 46, 8], BB* [32], LOOPS [94, 17], KB [23], SNePS [84, 85], RHET[3, 56], TELOS [59], PARKA [24], ALGERNON [18, 19], FRAPPE [25], Conceptual Graphs [92], MOPS, ART, and NEXPERT.

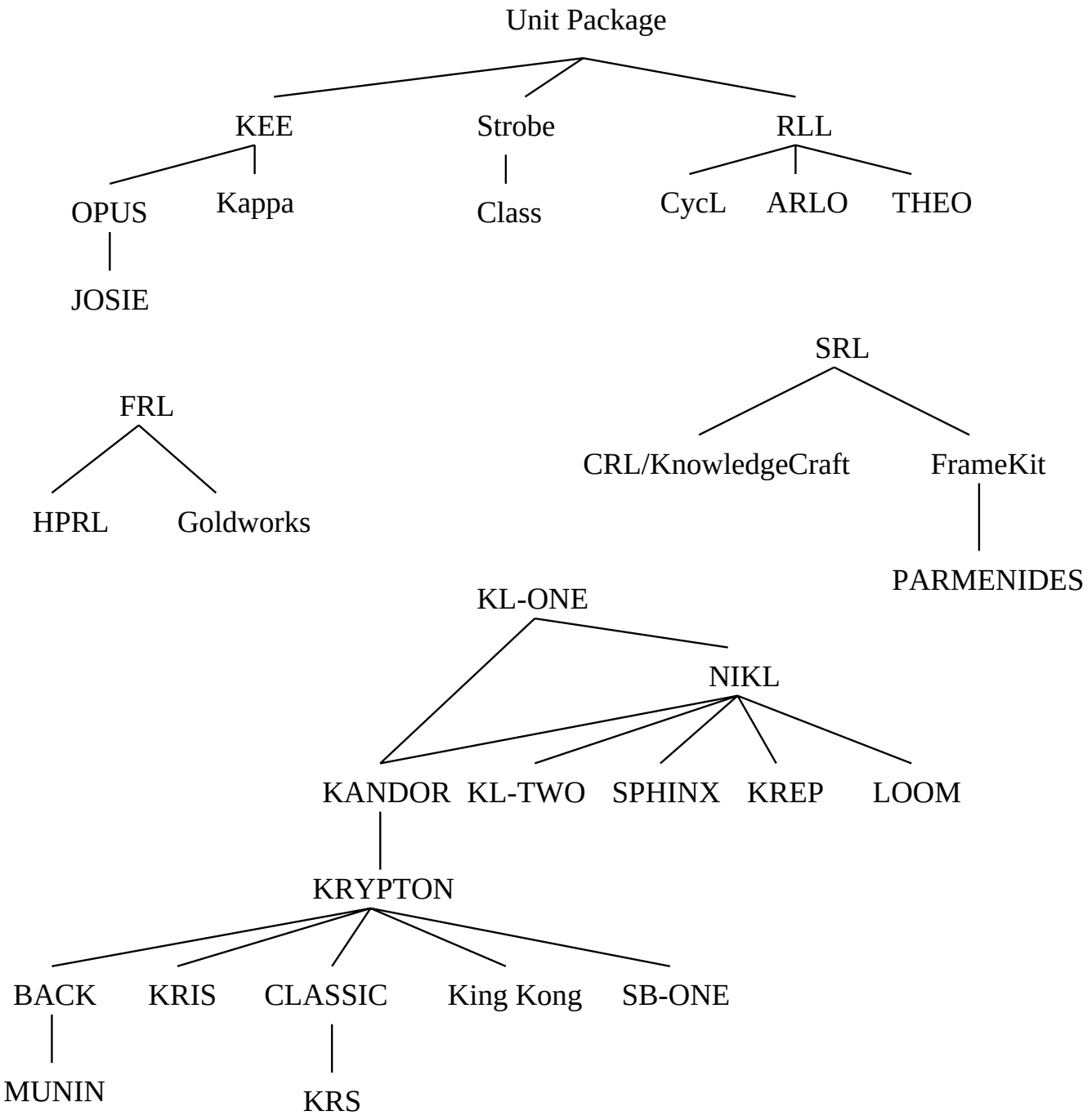

Figure 1: The Unit-Package, SRL, FRL, and KL-ONE families of frame representation systems.

4 FRAMES

Typically, FRSs define two different types of frames: *class frames*[2] represent a class or set of things, a general concept, or an abstraction. Examples: the class of all computers, the set of all computers manufactured by IBM, the concept of a father, or of a mother. *Instance frames*[3] represent individual things — concrete entities that exist in the world. Examples: the particular computer that I am using to type in this sentence, the person who is my father. Different researchers have different views as to the semantics of class and instance frames, and moreover they have defined a variety of other types of frames. This section explores the diversity of frames themselves.

4.1 Link Terminology

Before proceeding with a detailed discussion of frames, this section establishes a standard and comprehensive terminology for referring to the relationships among class and instance frames in a taxonomic hierarchy. Virtually every family of FRSs has its own terminology. The use of redundant and conflicting terminology has hampered communication among knowledge-representation researchers, and has confused researchers in other areas of computer science who often assume that different terms must describe different concepts. I propose the following terminology, which I developed in the spirit of the presentation by Russinoff [73]. Footnotes translate my terminology to that used by previous researchers.

We are interested in naming the relationships that exist between class frames and instance frames. If a class frame C_1 is linked directly above a class frame C_2 in the hierarchy, then we say that C_1 is a *direct-super* of C_2, and that C_2 is a *direct-sub* of C_1; [4] we call the link itself a *super–sub link.* If a class frame C is linked directly above an instance frame I then we say that C is a *template* of I, and that I is an *instance* of C. [5] We define the *all-supers* relation to be the transitive closure of the direct-supers relation, and we define the *all-subs* relation to be the transitive closure of the direct-sub relation (therefore C_1 is an all-sub of C_2 if C_1 is a direct-sub of C_2, or if C_1 is a direct-sub of any all-sub of C_2). [6] Similarly, we say that an instance frame I is in the *all-instances* of a class C if I is an instance of C or an all-sub of C, in which case we would say that C is in the *all-templates* of I. [7]

Finally, we say that A is a *parent* of B if A is either a direct-super or a template of B (in which case B is a *child* of A). We define *ancestor* as the transitive closure of the parent relation, and *descendant* as the transitive closure of the child relation. Table 1 summarizes this terminology, and also presents a notation that I have developed to express these relationships more succinctly.

[2] Synonyms: concept (KL-ONE), collection (CYCL), specialization (UNIT Package), frame (KANDOR), set, schema, generic (FRL), template, node.

[3] Synonyms: individual (KL-ONE, STROBE, FRL), individual object (CYCL), instantiated frame.

[4] Synonyms: superclass and subclass (KEE), parent and child (PROTEUS), superC (KL-ONE).

[5] Synonyms: member-parent-of and member-of (KEE), type-of and instance-of (PROTEUS), individuates

Notation	Meaning	
$C_1 > C_2$	Class C_1 is a direct-super of class C_2	Parents
$C \succ I$	Class C is a template of instance I	
$C_1 < C_2$	Class C_1 is a direct-sub of class C_2	Children
$I \prec C$	Instance I is an instance of class C	
$C_1 \gg C_2$	Class C_1 is an all-super of class C_2	Ancestors
$C \succ\succ I$	Class C is an all-template of instance I	
$C_1 \ll C_2$	Class C_1 is an all-sub of class C_2	Descendants
$I \prec\prec C$	Class I is an all-instance of class C	

Table 1: Notation and terminology for describing inheritance relations.

4.2 The Diversity of Frames

Some FRSs employ only a single type of frame rather than both class and instance frames. NIKL provides for class frames only, because its authors consider NIKL's mission to support the definition of concepts and the relations between them, not to facilitate reasoning about individuals.[8] THEO also defines only one type of frame, because its authors believe that the distinction between classes and instances is sometimes not well defined.

Every other FRS includes at least class and instance frames, but some systems employ the following additional types of frames:

- **Metaclasses** — PROTEUS utilizes frames called *metaclasses* to define sets of PROTEUS classes — every PROTEUS class is an *instance* of the metaclass called `CLASS`. Users can define other metaclasses as direct-subs of the metaclass `CLASS`. For example, we could define a metaclass called `Computer_Family` of which `VAX_Family` (a class) is an instance. `CLASS` is the class of all classes, or synonymously, the set of all sets. LOOPS also employs metaclasses, as does CYCL: every CYCL frame is an instance of either the frame `Collection` or the frame `IndividualObject`; the former types of frames are classes and the latter are instances. Thus the frame `Collection` is really a metaclass that is equivalent to the `CLASS` of PROTEUS. In CYCL, only class frames (instances of `Collection`) can have instances, direct-supers, or direct-subs, whereas only instance frames (instances of a class frame) can have parts.

- **Indefinites** — The UNIT Package and the CLASS FRS employ indefinite frames to represent instances whose identities are unknown (similar to the notion of skolem constants). Indefinites allow a user to say that two individuals are the same without knowing their identities [93].

(KL-ONE).

[6] Synonyms: superclass and subclass (PROTEUS).

[7] Synonyms: member (PROTEUS).

[8] NIKL does define "individual" class frames as classes that have only a single member — but classes nonetheless. To reason about individual objects a user must employ an additional system, for example KL-TWO [98] combines NIKL with a propositional reasoning system called RUP [54].

- **Descriptions** — These frames are variablized classes that are employed in the UNIT Package to represent goals in planning problems [93]. CLASS also has indefinites.
- **Prototypes** — KRL, RLL, and JOSIE employ prototype frames to represent information about a typical instance of a class, as opposed to the class itself and as opposed to actual instances of the class. In these systems instance frames inherit default information from prototype frames rather than from class frames (see Section 6).
- **SlotUnits** — In CYCL, slotUnits[9] are frames that encode information about slots themselves (see Section 5).
- **SeeUnits** — CYCL employs seeUnits as "footnotes" or annotations for other frames. They can hold constraints on slot values, dependency information (such as what inference stored a value in a slot), and epistemological information (who believes a slot value to be true).
- **CompactUnits** — KEE's compactUnits are instance frames that consume less storage and are faster to access than are normal instance frames. A compactUnit must have only one template frame (normal KEE instances can have multiple templates), and the compactUnit must have exactly the same set of slot definitions as its template (in KEE, users can define new slots in an instance frame that were not defined in the template of that frame). LOOM has a similar mechanism; its CLOS instances can provide a more efficient implementation of instance frames.

4.3 Discussion

Few KR principles exist to guide our understanding of the preceding features. Generally speaking, each feature adds expressiveness and potential succinctness to an FRS by allowing us to more faithfully render a complex epistemological landscape. Yet we have little knowledge of the performance costs or benefits of these features, of the effort required to implement them, of their effect on system modularity, of the frequency with which they are used, or of the optimal data structures and algorithms for implementing them.

CompactUnits provide faster performance with less expressiveness and flexibility than normal instance frames, but we do not know how much more performance, nor exactly why restricting an instance to a single template is required to provide this speedup. Presumably it allows a fixed mapping from a slot name to a location in an array. Are the same implementation techniques used in the KEE and LOOM implementations, and if not, exactly how do the speedups compare? Are the changes required to implement compactUnits fairly localized, or are they spread throughout the FRS code?

Metaclasses, prototypes, seeUnits, descriptions, and indefinites extend the expressiveness of FRSs, and in ways not considered in past expressiveness–tractability analyses. Lenat and Guha provide a detailed discussion of the types of knowledge that metaclasses can represent [47, p57]. But once again, we do not have a clear picture of the costs of these constructs.

[9] Synonyms: *relation frames* (OPUS, JOSIE).

And even their expressiveness benefits are not that clear: when are defaults provided by prototypes preferable to class-based defaults? Nado and Fikes argue that prototypes more clearly separate information about class members from information about the class itself [60]. Schoen reports (personal communication, 1991) that little use was made of indefinite frames in CLASS, whereas description frames were used.

Some insight on an engineering issue comes from the PARMENIDES system. All of the FRSs in the UNIT Package family implement classes and instances in essentially the same manner. In contrast, PARMENIDES implements classes and instances quite differently under the assumptions that many more instances than classes will exist in most KBs, and that instances will be accessed more frequently than classes will. PARMENIDES classes are implemented as association lists, whereas instances are implemented as adjustable arrays. Thus instances are more compact than are classes, and instances are faster to access since no search through an association list is required. However, this approach limits the run-time flexibility of PARMENIDES: although new slots can be created at run time by adding elements at the end of the adjustable array, in order to remove or modify slot definitions the user must restart the FRS to rebuild the knowledge base. Also, we have no data on exactly how much of a performance gain this technique yields.

5 WHAT'S IN A SLOT

Every frame consists of a set of slots, which usually represent properties or attributes of the object or concept represented by the frame.[10] Slots are also used to represent binary relations between their containing frame and another frame. The very simplest model of slots gives every slot a name (such as `Manufacturer`), and a value[11] — such as `Data_General`.

Researchers have embellished this simple model in a number of ways. Almost all systems specify a few other attributes for each slot besides its value, such as a slot datatype (described in more detail in Section 5.4), and restrictions on the allowable values for the slot (Section 5.5). Some researchers generalized this notion to allow slots to have arbitrary properties called *facets,* of which name, value, datatype, and value restriction are the usual complement. Other typical facets that we find are: a comment, a measure of belief such as a MYCIN-like certainty factor (used in CYCL), an explanation or justification that specifies what other slot values the current value was inferred from (used in CYCL and in THEO), a non-value (used to represent negation in ALGERNON), a description of what agent believes this slot value (used in CYCL), attached procedures, default values, and a specification of an inheritance mechanism for that class (see Sections 8 and 6).

THEO uses an even more general notion of slots: facets themselves can have facets, to any level — THEO allows an arbitrarily deep nesting of slots within slots within slots. Thus we could define a `Comment` subslot within the `Manufacturer` slot of the `Computer` frame. We could also create a subslot of `Comment`, perhaps to record the name of the user who

[10] The KL-ONE family of FRS calls slots *roles* because a slot such as `Manufacturer` names the entity that "plays the role of" a manufacturer for a given computer.

[11] Slot values are called *fillers* in KL-ONE-speak and *entries* in CYCL-speak.

created the comment. As well as facilitating the attachment of meta-information, THEO uses subslots extensively to cache information derived by a variety of inference mechanisms. For example, imagine that a user has queried the value of `Frank.Children`. If no local value were found, one of the THEO inference mechanisms would cause it to attempt to obtain the value from `Frank.Daughters` if it could determine (from consulting the slotUnit of `Children`) that the `Daughters` relation is a specialization of the `Children` relation. THEO would locally cache within the `Frank` frame some of the information computed in the process of answering this query. For example, it would cache the fact that `Daughters` is a specialization of `Children` within the subslot `Frank.Children.Slotspecs`.

The JOSIE system provides a way of viewing slot values as classes, that allows a wider set of assertions to be made about slot values. That is, if the values of a slot comprise a set, this mechanism lets us treat that set like a class, and use the JOSIE class-definition language to make assertions about that set. Section 6.1 elaborates on this idea.

OZONE takes yet another approach: slots are partitioned into separate groups called *spaces.* Each frame typically has three spaces called the system space (these slots list the name of the frame and the parents of the frame), function space (these slots contain attached procedures), and the variable space (used for most user-defined slots). Spaces provide two major benefits: they define separate name spaces for defining different types of slots, for example to prevent name clashes between system slots and user slots; they also facilitate incremental loading of frame contents from secondary storage (see Section 7) — system-space slots can be loaded independent of slots in other spaces.

FRSs also vary as to whether users can introduce new slots into an instance frame that did not exist in its template. KEE allows this but the KL-ONE family does not, under the interpretation that because a class frame strictly defines what it means to be an instance of some concept, introducing new slots into an instance frame would violate the definition of the concept.

5.1 Slot Notation

This section presents a notation for defining paths through complex frame structures that is an adaptation of the CYCL notation [47]. This notation should prove useful both in expositions and in declarative frame query languages.

To refer to the value of the `Manufacturer` slot of the `Computer` frame we write `Computer.Manufacturer;` to refer to the value of its `Comment` subslot we write `Computer.Manufacturer.Comment`. Note that since the value of a slot is simply a distinguished subslot, `Computer.Manufacturer.value` is equivalent to `Computer.Manufacturer`. Now imagine that we wish to refer to the number of employees of the manufacturer of the VAX-11/780. `Manufacturer` is a binary relation that names a frame which describes the manufacturing corporation. To refer to the `Number_Of_Employees` slot of that frame indirectly, we write `VAX-11/780.Manufacturer->Number_Of_Employees`. Note the difference between this specification and the specification `VAX-11/780.Manufacturer.Number_Of_Employees`. The for-

mer refers to a slot within the frame `Digital` whereas the latter refers to a subslot of the slot `VAX-11/780.Manufacturer`. THEO uses a list notation for the former type of reference only: `(Vax-11/780 Manufacturer Number_Of_Employees)`.

5.2 SlotUnits

Several FRSs contain a type of frame that the authors of CYCL call a *slotUnit*. A slotUnit is a frame that holds definitional information about a single slot that describes the use of that slot throughout a KB. A slotUnit might specify the domain and range of a slot S (the domain of `Manufacturer` is the set of frames in which it makes sense to use this slot — the class `Manufactured_Objects`), and the range of this slot describes its allowable values (instances of the class `Corporations`).

Lenat's experience in CYCL was that it was often desirable to represent a wide variety of information about slots. For example, CYCL (as well as FRAMEKIT and STROBE) use slotUnits to store *inverse* definitions. The slotUnit for the `Manufacturer` slot might record that the inverse of the `Manufacturer` relation is the `Manufactures` relation. Thus, when we record that `Digital` is a value of `Vax-11/780.Manufacturer`, CYCL will automatically add the value `Vax-11/780` to `Digital.Manufactures`. More generally, automatic maintenance of inverse links is equivalent to enforcing the constraint:

$$\forall G\ [G \in F.S \equiv F \in G.S^{-1}]$$

where S^{-1} is obtained from the slotUnit S as $S.inverse$. Although slotUnits provide a place to store general information about slots, note that some information about a slot must be stored in frames containing that slot. For example, although the slotUnit will specify a domain and range for a slot such as `Manufacturer`, additional value (range) restrictions that are defined at a class such as `Japanese_Computer` must be stored in the `Japanese_Computer` frame because they are specific to that class. We might want to constrain the `Manufacturer` of every `Japanese_Computer` to be an all-instance of `Japanese_Corporation`.

Within the Units family, the idea of slotUnits arose in both RLL and OPUS, and is also used in THEO. SRL and FRAMEKIT use slot Units only for slots that represent a binary relation between two frames. The KL-ONE family creates a hierarchy of slot definitions called a *role hierarchy,* however, each slot definition in the role hierarchy is not implemented as a frame, but with a special data structure. The role hierarchy allowed a user to define *role differentiations* — roles whose potential values are by definition a subset of the potential values of another role. Thus, in KL-ONE we can define `CPU_Manufacturer` to be a role differentiation of `Manufacturer` because the allowable values of the former are a subset of the allowable values of the latter. Given this definition, if `Motorola` was a value of `SUN-3.CPU_Manufacturer`, the system could infer that `Motorola` must also be a value of `SUN-3.Manufacturer`, given the subset relation between the two slots. Note that inheritance within a slotUnit hierarchy can be used to encode role differentiations.

5.3 Own Slots, Member Slots, and Bookkeeping Slots

KEE distinguishes what its developers call *member slots* from *own slots,* as do FRAMEKIT and FROBS. Member slots reside only within class frames. They describe properties of instances of that class, and are inherited by children of that class. Own slots, in contrast, can reside within either class or instance frames. Own slots are not inherited by children of a class frame C because these slots represent properties of only the class represented by C, and not its children. (If own slots were employed in a FRS that used metaclasses, the own slots of a class C should be inherited from the template of C (a metaclass) since own slots describe a frame as an instance.)

As an example, an own slot `Fastest` within the class `Computers` that named the fastest known processor should be defined as an own slot because it describes a property of the set of all computers, not a property of the instances of that set. But `Manufacturer` should be a member slot of `Computers` because it denotes a property of every individual computer.

In a similar vein, CYCL defines *bookkeepingSlots* to be slots that describe a frame F itself rather than the concept that F represents; for example, bookkeepingSlots might be used to record the name of the user who created F, and the time of creation.

5.4 Slot Data Types

Most FRSs model slot values as sets (such as CYCL, KEE, and the KL-ONE family). Other systems (such as THEO, OZONE, and the UNIT Package) treat slots as lists of values, the difference between sets and lists being that in lists the order of elements is preserved and duplicate values are allowed. LOOM treats slot values as ordered sets: duplicates are not allowed, but order is significant. Interestingly, virtually no FRS allows the user to explicitly select a slot datatype from among a variety of data types such as sets, ordered sets, lists, and bags. Exceptions to this rule are K-REP, which provides both sets and bags; PARMENIDES, which provides two groups of slot-access functions that treat slots as sets and lists, respectively; and PROTEUS, which allows the user to specify whether a slot is single valued or multivalued. Sets have the advantage over lists of simplifying the logic of frames and of having a well-defined subsumption relation.

Some systems allow the user to specify the datatypes of the individual values in the set or list, for example the UNIT Package allows datatypes of atom, boolean, integer, interval, lisp, number, string, table, text, or unit; KEE provides a similar set of datatypes.

Although not thought of as a datatype, LOOM allows the user to specify whether query operations on specific slots have open or closed-world semantics. CLASSIC provides similar control over the slots in specific instance frames.

5.5 Slot Value Constraints

Researchers have defined mechanisms for specifying constraints on the values of individual slots, and between the values of two different slots. These constraints are viewed as necessary conditions that must hold of the slots to which they are attached. A slot in an instance frame F is inherited from the all-templates of F. Typically the constraints present at a particular class C consist of those inherited from the supers of C, plus additional constraints defined locally at C. C must be a more specific class than its direct-supers, therefore we expect its value constraints to be more stringent. In this section we discuss the different types of constraints that different FRSs allow.

These constraints are typically used for two purposes by FRS. First, when new values are assigned to a slot the FRS verifies that the values satisfy the constraints — if they are violated then an error is signaled. The second use of these constraints is for classification, which is discussed in Section 9. The KL-ONE family treats constraints as *definitional* in nature: the constraints on the slots of a class C specify necessary and sufficient conditions for what it means to be an instance of that class. That is, if a concept is a predicate, the constraints form the operational definition of that predicate.

5.5.1 Constraints on Individual Slot Values

KL-ONE *value restrictions* allow us to constrain the range of a slot such that its values may only be the names of all-instances of a particular class within the KB. More precisely, value restrictions specify that every value of a slot S in a frame F must name another frame that is an all-instance of a class frame C: $\forall x[x \in F.S \supset x \prec\prec C]$ (henceforth we will omit the quantifier and simply write $F.S \prec\prec C$). For example, we might constrain the range of `Computer.Manufacturer` to be an all-instance of the class `Corporations`.

KL-ONE employs *number restrictions*[12] to specify upper and lower bounds on the number of values that a slot can have at one time. Both value restrictions and number restrictions are found in most FRSs.

KEE defines a fairly complex language of *valueclass specifications* that can be used to specify value restrictions. The syntax of valueclass primitives in the KEE language and their meanings are as follows:

- `(MEMBER.OF class)` or `class` — this specification is equivalent to a KL-ONE value restriction: when applied to slot S of frame F it specifies that $F.S \prec\prec class$.
- `(SUBCLASS.OF class)` — every $F.S$ must be an all-sub of $class$ ($F.S \ll class$)
- `(ONE.OF atom1 .. atomN)` — every $F.S$ must take on one of the explicitly specified values ($F.S \in \{atom1, .., atomN\}$)

[12] Synonym: *cardinality restrictions (KEE).*

- `(NOT.ONE.OF atom1 .. atomN)` — every $F.S$ cannot take on one of the specified values ($F.S \notin \{atom1, .., atomN\}$)
- `KEE-INTERVAL` — every $F.S$ must lie within the interval specified over some ordered type such as the integers ($low < F.S < high$)
- `(MEMBERP fn)` — every $F.S$ must satisfy the predicate defined by the LISP function `fn` ($TRUEP(fn(F.S))$)

The preceding primitives can be combined using the following compositional operators:

- `(NOT.IN vcspec)` — every $F.S$ must not satisfy the valueclass specification `vcspec`
- `(UNION vcspec1 .. vcspecN)` — every $F.S$ must satisfy one of the given valueclass specifications
- `(INTERSECTION vcspec1 .. vcspecN)` — every $F.S$ must satisfy all of the given valueclass specifications

KEE also has cardinality restrictions like those of KL-ONE. CLASSIC has enumerated types like those specified using the KEE `ONE.OF` operator, and requires each enumerated type to be an existing instance frame. FRAMEKIT provides a mechanism similar to KEE's `MEMBERP` operator: value restrictions can be specified as LISP S-expressions that FRAMEKIT evaluates on candidate slot values.

LOOM allows several additional types of constraints. As well as having an analog of the KEE `MEMBER.OF` operator, a LOOM operator can specify that only *some* value of a slot must be an all-instance of a given class (rather than all values). Constraint expressions can also be formed that compare slot values to constant expressions using operators such as "=". LOOM also allows the user to write arbitrary slot constraints using the LOOM query language.

5.5.2 Constraints Between Slot Values

As well as specifying absolute constraints on the value of a single slot, it is often desirable to specify a relationship that must hold between the values of two slots. For example, imagine that we wish to define a class frame that represented a horizontally integrated computer manufacturer. Such a corporation would manufacture the memory chip, the processor chip, and the disk controller used for their computer (we assume that each is described in a separate frame). In all instances `I` of such a class the values of `I.Memory_Chip->Manufacturer` and `I.Processor_Chip->Manufacturer` and `I.Disk_Controller->Manufacturer` must be the same.

The KL-ONE family of FRSs can express such constraints using *role value maps* [15] (called *role constraints* in NIKL [76]). These constraints allow users to specify that either an equality or a subset relationship must hold between the values of two slots. Since the slots

themselves might exist in different frames (as in the preceding chip example), the KL-ONE implementation uses pointers to explicitly identify the two sequences of slot compositions (or *role chains*) that are involved in the relationship.

Role value maps are a special case of a more general constraint mechanism in KL-ONE called *structural descriptions* [15]. Whereas role value maps specify either an equality or a subset relation between the values of two slots, structural descriptions allow the user to specify an arbitrary relation between the values of two slots. This relation itself must be described as another KL-ONE frame — if the user wished to constrain the value of one slot to be less than the value of another slot, the required `less-than` predicate must be described in a KL-ONE frame [15]. The user specifies a structural description by creating KL-ONE links between the predicate frame and the slots (role chains) that the constraint relates. This approach to specifying constraints is fairly clumsy and yields constraints that are very difficult to understand; the LOOM query language can be used to specify similar types of constraints, but is much more readable.

5.6 Discussion

One of the few performance studies in the FRS literature is that by Mitchell et al [57], who studied the effect of their caching mechanism (as well as other learning mechanisms) on the performance of THEO. In one experiment, a series of 300 queries were made to a KB of family relationships, with caching enabled during all queries. Queries early in the series took on the order of 10 seconds, whereas queries later in the series took on the order of .5 seconds, showing that the accumulation of cached information did improve performance. This study would be improved if it presented data for the same series of queries with caching completely disabled, so as to control for the cost of performing caching in the early queries. That is, we wish to know not only that the system performs faster as it caches more information, but that on average it performs faster than if caching is not used. Such data is given in an abbreviated form later in the paper for a different set of experiments.

Both facets and seeUnits (see Section 4.2) allow us to annotate slot values, but their relative merits remain to be determined. Perhaps by embedding annotations within the associated slot structure using facets that we achieve a locality of reference that increases performance. The meta information that they provide yields an increase in FRS expressiveness.

Some of the functionality provided by slotUnits could also be achieved by attaching procedures to a slot, or by storing the information in facets. SlotUnits have the advantage of providing a more succinct representation for commonly required information about slots; attached procedures suffer the additional disadvantage of being less declarative. In addition, when using slotUnits, information about slots is treated as a first-class entity within the FRS in the following respects. We can arrange slotUnits in a generalization hierarchy to use the benefits of inheritance in describing slots. Information about slots is represented globally in slots of the slotUnit, rather than locally (and perhaps redundantly or inconsistently) as facets in every frame that the slot is used in, and therefore a given slot has the same semantics throughout a KB, which increases the understandability and maintainability of

the KB. Finally, we can employ existing subsystems of the FRS (such as its query language) to manipulate information in slotUnits.

Similar comments apply to valueclass specifications since rules or attached procedures could be used to implement equivalent capabilities. Researchers have devised a special language that provides a succinct, declarative means of encoding commonly required classes of constraints. In the KL-ONE family, the declarative semantics of these constraints form the basis for classification. Fox et al note [31] that enforcing valueclass restrictions can slow down a running system significantly (although we are not told *how* significantly), so SRL provides a way of disabling value-restriction checks — they are typically enabled only during system development and debugging.

6 INHERITANCE

Inheritance is an inference mechanism in which beliefs about a frame in a taxonomic hierarchy are acquired from its parents in the hierarchy. This section explores the inheritance mechanisms present in a variety of FRSs to answer such questions as: When does the computation of inheritance occur in different FRSs? Can conflicts arise when a frame inherits information from multiple parents, and if so, how can these conflicts be resolved? What different semantic modes of inheritance exist? And finally, exactly what information is manipulated during inheritance?

6.1 What Information is Inherited?

Generally speaking, when we define a slot S in a class frame C, inheritance causes all children of C to contain the slot S. That is, to users it will appear that every child of C contains a slot named S that has the same datatype and value constraints as does the S in C.[13] The intended semantics of this operation are that if a certain attribute or relation applies to a class of objects, or to a concept, then that attribute or relation must apply to every child of that class or concept, with at least as strict value restrictions as applied to the class or concept.

A fundamental distinction between FRSs is whether or not child frames acquire slot *values* during inheritance. Systems in the the UNITS family, and the SRL family, do allow the inheritance of slot values. The semantics of this operation is that when a slot value is defined in a class frame, we would like the FRS to assume by *default* that the same value holds in a child of that class, unless the user explicitly stores a *local value* in the slot of the child. For example, when we define `32` as the value of the `Word_Size` slot in the `VAX_Family` frame, we wish that value to be the default value of this slot in all children of `VAX_Family`. This inference is a form of default reasoning.

Another variation is that some FRSs allow inheritance of properties across other types of

[13] An exception is own slots, which are never inherited by child frames (see Section 5.3).

relations than the instance or super–sub relations. For example, in OZONE, PARMENIDES, CYCL, HYPERCLASS, FRAMEKIT, and SRL, a user could define inheritance across a *part* relation so that the parts of an object would inherit information from the containing object. SRL and CYCL users employ a slotUnit to describe how inheritance is computed for each relation. The slotUnit specifies what slots are inherited across that relation (for example, the `owned-by` slot might be inherited across the `part` relation since we expect the owner of an object to also own all of the parts of that object). In SRL this mechanism also allows users to specify a mapping of slot values across a relation, for example, if the relation `previous-activity` associates frame `activity1` with frame `activity2`, we could specify that the value of `activity1.finish_time` is mapped to the value of `activity2.start_time` [31].

The JOSIE ability to treat slots as classes allows other types of inheritance relationships [60]. This facility would allow us to assert that Digital manufactures all members of the Vax family — that the values of a given slot *Digital.Manufactures* include all members of a class `Vax_Family`. It allows us to assert that the values of a given slot in a given frame are a subset of the values of some other slot in some other frame. And we can assert that all of the computers manufactured by Digital are manufactured in the United States `Digital.Manufactures.Location=USA`.

Inheritance in most of the KL-ONE family of FRSs does not include slot values because the notion of a default conflicts with the definitional view of concepts in the KL-ONE family (see Section 9.2). Thus, systems such as KL-ONE, NIKL, and KRYPTON cannot define default values. The JOSIE system distinguishes necessary defaults — those for which no exceptions exist — from defaults that can be overridden.

6.2 Semantic Modes of Inheritance

The semantics of some slots requires a different interaction between default values and local values than that described in the previous section, where local information overrides inherited information. Therefore, some FRSs allow the user to specify one of several different *modes* of inheritance for slots. The KL-ONE and SRL families do not provide multiple types of inheritance, whereas the UNIT Package, KEE, and PROTEUS do — and each defines a somewhat different set of inheritance types. In KEE users define a slot inheritance mode by setting the value of a special facet that is defined for each slot. KEE supports the following inheritance modes to determine the observable value of a slot S in a child frame C ($C.S$) given the value of S in a parent frame P ($P.S$), and the value of S that is stored locally in C ($C.S'$):

- `OVERRIDE.VALUES` — Local values override inherited values so that:
  $C.S = \{if\ (C.S' == NULL)\ then\ P.S\ else\ C.S'\}$
- `UNION` — Local values are unioned with inherited values:
  $C.S = P.S \cup\ C.S'$
- `RUNION` — Same as UNION but the order of the slot values is reversed.

- `SAME.VALUES` — No differing local values allowed: $C.S = P.S$
- `UNIQUE.VALUES` — Inheritance is blocked completely: $C.S = C.S'$
- `MINIMUM` — Minimum of local and parent values (a `MAXIMUM` mode exists also): $C.S = min(P.S, C.S')$
- `METHOD` — Used for attached procedures
- `UNION.EACH.VALUE` — Analogous to UNION except that the slot value must be a list, and individual elements of the lists are unioned
- `VCSIMPLIFY` — Used to simplify inherited slot constraints

`OVERRIDE.VALUES` is the mode found in most FRSs, but the other modes do have utility. For example, bookeepingSlots (see Section 5.3) should use the `UNIQUE.VALUES` mode since information about a frame (such as the name of its creator) should not necessarily be inherited by the children of that frame.

6.3 Conflicts in Inheritance

Early systems such as the UNIT Package allowed a frame to have only a single parent in the inheritance hierarchy. Later systems such as KEE, CYCL, SRL, and the KL-ONE family allow a node to have multiple parents, and therefore to inherit information from more than one parent. The KL-ONE family does not allow multiple parents to provide conflicting information to their children — this situation would be declared an inconsistency by the classifier (see Section 9). Most FRSs that do allow inheritance of conflicting attributes provide different mechanisms to resolve potential conflicts. FROBS allows the user to explicitly specify which parent the value should be inherited from, thus specifying the "context" from which the value should be taken. By default the STROBE inheritance mechanism uses the first value that it finds in a breadth-first search of the child frame's ancestors, but the user can specify a depth-first search on a per-slot basis. FRAMEKIT allows the user to specify best–first or depth–first search, or an exhaustive search in which the final inherited value is the union of all values encountered during the search. OZONE allows user-defined search functions.

Conflicts in inheritance become an even greater problem when a FRS computes inheritance over multiple relations (such as the `part` relation discussed in Section 6.1). Now conflicts can occur over multiple relations, in addition to the conflicts from multiple parents over the super–sub relation. SRL users can specify the order in which links are to be searched from a given frame.

6.4 Time of Inheritance

Different FRSs compute slot inheritance at different times. In systems that perform *fetch-time* inheritance, inheritance occurs at the time an application program requests the value

of a slot — the inheritance system climbs the inheritance hierarchy searching for a value to inherit (used by the UNIT Package and KEE, for example). The inherited value is then returned to the application, but is never physically stored at the child frame. Conversely, in systems that perform *assertion-time* inheritance,[14] when a parent frame is defined or altered or classified, inherited slot information is physically copied to all child frames (also used by the KL-ONE family). In many systems that compute inheritance at assertion time (such as NIKL), if a user wishes to change a slot definition in a parent frame, the user must reload the entire KB because the system is unable to incrementally update the cached information.

THEO combines these two approaches by allowing users to specify that inherited slot information should be cached in child frames at fetch time (RLL and CLASS provided a similar caching mechanism). THEO also records justifications that describe the dependencies between the cached value and the slots it was derived from, so that cached values can be removed in response to changes in the values that they depend on. PARMENIDES users can specify for every slot whether inheritance is to occur at fetch time or at assertion time. When a user changes the definition of a PARMENIDES slot that is inherited at assertion time, the system immediately propagates the new slot definition to all children of that frame — but information defined locally at a child frame is not overwritten.

6.5 Discussion

Assertion-time and fetch-time inheritance have different speed and space requirements, which are not understood, and they also have different run-time flexibility. Schoen made an initial step towards understanding this tradeoff by measuring that in a system that performs fetch-time inheritance, each additional direct-super encountered during an inheritance search slowed down inheritance by an additional 17%, assuming 8 slots per class frame [81, p203]. By performing inheritance at assertion time, the authors of NIKL eliminate the need for fetch-time searches up the inheritance hierarchy. But they force the user to reload the entire KB when a class definition changes, thus decreasing the run-time flexibility of these definitions, and providing an approach that probably will not scale up to very large KBs (neither of which disadvantages are noted in a recent assessment of NIKL [78]). Since Schmolze and Mark have stated that a major goal of implementing NIKL was to improve upon the speed of KL-ONE [78], we would expect to find a thorough evaluation of exactly what improvements were made to NIKL, and of the speedup obtained through each improvement. Instead, this paper is an example of vague, imprecise analysis of an implemented FRS. The most detailed timing measurement we are given is: "Overall, the NIKL system was an order of magnitude faster than KL-ONE." [78, p9]. What contributed to this speedup? "This decision to trade off more space for less time has borne out well, given the current economics of computation and the needs of most applications" [78, p9]. We should like to know *exactly how* space was traded for time, and we should be given rigorous experimental data that convinces us that part of the speedup was not due to the use of faster hardware. Another secret of the performance improvement is explained as "hashing and other fast schemes were used when possible" [78, p9] — in other words, we

[14] Called *propagation* in PARMENIDES.

have no idea what the "other fast schemes" are, or exactly how any of these schemes were implemented.

PARMENIDES also performs inheritance at assertion time, but it recomputes inheritance when new assertions are made about classes — providing more flexibility but requiring a more complex implementation. We are not told how complex the implementation is.

Similar to slotUnits and valueclass restrictions (see Section 5.6), multiple inheritance modes could also be implemented with attached procedures, but a special language provides a succinct, declarative means of encoding common operations. We lack information about how useful multiple inheritance modes are in practice, or the costs of implementing them. They would surely complicate the computation of subsumption were they introduced into the KL-ONE family.

Similarly, inheritance across multiple link types could be implemented through rules or attached procedures, but may have benefits of succinctness and declarativeness. The ability to explicitly establish an ordering for link searches in SRL is similar to a metarule facility, thus providing an additional type of expressibility. However, rules are such an obvious competitor to this approach that a more detailed analysis is needed. Is it preferable to use inheritance for inference over instance and super–sub relations, and rules for other types of inference, or is inheritance preferable for all these forms of inference?

Although a production-rule facility could provide many of the same inferences as described in this section (more precisely, a default-rule facility), another key reason to prefer the inheritance mechanisms described in this section is that they optimize the performance of this special type of inference. Using rules to implement inheritance would require backward-chaining searches through a large rule base to compute inherited values — unless some type of compilation were used. The super–sub links in a taxonomic hierarchy can be viewed as a way of compiling rules into chains that support a particular set of inferences. Systems that compute inheritance at run time still search these rule chains, whereas systems that compute inheritance at assertion time have even compiled out the searches along rule chains.

The publications that described alternative ways of resolving conflicting inherited information (such as breadth-first, depth-first, and exhaustive search through the ancestors of a frame) do not report how acceptable the inferences produced by these methods are in practice, nor the computational costs of these methods.

A number of researchers have investigated theoretical aspects of inheritance in network-based KR systems [97, 96, 83, 74]. Although these researchers have obtained interesting results, their utility with respect to the FRSs discussed in this paper is unclear because their research was in the context of toy problems and has not been proven to generalize to real-world problems of the sort addressed by the FRSs discussed herein. In addition, their basic model of inheritance intimately involves a type of relation called `IS-NOT-A` that is not used by any of the FRSs discussed in this paper; it is not clear how easy it will be for either model to accommodate the other.

7 PERSISTENT KNOWLEDGE BASES

If FRSs are to be employed to build and manage large knowledge repositories, they must provide sophisticated facilities for transferring frame knowledge bases between virtual memory — where all processing of frame information takes place — and persistent secondary storage. Such facilities should use modern database techniques such as transactions to protect data from corruption by events such as operating system crashes or hardware failures. Currently, the standard FRS approach to saving KBs in persistent form is for the user to explicitly execute an operation that saves an entire KB to disk storage. This approach does not scale as KB size increases because the time required for the save operation is proportional to the size of the KB, not to the number of updates made since the KB was saved last. The standard approach to moving frame data into virtual memory is to load an entire KB into memory before processing begins, which again takes time proportional to the size of the KB rather than to the amount of information that will be accessed. These simplistic approaches to loading and saving frame KBs will become less and less palatable as KBs increase in size. It is not the idea of keeping substantial amounts of frame data in virtual memory that is antiquated — in fact the trends in the database community toward main-memory databases support this basic model. Rather, FRSs must be more selective about what data is transferred between virtual memory and persistent storage to these transfers faster. This section reviews a number of approaches to the storage and retrieval of persistent frame KBs.

THEO and CLASS provided an early mechanism for managing large KBs. Frames in a single KB are partitioned into separate files that can be saved or loaded independently; the system automatically tracks which frames belong in which files. LOOM provides a similar facility: it can automatically save all classes, instances, relations, or methods within a KB to separate files. THEO and CLASS manage only a single frame name space. In contrast, KEE, STROBE, ARLO, and LOOM provide a multiple KB facility. In a single session, users of these systems can load one or more KBs into virtual memory, where each KB has a separate frame name space, and can be saved to secondary storage (or deleted) independently of the other KBs. Frames in different KBs may reference one another, so that frame F_1 in KB_1 could be a child of frame F_2 in KB_2.

In SB-ONE, ARLO, and K-REP, KBs themselves are linked in a hierarchy; all of the frames defined in a knowledge base K also appear to the user to be defined in all children of K. The authors of K-REP describe this facility as analogous to the COMMON LISP package system.

The UNIT Package and RLL employed variants of an early frame storage management scheme that provided demand paging of frames [89, p69][87]. These techniques were developed to allow KB size to exceed virtual-memory size on machines of 1970s vintage that had limited (18 bit) address spaces. Frames are read into virtual memory when first accessed, and are saved back to disk on a least-recently used basis during LISP garbage collections. One limitation of this approach is that the data stored on secondary storage is not protected from corruption. OZONE used similar techniques, but its slot spaces were used to distinguish slots that were read on demand from system slots that were always memory resident (such

as slots containing parent-link information).

A recent paper by Mays et al [53] describes storage management facilities for K-REP that are a leap beyond the preceding capabilities. They interfaced K-REP to an object-oriented DBMS to support a versioned KB that can be updated by multiple users via transactions. In addition, Ballou et al interfaced PROTEUS to the ORION object-oriented DBMS [5], Abarbanel and Williams coupled KEE to a relational DBMS [1] to produce KEE Connection, and Peltason et al interfaced BACK to a relational DBMS [67]. McKay et al have developed an Intelligent Database Interface (IDI) [55] with essentially the same architecture as KEE Connection, although they have improved upon a number of the details of the AI/database coupling, for example, the IDI includes an intelligent cache of database information, and it can automatically obtain schema information from the database.

A final point of variability concerns the form in which KBs are stored persistently. Most KL-ONE descendants store KBs as a set of LISP forms (expressions) that, when executed, recreate the KB in virtual memory. One such form would both define a new concept, and then reclassify it into the KL-ONE hierarchy. Thus, all existing members of the KL-ONE family reclassify every concept every time a KB is loaded — the results of classification are not saved persistently. In contrast, most other systems store KBs as data files that contain a more direct representation of the virtual-memory form of the KB, and that are not processed by the LISP evaluator during loading.

7.1 Discussion

A multiple KB facility is invaluable for users and applications that access many different KBs over the long term, but who access few KBs on a day-to-day basis, because it allows them to load only the KBs that they require at a given time. Users also benefit who update only a subset of the KBs that they have loaded, and therefore need only save a subset of the frames that they have loaded. Multiple name spaces are useful for operations such as simultaneously processing both a KB and a copy of that KB, which will necessarily contain many frames with the same name. COMMON LISP packages obviate the need for a multiple KB facility to a large degree since packages provide separate name spaces. But package systems do not provide a way of renaming all symbols to another package, as is required for a KB rename operation. Packages also negate a motivation of OZONE spaces, namely providing separate name spaces for user and system slots.

The paper by Mays et al is notable in providing the best, most precise description of implementation techniques in the FRS literature. Otherwise we have little information about the implementation techniques employed for the capabilities in this section.

We have essentially no knowledge of the performance of any of the techniques discussed in this section, such as the cost of providing inheritance among KBs themselves, or the performance of the storage management facilities. An exception is the work by McKay et al, which does give some performance measurements. Since performance is a key property of a storage system, it is virtually impossible to evaluate the relative merits of these alternatives (e.g., Ballou et al versus Mays et al) without comprehensive performance data. Even with

such data, a comparison will be difficult since, for example, the Ballou and Mays groups utilized different underlying DBMSs — STATICE and ORION — and therefore it will be difficult to determine how much of the performance depends on the DBMS itself.

8 OBJECT-ORIENTED AND ACCESS-ORIENTED PROGRAMMING

Some FRSs contain an object-oriented programming (OOP) facility [33], or an access-oriented programming (AOP) facility [95], or both. As noted in [95], these two programming paradigms are duals of one another.

8.1 Object-Oriented Programming

The UNIT Package provided support for OOP that was developed further in KEE and STROBE. A user of the UNIT Package could attach a LISP function (or *method*) to either a frame or a slot. Users could invoke a method by *sending a message* to that frame or slot. For example, we would attach a method called `If-Deleted` to a frame by creating a slot within that frame called `If-Deleted`, and storing the definition of a LISP function in that slot. We could send a message to that method by calling a UNIT Package function called `UNITMSG` whose parameters include the name of the slot and the name of the method. Methods are associated with slots in a similar fashion (they are stored in a facet of the desired slot). Methods are inherited along the generalization hierarchy.

The OOP facilities of STROBE and KEE differ in several respects. Like the UNIT Package, STROBE searches up the generalization hierarchy for a method to execute when a message is sent. But in addition to executing the first method found in this search, the user can specify that STROBE execute *all* of the methods found in the ancestors of the frame to which the message was sent — in either parent–child or child–parent order. In this way frames can acquire specialized behaviors in addition to the general behaviors they inherit from their ancestors. In addition, if no method definition is found via inheritance, STROBE will attempt to perform *datatype rerouting*: the message is sent to the frame that describes the datatype of the slot to which the original message was sent. Thus the user can describe how all slots of a given datatype should respond to a particular type of message.

KEE provides a sophisticated mechanism whereby methods can by modified by inheritance such that methods defined in frames high in the hierarchy can be altered by the children of those frames. Specifically, every method consists of four sections called *main code, before code, after code,* and *wrapper code.* The before code and after code sections of a method define LISP code that is executed before and after the main code section, respectively, and which is inherited using union inheritance (see Section 6.2). The before code present in frame F is the concatenation of the before code defined in F and the before code defined in the parents of F. Main code is inherited using override inheritance, so that locally-defined

main code overrides parental main code. The wrapper code section defines code that is wrapped around the concatenation of a frame's before code, main code, and after code.

8.2 Access-Oriented Programming

Access-oriented programming is more prevalent in FRSs than is OOP, and is used in systems such as the UNITS family, KL-ONE, LOOM, the SRL family, FROBS, and OZONE. An AOP facility allows the programmer to associate programmatic annotations with data, such that the annotations are automatically executed when different classes of operations are performed on the data (in a FRS, these data are slot values). The annotations can be dynamically attached to and removed from the data, and the existence of the annotations is transparent to programs that are not explicitly attempting to manipulate the annotations (i.e., to programs that are only attempting to manipulate the data). Usually the annotations are LISP procedures, but in THEO, for example, annotations can be PROLOG rules.

One common type of annotation function is invoked when a user requests the value of the slot; the function computes the value of the slot and returns the value to the user in a transparent fashion — as if the value were stored in that slot. Another common type of annotation function is invoked when a user modifies the value of a slot. The annotation function might update other slots, or perhaps external databases, whose values depend on the modified slot.

In FRAMEKIT for example, a user can annotate a slot by storing LISP functions in facets called `If-Needed`, `If-Accessed`, `If-Added`, and `If-Erased`. The `If-Needed` function will be invoked if a user attempts to get the value of the associated slot, and if that slot currently has no value; if the slot does have a value then the `If-Accessed` function (if any) will be invoked instead. The `If-Added` function will be invoked when the user adds a new value to a slot, and the `If-Erased` function will be called when the user erases the value of a slot. PARMENIDES provides for additional annotations called `Pre-If-Set` and `Post-If-Set` that are invoked before and after a new value is added to a slot, respectively. SRL — the ancestor of both of these systems — has a more general mechanism. An SRL annotation is itself described by a frame that specifies such things as: by what type of slot operation the annotation is to be invoked by (e.g., a get or a put operation); whether the annotation should be invoked before or after that slot operation is performed; and what "effect" the annotation has — does it alter the value returned by the slot operation, does it have a side effect that does not alter the returned value, or should the slot operation be completely blocked from occurring?

Other FRSs allow similar types of annotations: STROBE allows annotations that are executed upon creation and deletion of frames and slots, and before and after modification of and access to slot values. KL-ONE allows attached procedures to be invoked before and after the KB operations Individuate, Restrict, Create, and Remove. FRSs that do not support AOP include NIKL, KRYPTON, and PROTEUS.

8.3 Discussion

The authors of STROBE and SRL note that the use of AOP can impose a fairly heavy computational cost [88, 31]. However, no publication contains actual measurements of the performance cost. Both systems allow users to disable this facility [88, 31]. And in STROBE, the mechanism is usually disabled [79]. The authors of STROBE and of SRL state that the major component of the cost is performing inheritance searches to check for the existence of annotations whenever a slot value is accessed — this search must be performed even for slots that have no annotations in case inherited annotations exist. Smith and Carando suggest an optimization to avoid repeating these searches when the same slot is accessed multiple times, which is to cache a null value for the annotations locally within a slot when a search yields no annotations. This approach is used by THEO. Another approach would be for the FRS to automatically record in the slotUnit for slot S whether or not S is ever annotated *anywhere* in the KB; for slots that are unannotated, this global information could provide significant savings. We have no other knowledge about implementation techniques for OOP or AOP, nor about the frequency with which these capabilities are needed in applications.

9 CLASSIFICATION

9.1 Overview

Intuitively, one class (or *concept*) definition A *subsumes* a second class definition B if the concept that A represents is more general than is the concept that B represents. For example, the class `man` subsumes the class `father` since all fathers are men. We can describe subsumption more precisely in set-theoretic terms. The *extension* of a concept is the set of instances of that concept that exist in the world. A subsumes B if the extension of A (written $|A|$) is a proper superset of the extension of B:

$$A > B \equiv |A| \supset |B|$$

For example, the set of all fathers is a proper subset of the set of all men. Members of the KL-ONE family automatically compute whether one class frame subsumes another, based on the definitions of the slots that comprise those classes. The computation of subsumption is the basis of *classification.* The operation positions a frame A into its proper position in the inheritance hierarchy. Positioning A as a direct-super of B is proper if and only if A is the most specific subsumer of B, that is, if there exists no concept definition C such that A subsumes C and C subsumes B. A *realizer* (also called a *recognizer*) positions an instance frame in the hierarchy. The realizer finds the most specific concept A in the KB such that the instance is in the extension of A.

The definition of a concept C consists of two parts: a list of concepts that are more general than C, and a list of conditions that differentiate C from those ancestor concepts. These conditions are a variant of the slot value restrictions discussed in Section 5.5. For example,

in KL-ONE we would define a `father` as an all-sub of `man` that has at least one value in the `child` slot. At first glance classification may seem redundant since every class definition already contains information about the placement of the concept in the inheritance hierarchy — the definition of `father` explicitly notes that it is an all-sub of `man`. Classification is needed because these definitions may not be precise (or even consistent) in a global sense. For example, the definition of `father` tells us that `father` is more specific than `man`, and therefore that `father` belongs below `man` in the taxonomy. But the definition does not explicitly tell us that `man` is the most specific subsumer of `father`, because `father` might be subsumed by other concepts that have been defined relative to `man`. As another example, the class `Husband` might be defined as an all-sub of `Married_Person`, whose `Wife` slot is restricted to store values of type `Woman`. `Husband` may not belong directly below `Married_Person` in the hierarchy. A KB describing royalty might define a concept called `Royal_Husband` to take into account the fact that royal children sometimes marry. `Royal_Husband` is defined as an all-sub of `Married_Person` whose `Wife` slot is restricted to take values of type `Female` (an ancestor of `Woman`). In this KB, a classifier would link `Husband` as a direct-sub of `Royal_Husband` rather than of `Married_Person`.

In a sense the original specification that `Husband` is an all-sub of `Married_Person` is treated as advice that should be refined by the classifier. FRSs outside the KL-ONE family assume that all subsumption relationships are given by the user at class definition time, and therefore that no additional relationships remain to be discovered.

The basis of classification lies in comparing the intensional concept definitions that give necessary and sufficient conditions for membership in a class. That is, the KL-ONE family of FRSs interpret the slot value restrictions for a given class C as defining necessary and sufficient conditions for recognizing an instance of C. We infer that `Royal_Husband` subsumes `Husband` by comparing the definitions of their `Wife` slots. In practice, this precise definitional view of concepts breaks down when we attempt to define necessary and sufficient conditions for *natural kinds* such as the concept of a fish. We cannot list necessary and sufficient conditions for what it means to be a fish because the concept is so complex, and so imprecise. Therefore, the KL-ONE family allows users to define a concept as *primitive* if it is impossible to specify necessary and sufficient conditions for membership in that class — the definition of a primitive concept specifies necessary conditions only. Primitive status tells a FRS that the definition of the class cannot be expressed within the existing language of definitions, and therefore that the subsumption relation cannot be computed between two primitive concepts. However, classifiers can compute the subsumption relation between two concepts that are defined as all-subs of the *same* primitive concept (e.g., between two nonprimitive all-subs of `Fish`). See [14, p420] for a more detailed discussion of primitive versus defined concepts.

FRSs that employ classification are often called *terminological reasoners* because they maintain relationships between a set of *terms* — class or concept definitions. The KL-ONE-style emphasis on term definition — on the manipulation of *structured descriptions* — led to a system architecture that separates the subsystem that manipulates terms, from the subsystem that manipulates assertions with respect to those terms. This distinction arose in KRYPTON and is maintained in KRYPTON's successors. The TBox (terminologic box)

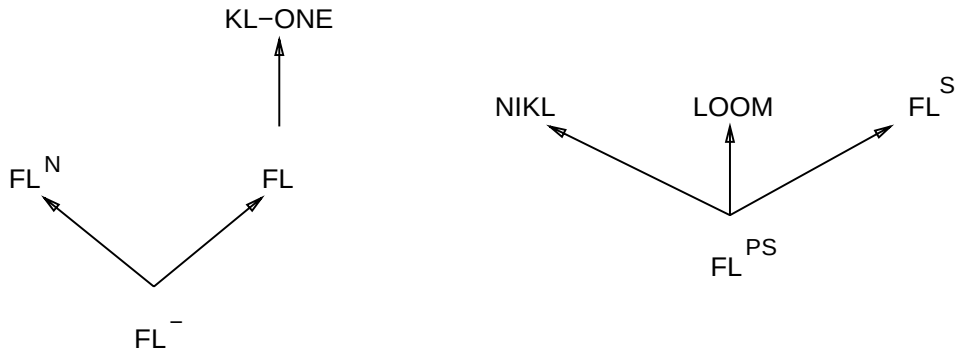

Figure 2: The hierarchy of expressiveness and complexity relationships that have been proven among different KR languages. Languages lower in the figure are less expressive and allow less expensive computation of subsumption.

subsystem is responsible for computing subsumption and classification, for computing inheritance of slot value restrictions, and for managing the hierarchy of class definitions. The ABox (assertion box) subsystem is responsible for managing instances: for processing assertions about instances, for performing realization (recognizing when an instance instantiates some concept), and for performing query processing.

A more expressive representation language allows a user to define more complex restrictions on slot values, and thus to define more complex relationships among concepts. The complexity of computing whether concept A subsumes concept B is a function of the size of the definitions of A and B, and of the sizes of concepts referenced by the definitions of A and B. More precisely, a number of theoretical results have been obtained concerning the computational complexity of subsumption. Levesque and Brachman obtained the first such results [50] by proving the complexity of (computing subsumption in) two related concept definition languages, called $\mathcal{FL}$ and $\mathcal{FL}^-$, that differed by a single terminological construct. See Figure 2. The complexity for the simpler language, $\mathcal{FL}^-$, was $O(n^2)$, whereas the cost for $\mathcal{FL}$ was co-NP hard. This result was startling because a small change in the representation language yielded a large change in complexity. $\mathcal{FL}$ is a subset of the language used in KL-ONE, which language therefore must be at least co-NP hard. The KL-ONE classifier [77] ran in polynomial time because it was incomplete — it overlooked certain subsumption inferences.

Nebel proved that the complexity of a language $\mathcal{FL}^{\mathcal{N}}$, comprising a subset of the BACK language, is also co-NP hard [61]. $\mathcal{FL}^{\mathcal{N}}$ is a different extension of Levesque and Brachman's language $\mathcal{FL}^-$. Schild proved that computing subsumption is undecidable in a very expressive terminological language [75] that I call $\mathcal{FL}^{\mathcal{S}}$. Patel-Schneider considered a less expressive language ($\mathcal{FL}^{\mathcal{PS}}$) than did Schild — a language that in turn is a subset of the languages of NIKL and LOOM — and he proved that language to be undecidable [66]. Nebel later reexamined the $\mathcal{FL}$ language, and proved that the cost of computing subsumption in the *context* of a larger base of terminology, rather than for two isolated definitions (as in all previous results considered), is co-NP complete. This last result takes into account the costs of traversing super–sub links in the hierarchy.

The principle that these results yield is that the more expressive a representation language is, the higher is the cost of computing subsumption in that language. Because of the central role subsumption plays in the KL-ONE family (it is an intimate part of the definition of new concepts and of problem solving), these results have animated researchers in the KL-ONE community to seek alternative ways of handling this tradeoff between expressive power and computational cost. They have concluded that the solution requires some combination of less expressive representation languages, and subsumption algorithms that are either incomplete or unsound or both, to yield faster classification.

9.2 Discussion

Although many researchers view classification as the central operation in FRSs, the role of the classification operation is far from clear. Classification plays a central role in the KL-ONE family of FRSs, and KR researchers have devoted a vast amount of effort to understanding the theoretical properties of the classification operation. Many KR researchers have come to believe that classification must necessarily play a central role in every FRS. For example, Schmolze and Mark write, "The contribution of KL-ONE was to recognize that the *ad hoc* manner in which frames could be related to each other made it extremely difficult (1) to build large knowledge bases (because the relationships soon became incomprehensible to human knowledge base builders), and (2) to reliably characterize the competence of the system's reasoning mechanisms (because the interpretation of relationships usually depended on procedures written by the application programmers). KL-ONE's solution was to make the organization of these frames ... the responsibility of the *system*, not the programmer." [78, p5] (which responsibility is implemented through classification).

But in fact, no systematic study validates the claim that classification decreases the time required to engineer or maintain large knowledge bases. Furthermore, I estimate that the majority of industrial strength FRS applications have been constructed from FRSs that do not classify. All of the commercialized FRSs are descendants of the UNIT Package family, the FRL family, or the SRL family, and none of these families employ classification. Therefore, it is plausible to believe that FRSs that do not classify have seen more widespread use in real-world applications than have FRSs that do classify. In addition, Lenat and his colleagues are successfully using the CYCL FRS, a member of the UNIT Package family, to construct what is probably the largest knowledge base ever built [48]. I do not say that a FRS that classifies could not be successfully commercialized, but simply that none have been, and that we should therefore question the practical utility of systems within the classification paradigm. The empirical success of FRSs that do not classify is compelling evidence that classification is neither a necessary nor a sufficient quality of a successful FRS. Therefore, KR researchers must sharpen and justify their hypothesis about the degree to which classification is required in a FRS. Rather than viewing classification as the focal point of all KR research, it is merely one of many FRS capabilities whose tradeoffs must be understood.

Virtually all of the theoretical results in KR research have been achieved within the classification paradigm, whereas more practical results have been obtained outside the classifica-

tion paradigm. We might ask how relevant the theoretical results within the classification paradigm are to the design of FRSs that do not classify. This section examines the role of classification in FRSs from a broad perspective to assess what is known about this operation and what is not, and to ask how these two paradigms of knowledge representation might be combined.

9.2.1 Performance

Section 9.1 summarized a substantial body of work on the *theoretical worse-case* cost of computing subsumption. However, despite years and years of theoretical work, the literature contains only a single empirical study of classification [38]. Furthermore, no one has asked for which (if any) of the tasks in which it is employed, classification is too slow. Is it a limiting factor in KB housekeeping, or query answering, or problem solving? Each of these tasks might have different performance requirements, and different strategies might be used to overcome the limits imposed by using classification in these tasks.

For example, if the main problem is in KB housekeeping, are FRSs too slow during KB loading, or during interactive creation of single classes, or during instance realization? Consider that when current FRS implementations load a KB containing N frames, that they perform N classification operations. Therefore, the speed of KB loading could be greatly increased if KB-save operations saved all computed super–sub relationships, eliminating the need for classification during KB loading. This approach may require the development of algorithms that reclassify a concept whose definition has changed. A number of FRSs assume that changes to concept definitions are made directly in the KB data file using a text editor, precisely because that file does not record subsumption relationships. The entire KB must then be reloaded. This approach is extremely awkward for large KBs with many changing concept definitions.

Consider also that for the task of KB housekeeping, classification need not be an interactive operation, but could proceed in the background as the user worked on other tasks (Doyle and Patil also make this point [22]). To guarantee sound inference, all background classification operations must complete before the first query or problem-solving operation is invoked. Doyle and Patil also suggest that this condition could be relaxed — that in some cases it may be acceptable for problem solving to begin before subsumption relationships between all pairs of classes in the KB are known. Furthermore, consider that some applications employ classification for KB housekeeping *only*, because they perform specialized forms of inference, and therefore have very informal requirements as to when KB housekeeping must complete.

I next argue that the significance of the theoretical results concerning the complexity of classification have been seriously overstated by authors who assert that classification is intractable. I refer to the results that for a concept-description language L, the cost of computing subsumption between two concepts defined in L is co-NP hard. I do not refer to languages for which subsumption is undecidable. Subsumption for a language L is co-NP hard for the concepts C_1 and C_2 that are to be compared, *with respect to the size of the*

definitions of C_1 and C_2. Let us call those sizes N_1 and N_2. I argue that N_1 and N_2 will not generally increase without bound as applications increase in complexity. As applications scale up I posit that KBs will contain *more* concepts, but that the complexity of individual class definitions will remain about the same as it is today. Therefore, for languages that are decidable, we can probably view classification as a constant time computation with respect to application size because the complexity of computing subsumption is independent of the number of concepts in a KB, N_C.

However, Nebel showed that the complexity of computing classification (as opposed to subsumption) is co-NP complete with respect to N_C [62]. As Nebel notes, however, that result is dependent on the assumption that the depth of the taxonomic hierarchy is not small (e.g., logarithmic) with respect to N_C — an assumption that I expect would be violated in practice for virtually all applications. In my experience, the depth of the concept hierarchy tends to remain essentially constant as large applications become even larger, perhaps because once a KB designer arrives at an adequate conceptualization of the domain, they simply integrate new classes and instances into that existing conceptualization (class hierarchy). My claims are of course unproven, and should be treated as hypotheses to be investigated empirically. Woods makes related arguments in [100, p86].

The recent study by Heinsohn et al [38, 39] validates these claims to a degree by showing that the cost of classification in real-world KBs is approximately quadratic. They performed a series of experiments involving six members of the KL-ONE family (BACK, CLASSIC, KRIS, LOOM, MESON, and SB-ONE) in which they measured the time required by each FRS to classify six real-world KBs developed in different applications at different institutions, as well as a series of larger, randomly generated KBs that had similar characteristics to the real-world KBs (e.g., average number of roles and direct–supers per class). Their conclusion was that classification time for these KBs was quadratic in the number of concepts. They were able to demonstrate exponential classification times for small, hand-generated KBs that were expressly designed for worst-case behavior.

The Heinsohn et al study also provides hard evidence for the importance of elucidating engineering principles and implementation techniques for FRSs. The variation in the time required by different FRSs to classify the same KBs in Heinsohn et al's experiments varied tremendously. In one experiment, CLASSIC outperformed KRIS by a factor of 800. Across a range of KBs, the relative speeds of the different FRSs was virtually unchanged. Thus, some implementations clearly utilized more advanced methods than others, but no publications describe these techniques in detail.

9.2.2 How is Classification Used?

We have already noted that classification is used for at least three different tasks in FRS applications: KB housekeeping, query answering, and problem solving. What we do not know is the frequency with which these different operations are used within different types of applications. Are all of these operations central to the workings of every application, or are some used only peripherally? For example, natural language applications probably use

KB housekeeping and query answering most often; medical diagnosis applications probably use KB housekeeping and problem solving; whereas a qualitative reasoner might use KB housekeeping only.

Doyle and Patil state that in a major biomedical KB constructed using NIKL, two thirds of the concepts in this KB were primitive concepts, and therefore unclassifiable. They conclude that "KL-ONE-style languages significantly restrict their languages in order to speed up an operation applicable to only a small fraction of concepts" [21, p25]. Speed aside, we must ask whether the framework of intensional concept definitions based on necessary and sufficient conditions is useful in application domains that are dominated by natural-kind concepts, and what fraction of domains have this property.

We must also question the current use of classification in problem solving. First, many classification tasks are heuristic in nature [16] — often we can only provide approximate procedures for nondefinitional classification of entities. Indeed, a good many expert systems do just this. Furthermore, experience with expert systems suggests that if classification is to be used for general and potentially intractable problem solving, it is not wise to build classifiers that are atomic, opaque bodies of code. Users are likely to want to bring heuristic domain knowledge, control knowledge, and preferences to bear on classification (to increase its efficiency, for example), which is impossible using current classifiers. Doyle and Patil make a similar point when they suggest that we should relax the requirement that every invocation of a classifier should provide sound and complete classification of a concept with respect to every other concept in the KB. Rather, the classifier should rationally balance its expenditure of computational resources against the preferences of the user and the requirements of the problem. Furthermore, users who seek expert classifications often require an explanation of the problem-solving process, which existing classifiers cannot produce.

9.2.3 Combining the Paradigms

FRSs that do and do not classify embody two different paradigms of KR that are practiced by different communities of KR researchers. Some researchers who work within the classification paradigm feel that FRSs of the other paradigm are "merely programming systems" — that they lack formal (or informal) semantics, that their taxonomic hierarchies suffer from a lack of discipline, and that the programming mechanisms supported by these systems (such as multiple inheritance, defaults that can be canceled, and access-oriented programming) are so complex that they render KBs and associated problem-solving programs hopelessly complex and hard to understand. Conversely, some researchers outside the classification paradigm feel that FRSs that classify are *not* programming systems — that the representational restrictions required to permit a derivation of formal semantics constrain these systems so much as to make them useless for complex applications.

One practical principle of classification is that FRSs that perform classification differ from FRSs that do not in systematic ways due to constraints that the computation of subsumption imposes on a FRS. One way to view this principle is that, just as we can increase

the speed of subsumption by decreasing the expressiveness of the language, providing the capability to compute subsumption at all also requires language restrictions.[15] Classifying FRSs do not provide inheritable default values that can be canceled because the notion of cancellation conflicts with the notion of necessary conditions. That is, classification implies the comparison of definitions, and definitions are true by necessity, not by default [10]. The notion of definition also implies that a child frame cannot inherit conflicting information from its parents since conflicting definitions would indicate an inconsistency in a KB. FRSs that classify do not provide metaclasses, seeUnits, facets, nor own slots and member slots. Classifying FRSs do not provide multiple inheritance modes (see Section 6) because each additional mode complicates the computation of subsumption. Classifying FRSs do not provide slot data types other than the set, such as the list, because it is not clear how to compute subsumption between two lists. Classifying FRSs typically provide neither access-oriented nor object-oriented programming because it is impossible to compute subsumption between two LISP programs (in classifying FRSs that do provide AOP/OOP, use of these features renders concepts unclassifiable).

I assert that because FRSs outside the classification paradigm have had such significant empirical success, and because classification has yielded so many theoretical results, that attempts should be made to synthesize the two paradigms to produce FRSs that have the best characteristics of both. Two alternative strategies might be used to approach this problem. The first is to investigate formal semantics for operations not currently supported within the classification paradigm, such as multiple modes of inheritance.

The second approach is to include additional expressiveness through nondefinitional slots that describe incidental properties of a class. The classification paradigm assumes that *all* slots of a given concept are definitional in that they are part of the specification of necessary conditions for what it means to be an instance of that concept. I suggest that by annotating slots as to whether they describe definitional or incidental attributes, and by having the classifier ignore incidental slots, we can provide greater expressiveness within the classification paradigm. For example, although a necessary condition of maleness is that the `Y-chromosome` slot have at least one value, we might want to record that the default weight of males is 150lb without making a definitional statement, and to allow a specific inheritance mode and conflict-resolution strategy to apply to this default. The separation of definitional from nondefinitional information may allow the two paradigms of knowledge representation to be unified.

LOOM and CLASSIC take steps in this direction. LOOM allows nonconflicting default information to be encoded in concert with concept definitions; CLASSIC allows default information to be encoded as rules. In both systems, this default information is ignored by the classifier. In addition, LOOM allows the user to define constraints that must hold for a given slot, but that are nondefinitional, i.e., are ignored by the classifier.

Another major difference between the two paradigms is the interpretation they ascribe to universally quantified formulas. Consider the concept of a red ball — a ball whose color is

[15] Traditionally, the word "expressiveness" has been used within the KR community to refer to the complexity of a definition. I use it here in the more general sense of the ability to represent richer types of definitional or nondefinitional knowledge, e.g., through facets or metaclasses.

red:

$$\forall\ x(RedBall(x) \equiv\ \ Ball(x) \wedge\ Color(x, red))$$

Such definitions are manipulated in four distinct ways by FRSs: in subsumption, in realization, in constraint checking, and in assertion.

A member of the KL-ONE family would encode such universally quantified information as a concept definition; it can compare such definitions to classify the Red-Ball concept, and it could recognize instances that satisfy this definition during realization. In contrast, the KEE FRS could encode this information as a value constraint on the color slot that could be used for checking if a particular frame that is asserted to be an instance of the Red-Ball class actually has a color of red. KEE could also encode this information as a default so that for every instance of the Red-Ball class, inheritance asserts that the value of the color slot is red.

Brachman made a similar point when he contrasted what he called assertional FRSs (e.g., KEE), from those that manipulate descriptions [9]. However, Brachman did not identify all four interpretations of universally quantified information, nor did he note that all four uses of such information can be required in certain situations. That the KL-ONE family cannot deduce that if we know a ball is a red ball, then its color is red, seems rather surprising. Newer members of the KL-ONE family can of course deal with assertional information, but such information must be stored by the A-Box, not the T-Box. Therefore, to employ a given piece of universally quantified information both definitionally and assertionally, it must be stated twice.

In summary, universal quantifications have several interpretations. They can be compared, matched against ground formulas for recognition and checking, and instantiated. No FRS can currently put universally quantified information to all these uses, although for every interpretation, there exists an FRS that employs it.

10 SUMMARY

FRSs are a surprisingly large and a surprisingly diverse group of information-management systems. Over 50 FRSs have been constructed by KR researchers. Three main families of FRSs have emerged: the SRL family, the KL-ONE family, and the UNIT Package family.

This paper surveyed the diversity of FRSs by examining the architectural variations that different system designers have explored for the frame, the slot, the knowledge base, access-oriented programming, and object-oriented programming.

Common to virtually all FRSs are the class frame and the instance frame. Other frame types include the following. Metaclasses represent classes of classes. Prototypes represent typical class instances. SlotUnits store global information about slots. SeeUnits act as annotations for other frames. CompactUnits are a variation of instance frames that require less storage and can be accessed more quickly than normal instance frames. Descriptions

are variablized frames that represent goals in planning problems. Indefinites are similar to skolem constants in that they represent instance frames of unknown identity.

In the simplest model of a slot, it has a name and a value. Many systems add arbitrary additional facets to the slot, of which a datatype and a value restriction are typical. In some cases, facets themselves can have facets, to an arbitrary depth. If we view a set of slot values as a class, we gain the ability to make powerful assertions about that set. Own slots contain information about the frame that they reside in, whereas member slots describe the properties of instances of a class frame. BookkeepingSlots describe an actual frame rather than the thing that the frame represents. FRSs employ a number of different slot datatypes, such as sets, ordered sets, lists, and single values. Constraints on slot values can be written in a range of languages. The constraints can apply to individual slots or can relate the values of a collection of slots. The constraints are generally used to detect unallowed values, or can serve a definitional role that forms the basis of classification.

Knowledge bases are collections of frames that are often stored in a single file. In some cases, KBs can be arranged in a hierarchy so that frames present in a given KB are also visible in its children. Many KB-storage mechanisms are simplistic because they require time proportional to the size of the KB to either save the KB persistently, or to load a KB into memory. More desirable would be if the time were proportional to the number of KB updates or to the amount of the KB that is accessed, respectively. Several researchers have begun to address these issues by exploring ways of interfacing FRSs to relational and object-oriented database systems.

FRSs employ the classification operation to compare the definitions of two classes to determine which is more general. Each member of the KL-ONE family uses a somewhat different language to encode concept definitions. Many theoretical studies have addressed the computational complexity of computing the subsumption relationship. For the class-definition languages that have been studied, the following relationship has always held: the more expressive the language, the higher the computational cost of computing subsumption within that language.

This survey makes apparent the fact that a large space of alternative FRS designs exists because as a designer plans a new FRS, they must decide what model(s) of the frame to employ, what model(s) of the slot to employ, and so on. FRS design principles should guide an FRS designer in making the optimal set of decisions with respect to their intended class of applications. These design decisions have a number of ramifications. They affect the worst-case theoretical complexity, the average-case theoretical complexity, and the real-world performance of a variety of FRS operations — from computing subsumption to altering slot values to computing slot inheritance. They affect the expressiveness and succinctness of the representation language, determining which nuances of the application can be encoded in the FRS, and how compact and understandable that encoding is. They affect the runtime flexibility of the FRS — the type of changes that can be made to a KB without requiring that time be spent reloading the KB (complex applications require frequent KB changes to class definitions and instances). They affect the modularity of the FRS; certain FRS features are highly interdependent, thus affecting the ease with which the system can be maintained and extended. They also affect the effort required to implement the FRS, since

some features and some engineering techniques are more costly to program than others.

This paper has shown that, unfortunately, few design principles exist to guide an FRS designer as to how particular design decisions will impact the preceding qualities of the resulting system, and what tradeoffs exist among alternative design choices. First and foremost, we must understand the scope of the FRS design space. This paper defines the FRS design space by surveying the features present in a large set of FRSs. This paper has closely examined the design principles that do exist. When possible, we have discussed the tradeoffs and interactions among different FRS features. We found that past research has yielded little understanding of the alternative engineering techniques that can be used to implement different FRS capabilities because authors rarely discuss the engineering techniques that they employ. Engineering decisions intimately affect many of the preceding FRS qualities.

The paper examined classification in detail. Many FRSs treat classification as central to their operation, and a large number of theoretical studies have investigated the computational costs of classification. The paper questions the central importance of classification by noting that neither the UNIT Package nor the SRL families of FRSs employ it all, and that the use of classification constrains the other features that can be present in an FRS. Research on ways to remove these constraints could prove quite fruitful. Furthermore, we argue that the theoretical results concerning the computational costs of subsumption may have been exaggerated. There is no evidence that the worst-case predictions will hold as applications scale up, and there is empirical evidence that real-world KBs do not require exponential time for classification. Finally, classification is used in a variety of tasks, and it is not clear for what tasks (if any) the speed of classification is supposed to be rate limiting. We considered alternative ways of decreasing the dependence of each of these tasks on classification, and suggested that classification may be completely inappropriate for one of these tasks.

The field of KR has the potential for entering a mature phase of great progress if we can capitalize on the wealth of research results that have accumulated over the past 20 years. This paper should help to unite many fragmented realms of FRS research to provide a solid foundation for future research.

11 ACKNOWLEDGEMENTS

Richard Fikes, Walter Sujansky, and David E. Wilkins provided helpful comments on earlier drafts of this paper. In addition, the paper benefitted from discussions with Ramesh Patil, David Israel, Tom Mitchell, and Peter Shell. This work was supported by the National Library of Medicine while the author was a postdoctoral fellow there, and by SRI International.

References

[1] R. Abarbanel and M. Williams. A relational representation for knowlege bases. Technical report, IntelliCorp, 1986.

[2] G. Abrett, M. Burstein, J. Gunsbenan, and L. Polanyi. KREME: A user's introduction. Technical Report 6508, BBN Laboratories Inc., Cambridge, MA, 1987.

[3] J.F. Allen and B.W. Miller. The Rhetorical knowledge representation system: A tutorial introduction. Technical Report 325, University of Rochester Computer Science Department, 1990.

[4] F. Baader and B. Hollunder. KRIS: Knowledge representation and inference system. *SIGART Bulletin*, 2(3), 1991.

[5] N. Ballou, H.T. Chou, J.F. Garza, W. Kim, C. Petrie, D. Russinoff, D. Steiner, and D. Woelk. Coupling an expert system shell with an object-oriented database system. *Journal of Object-Oriented Programming*, pages 12–21, June/July 1988.

[6] R.L. Bates and R. MacGregor. Knowledge representation in NIKL. In M.H. Richer, editor, *AI Tools and Techniques*, pages 183–196. Ablex Publishing, 1989.

[7] D.G. Bobrow and T. Winograd. An overview of KRL, a knowledge representation language. *Cognitive Science*, 1(1):???, 1977.

[8] D.G. Bobrow and T. Winograd. KRL another perspective. *Cognitive Science*, 3:29–42, 1979.

[9] R.J. Brachman. What's in a concept: structural foundations for semantic networks. *Int. J. Man–Machine Studies*, 9:127–152, 1977.

[10] R.J. Brachman. I lied about the trees. *AI Magazine*, 6(3):80–93, 1985.

[11] R.J. Brachman. The basics of knowledge representation and reasoning. *AT&T Technical Journal*, 67(1):7–24, 1988.

[12] R.J. Brachman, A. Borgida, D.L. McGuyinness, and L.A. Resnick. The CLASSIC knowledge representation system, or, KL-ONE: The next generation. In *Proceedings of the 1989 International Joint Conference on Artificial Intelligence*, page ??? Morgan Kaufmann Publishers, August 1989.

[13] R.J. Brachman, R.E. Fikes, and H.J. Levesque. KRYPTON: A functional approach to knowledge representation. *IEEE Computer*, 16(10):67–73, October 1983.

[14] R.J. Brachman, D.L. McGuinness, P.F. Patel-Schneider, and L.A. Resnick. Living with CLASSIC: When and how to use a KL-ONE-like language. In J. Sowa, editor, *Principles of semantic networks: Explorations in the representation of knowledge*, pages 401–456. Morgan Kaufmann Publishers, Los Altos, CA, 1990.

[15] R.J. Brachman and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[16] W.J. Clancey. Classification problem solving. In *Proceedings of the 1984 National Conference on Artificial Intelligence*, pages 49–55, 1984.

[17] Xerox Corporation. XEROX LOOPS reference manual. Technical report, Xerox Palo Alto Research Center, 1988.

[18] J.M. Crawford. *Access-Limited Logic — A language for knowledge-representation.* PhD thesis, University of Texas at Austin, 1990. Technical report AI90-141.

[19] J.M. Crawford and B. Kuipers. ALL: Formalizing access limited reasoning. In J. Sowa, editor, *Principles of semantic networks: Explorations in the representation of knowledge*, pages 299–330. Morgan Kaufmann Publishers, Los Altos, CA, 1990.

[20] J. De Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(1):127–162, 1986.

[21] J. Doyle and R.S. Patil. Two dogmas of knowledge representation. Technical Report MIT/LCS/TM-387., Massachusetts Institute of Technology Laboratory for Computer Science, September 1989.

[22] J. Doyle and R.S. Patil. Two theses of knowledge representation: Language restrictions, taxonomic classification, and the utility of representation services. *Artificial Intelligence*, 48(3):261–298, 1990.

[23] J. Esakov. KB: A knowledge representation package for Common Lisp. Technical Report MS-CIS-90-03, University of Pennsylvania Department of Computer and Information Science, Phila, PA, 1990.

[24] M. Evett, J. Hendler, and L. Spector. PARKA: Parallel knowledge representation on the connection machine. Technical Report UMIACS-TR-90-22, Department of Computer Science, University of Maryland, February 1990.

[25] Y.A. Feldman and C. Rich. Bread, Frappe, and Cake: The gourmet's guide to automated deduction. In *Proc. 5th Israeli Symp. on Artificial Intelligence, Vision, and Pattern Recognition*, Tel Aviv, Israel, December 1988.

[26] R. Fikes. Building the foundation for tomorrow's expert systems: Advances in object-oriented modeling. In *Proceedings of the fifth Australian conference on applications of expert systems*, May 1989.

[27] R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Communications of the Association for Computing Machinery*, 28(9):904–920, 1985.

[28] R. Fikes, R. Nado, R. Filman, P. McBride, P. Morris, A. Paulson, R. Treitel, and M. Yonke. OPUS: A new generation knowledge engineering environment. Technical report, IntelliCorp, Mountain View, CA, 1987. DARPA Contract F30602-85-C-0065 final report.

[29] R.E. Filman. Reasoning with worlds and truth maintenance in a knowledge-based system shell. *Communications of the Association for Computing Machinery*, 31(4), April 1988.

[30] T. Finin. Understanding frame languages. *AI Expert*, pages 44–50, November 1986.

[31] M. Fox, J. Wright, and D. Adam. Experiences with SRL: An analysis of a frame-based knowledge representation. In *Expert Database Systems*. Benjamin/Cummings, 1985.

[32] A. Garvey, M. Hewett, M.V. Jr. Johnson, R. Schulman, and B. Hayes-Roth. BB1 user manual — Common Lisp version 2.0. Technical Report KSL-86-61, Stanford University Knowledge Systems Laboratory, 1987.

[33] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[34] R. Greiner and D.B. Lenat. Details of RLL-1. Technical Report HPP-80-23, Stanford University Heuristic Programming Project, Stanford, CA, 1980.

[35] R. Greiner and D.B. Lenat. RLL: A representation language language. In *Proceedings of the 1980 National Conference on Artificial Intelligence*, pages 165–9. Morgan Kaufmann Publishers, 1980.

[36] R.V. Guha. Micro-theories and contexts in Cyc part I: Basic issues. Technical Report ACT-CYC-129-9, MCC, June 1990.

[37] K.W. Haase Jr. ARLO: another representation language offer. Technical Report 901, Massachusetts Institute of Technology AI Laboratory, 1986.

[38] J. Heinsohn, D. Kudenko, B. Nebel, and H.J. Profitlich. An empirical analysis of terminological representation systems. In *Proceedings of the 1992 National Conference on Artificial Intelligence*, pages 767–773. Morgan Kaufmann Publishers, 1992.

[39] J. Heinsohn, D. Kudenko, B. Nebel, and H.J. Profitlich. An empirical analysis of terminological representation systems. Technical Report KR-92-16, German Research Center for Artificial Intelligence, Kaiserslautern, FRG, 1992.

[40] ISX Corporation. *LOOM Users Guide, version 1.4*, August 1991.

[41] T.S. Kaczmarket, R. Bates, and G. Robins. Recent developments in NIKL. In *Proceedings of the 1986 National Conference on Artificial Intelligence*, pages 978–985. Morgan Kaufmann Publishers, 1986.

[42] T.P. Kehler and G.D. Clemenson. KEE the knowledge engineering environment for industry. *Systems And Software*, 3(1):212–224, January 1984.

[43] A. Kobsa. Utilizing knowledge: The SB-ONE knowledge representation workbench. In J. Sowa, editor, *Principles of semantic networks*, pages 457–486. Morgan Kaufmann Publishers, 1991.

[44] D. Lanam, R. Letsinger, S. Rosenberg, P. Huyun, and M. Lemon. Guide to the heuristic programming and representation language part 1: Frames. Technical Report AT-MEMO-83-3, Application and Technology Laboratory, Computer Research Center, Hewlett-Packard, January 1984.

[45] C. Lane. The Ozone reference manual. Technical Report KSL-86-40, Stanford University Knowledge Systems Laboratory, July 1986.

[46] W. Lehnert and Y. Wilks. A critical perspective on KRL. *Cognitive Science*, 3:1–28, 1979.

[47] D.B. Lenat and R.V. Guha. The world according to CYC. Technical Report ACA-AI-300-88, MCC, September 1988.

[48] D.B. Lenat and R.V. Guha. *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project.* Addison-Wesley, 1990.

[49] D.B. Lenat, R.V. Guha, and D.V. Wallace. The CycL representation language. Technical Report ACA-AI-302-88, MCC, September 1988.

[50] H.J. Levesque and R.J. Brachman. Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, 3(2):78–93, 1987.

[51] R. MacGregor. The evolving technology of classification-based knowledge representation systems. In J. Sowa, editor, *Principles of semantic networks*, pages 385–400. Morgan Kaufmann Publishers, 1991.

[52] E. Mays, C. Apte, J. Griesmer, and J. Kastner. Experience with k-rep: An object-centered knowledge representation language. In *Proc. IEEE Conference on AI Applications*, San Diego, California, March 1988.

[53] E. Mays, S. Lanka, B. Dionne, and R. Weida. A persistent store for large shared knowledge bases. *IEEE Trans. on Knowledge and Data Eng.*, 3(1):33–41, 1991.

[54] D.A. McAllester. Reasoning utility package user's manual. Technical Report AI Memo 667, Massachusetts Institute of Technology AI Laboratory, 1982.

[55] D.P. McKay, T.W. Finin, and A. O'Hare. The intelligent database interface: Integrating AI and database systems. In *Proceedings of the 1990 National Conference on Artificial Intelligence*, pages 677–684. Morgan Kaufmann Publishers, 1990.

[56] B.W. Miller. The Rhetorical knowledge representation system reference manual. Technical Report 326, University of Rochester Computer Science Department, 1990.

[57] T.M. Mitchell, J. Allen, P. Chalasani, J. Cheng, E. Etzioni, M. Ringuette, and J.C. Schlimmer. Theo: A framework for self-improving systems. In *Architectures for Intelligence*. Erlbaum, 1989.

[58] E. Muehle. FROBS user guide. Technical Report 87-05, University of Utah PASS Project, 1989.

[59] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: A language for representing knowledge about information systems. *ACM Transactions on Information Systems*, 1991. In press.

[60] R. Nado and R. E. Fikes. Saying more with frames: Slots as classes. *Computers and Mathematics with Applications*, 23(6–9):719–731, 1992.

[61] B. Nebel. Computational complexity of terminological reasoning in BACK. *Artificial Intelligence*, 34(3):371–384, 1988.

[62] B. Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43(2):235–250, 1990.

[63] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W.R. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36–56, 1991.

[64] E.H. Nyberg. The FrameKit user's guide, version 2.0. Unpublished system manual, Carnegie Mellon University, 1988.

[65] P. Patel-Schneider. Small can be beautiful in knowledge representation. In *Proceedings of IEEE workshop on principles of knowledge-based systems*, 1984.

[66] P. Patel-Schneider. Undecidability of subsumption in NIKL. *Artificial Intelligence*, 39:263–273, 1989.

[67] C. Peltason, A. Schmiedel, C. Kindermann, and J. Quantz. The BACK system revisited. Technical Report KIT 75, Department of Computer Science, Technische Universitaet Berlin, September 1990.

[68] C.J. Jr. Petrie, D.M. Russinoff, D.D. Steiner, and N. Ballou. Proteus 2: System description. Technical Report AI-136-87, MCC, May 1987.

[69] L.A. Resnick, A. Borgida, R.J. Brachman, D.L. McGuinness, P.F. Patel-Schneider, and K.C. Zalondek. CLASSIC description and reference manual for the Common Lisp implementation, version 1.1. Unpublished system manual, 1991.

[70] R.B. Roberts and I.P. Goldstein. The FRL manual. Technical Report 409, Massachusetts Institute of Technology AI Laboratory, 1977.

[71] R.B. Roberts and I.P. Goldstein. The FRL primer. Technical Report 408, Massachusetts Institute of Technology AI Laboratory, 1977.

[72] S. Rosenberg. HPRL: A language for building expert systems. In *Proceedings of the 1983 International Joint Conference on Artificial Intelligence*, 1983.

[73] D.M. Russinoff. Proteus: A frame-based nonmonotonic inference system. Technical Report ACA-AI-302-87, MCC, 1987.

[74] P. Rychlik. Multiple inheritance systems with exceptions. *Artificial Intelligence Review*, 9:159–176, 1989.

[75] K. Schild. Undecidability of subsumption in U. Technical Report KIT report 67, Fachbereich Informatik, Technical University of Berlin, 1988.

[76] J.G. Schmolze. The language and semantics of NIKL. Technical Report 89-4, Tufts University Department of Computer Science, September 1989.

[77] J.G. Schmolze and T.A. Lipkis. Classification in the KL-ONE knowledge representation system. In *Proceedings of the 1983 International Joint Conference on Artificial Intelligence*, Los Altos, CA, August 1983. Morgan Kaufmann Publishers.

[78] J.G. Schmolze and W.S. Mark. The NIKL experience. *Computational Intelligence*, 1991. In press; see also Tufts Unviersity Department of Computer Science technical report 90-6.

[79] E. Schoen. Personal communication. unpublished, 1991.

[80] E. Schoen, R.G. Smith, and P. Carando. Ruling with Class. Technical Report TD-89-17, Schlumberger Laboratory for Computer Science, 1989.

[81] E.J. Schoen. *Intelligent assistance for the design of knowledge-based systems.* PhD thesis, Stanford University Computer Science Department, 1990. Report number STAN-CS-90-1332.

[82] L.K. Schubert. Semantic nets are in the eye of the beholder. In J. Sowa, editor, *Principles of semantic networks*, pages 95–108. Morgan Kaufmann Publishers, 1991.

[83] B. Selman and H.H. Levesque. The tractability of path-based inference. In J. Sowa, editor, *Principles of semantic networks*, pages 283–298. Morgan Kaufmann Publishers, 1991.

[84] S.C. Shapiro. The SNePS semantic network processing system. In N.V. Findler, editor, *Associative networks: The representation and use of knowledge by computers*, pages 179–203. Academic press, New York, 1979.

[85] S.C. Shapiro. SNePS-2.1 user's manual. Technical Report SNeRG technical note 4, Department of Computer Science, SUNY at Buffalo, 1989.

[86] P. Shell and J. Carbonell. Parmenides: A class-based frame system. Unpublished system manual, Carnegie Mellon University, 1989.

[87] D.E. Smith. CORLL manual: A storage and file management system for knowledge bases. Technical Report HPP-80-8, Stanford University Heuristic Programming Project, Stanford, CA, 1980.

[88] R.G. Smith and P. Carando. Structured object programming in Strobe. Technical report, Schlumberger-Doll Research, December 1986.

[89] R.G. Smith, P. Friedland, and M. Stefik. Unit Package user's guide. Technical Report HPP-80-28, Stanford University Knowledge Systems Laboratory, December 1980.

[90] R.G. Smith, G.M.E. Lafue, E. Schoen, and S.C. Vestal. Declarative task description as a user interface structuring mechanism. *IEEE Computer*, 17(9):29–38, September 1984.

[91] R.G. Smith and E. Schoen. Programming with Class. Technical Report TD-90-17, Schlumberger Laboratory for Computer Science, 1990.

[92] J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley, Reading, MA, 1984.

[93] M. Stefik. An examination of a frame-structured representation system. In *Proceedings of the 1979 International Joint Conference on Artificial Intelligence*, pages 845–852. Morgan Kaufmann Publishers, 1979.

[94] M. Stefik and D.G. Bobrow. Object-oriented programming: Themes and variations. *AI Magazine*, 6(4):40–62, 1986.

[95] M.J. Stefik, D.G. Bobrow, and K.M. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 3(1):10–18, January 1986.

[96] A.L. Stein. Extensions as possible worlds. In J. Sowa, editor, *Principles of semantic networks*, pages 267–282. Morgan Kaufmann Publishers, 1991.

[97] R.H. Thomason and D.S. Touretzky. Inheritance theory and networks with roles. In J. Sowa, editor, *Principles of semantic networks*, pages 231–266. Morgan Kaufmann Publishers, 1991.

[98] M. Vilain. The restricted language architecture of a hybrid representation system. In *Proceedings of the 1985 International Joint Conference on Artificial Intelligence*, pages 547–551. Morgan Kaufmann Publishers, 1985.

[99] K. von Luck, B. Nebel, C. Peltason, and A. Schmiedel. The anatomy of the BACK system. Technical Report KIT 41, Department of Computer Science, Technische Universitaet Berlin, 1987.

[100] W.A. Woods. Understanding subsumption and taxonomy: A framework for progress. In J. Sowa, editor, *Principles of semantic networks*, pages 45–94. Morgan Kaufmann Publishers, 1991.

[101] J. Yen, H.L. Juang, and R. MacGregor. Using polymorphism to improve expert system maintainability. *IEEE Expert*, 6(1):48–55, April 1991.