A Deductive Pattern Matcher*

Robert M. Mac Gregor
USC/Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292
macgreg@vaxa.isi.edu

Abstract

This paper describes the design of a pattern matcher for a knowledge representation system called LOOM. The pattern matcher has a very rich pattern-forming language, and is logic-based, with a deductive mechanism which includes a truth-maintenance component as an integral part of the pattern-matching logic. The technology behind the LOOM matcher uses an inference engine called a classifier to perform the matches. The LOOM matcher is more expressive and more complete than previous classification-based pattern-matchers, and is expected to be significantly more efficient.

1 Introduction

This paper describes the pattern-matching facility that has been designed for a knowledge representation system called LOOM [MacGregor and Bates, 1987]. LOOM belongs to the KL-ONE [Brachman and Schmolze, 1985] family of knowledge representation systems. A distinctive pattern-matching architecture has been developed for some of these systems—they use an analytic inference engine called a classifier to perform the match operation. As we shall see later, these pattern matchers apply the results of logical deductions to the pattern-matching process—hence, we refer to them as *deductive* pattern matchers.

In the classification-based approach to pattern matching, an instance is matched to a pattern by first abstracting it, and then classifying the abstraction. This strategy is employed by two recent systems, KL-TWO [Vilain, 1985] and BACK [Luck *et al.*, 1987; Nebel and Luck, 1987]. In this paper, we describe an extension of this approach which (1) is deductively more powerful than, and (2) is expected to be more efficient than, the strategies used in these earlier systems.

In many KL-ONE-style knowledge representation systems, two languages are provided for expressing knowledge, a *concept* language and a *fact* language.[1] The concept language expresses knowledge about unary relations (which we call *concepts*) and binary relations (which we simply call *relations*). The fact language states facts about individuals. If the assertions about an individual I collectively satisfy the definition of some concept C, then I is an instance of C. In the classification-based approach, a concept P is associated with a pattern $P(x)$; thus, matching an individual to a pattern corresponds to recognizing an instantiation relationship between the individual and the corresponding concept.

Section 2 describes LOOM's language for defining concepts/patterns; Section 3 introduces the notion of the *type* of a database individual, and illustrates how an individual's type can change as facts are asserted or retracted; Section 4 opens with an outline of the deductive architecture of the LOOM matcher, and then illustrates it with an extended example; Section 5 shows how the expressiveness of the pattern language increases when implications between concepts/patterns can influence the pattern semantics; Section 6 briefly suggests how LOOM's pattern matcher can be employed to drive a production-rule system; Section 7 contains a discussion of some of the practical implications of the LOOM architecture.

2 The Concept-Forming Language

LOOM provides a relational algebra for creating definitions of concepts and relations. The operators `defconcept` and `defrelation` are invoked to bind a symbol to a relational algebra expression—binding a symbol to a concept (or relation) expression effectively defines a new predicate symbol. For example, after evaluating the `defconcept` for `Person` in Figure 1, we can employ `Person` in our fact language: `(tell (person Bill))` asserts that `Bill` is a `Person`, while `(ask (Person Bill))` tests to see if `Bill` satisfies `Person`.

The language contains three classes of elementary concept expressions: (1) The term `:primitive` denotes a unique[2], primitive[3] concept; (2) A role-restriction quantifier can be applied to a relation to generate a *role-restricting* concept defined by the restriction placed on that relation, e.g., the expression `(:at-least 2 child)` denotes the concept such that the attached role `child` must have at least two role fillers. The language provides the numeric quantifiers `:at-least`, `:at-most`, and `:exactly`, and a universal quantifier `:all`; (3) A *role-relating* concept expression specifies a relationship which constrains

*This research was sponsored by the Defense Advanced Research Projects Agency under contract MDA903-81-C-0335.

[1] We are referring here to *hybrid* knowledge representation systems—a hybrid system incorporates multiple reasoners which apply to separate partitions of the knowledge space (see [Vilain, 1985; Brachman *et al.*, 1983]).

[2] Formally, each appearance of the term `:primitive`, in a sequence of concept expressions denotes a *different* concept, i.e., the i^{th} appearance of `:primitive` represents the i^{th} primitive concept.

[3] A concept or relation is *primitive* if it cannot be completely characterized in terms of other concepts(relations).

```
(defconcept Person :primitive)
(defconcept Male (:and Person :primitive))
(defconcept Female (:and Person :primitive))
(defconcept Married (:and Person :primitive))
(defrelation child :primitive
    (:implies (:domain Person) (:range Person))
    :closed-world)
(defrelation daughter
         (:and child (:range Female)))
(defconcept Father
         (:and Male (:at-least 1 child)))
(defconcept Successful-Father
         (:and Father (:all daughter Married)))
```

Figure 1: Concept and Relation Definitions

the fillers of two or more of the concept's roles, e.g., the expression `(= input-voltage output-voltage)` specifies the concept such that the fillers of the roles `input-voltage` and `output-voltage` have the same value.

Compound expressions are built-up from simple expressions by applying the operators `:and`, `:or`, and `:not`, which correspond to the operations of intersection, union, and relative complement, respectively.

An elementary relation expression consists of either the term `:primitive` (which in this context denotes a unique *relation* instead of a concept) or the projection of a relation defined by restricting its domain or its range. For example, in Figure 1, the relation `daughter` is defined by intersecting the relation `child` with the relation defined by restricting the range of the universal binary relation to the concept `Female`.

The language provides specialized operators other than those just mentioned (e.g., the unary operator `:inverse` which generates the inverse of a relation). It also contains a special syntax for defining concepts representing sets or intervals, e.g., the set of colors {`Red, Blue, Yellow, ...`} or the range of numbers greater than 4.

In Figure 1, `Father` is defined as "a `Male` with at least one `child`", while a `Successful-Father` is "a `Father` all of whose `daughters` are `Married`." These declarations define the predicates `Father` and `Successful-Father`. If we assert in our fact language `(tell (Male Bill) (child Bill Mary))` then the query `(ask (Father Bill))` returns `TRUE`.

Figure 3 provides a formal semantics for the expressions illustrated in Figure 1.

The `defconcept` and `defrelation` operators permit non-terminological knowledge to be asserted about a newly-defined concept or relation. The `:implies` clause in the definition of `child` (see Figure 1) asserts that its domain and range fillers must satisfy the predicate `Person`. Section 5 contains further examples illustrating the use of `:implies`. Covering and disjointness relationships can also be asserted between concepts and relations.

We note that the ability to attach a *name* to a pattern, so that it can be referenced[4] within other patterns, is absent in most rule-based languages.

[4]Note: LOOM requires that references between patterns not form a cycle.

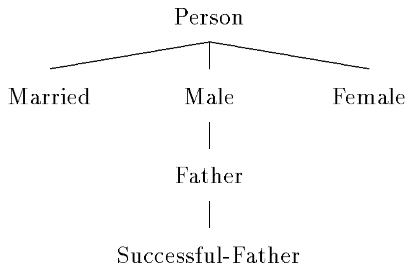

Figure 2: Tree of Classified Concepts

LOOM adopts an open-world semantics, and employs a three-valued (TRUE, UNKNOWN, FALSE) logic. The keyword `:closed-world` in the definition of `child` in Figure 1 indicates that closed-world semantics apply when determining membership in the relation `child`.

3 Types

LOOM allocates a *database object* to represent each individual about which one or more facts have been asserted. The primary function of the LOOM pattern matcher is to maintain an up-to-date record of all *instantiates* relationships between individuals in the database and concepts whose definitions they match. The intersection of all concepts matched by a particular individual is called the *type* of the individual. An encoding of the type, consisting of a list of the *most-specific* concepts belonging to the type, is attached to each database object. Figure 2 shows the hierarchy of concepts defined by the definitions in Figure 1. If a database object matches the concepts `Married`, `Person`, and `Female`, then its type is `(:and Married, Person, Female)`, while the list `(Married Female)` represents an encoding of that type.

As facts about database individuals are asserted or retracted, the types of those individuals will change. The left side of Figure 4 shows a sequence of assertions and retractions made to an (initially empty) database, while the right side of the figure shows the changes in the (encoded) types of the individuals that result from the updates.

The next section walks us through this sequence of updates, and discusses how the computation of these types is accomplished.

4 The Deductive Cycle

This section outlines the deductive architecture of the pattern matcher, and illustrates how the deductive machinery works by tracing the deductions applied to the database individual `Fred` after each of the assertions shown in Figure 4.

A concept A *subsumes* a concept B if the definitions of A and B logically imply that members of B must also be members of A. For example, in our knowledge base, `Male`(X) implies `Person`(X) for all individuals X; hence, `Person` subsumes `Male`. An important feature of classification-based systems such as LOOM is their ability to compute subsumption relationships between concepts and relations. A *classifier* [Schmolze and Lipkis, 1983]

Expression e	*Interpretation* $[\![e]\!]$
:primitive (*concept*)	a unique primitive concept
:primitive (*relation*)	a unique primitive relation
(**:and** C_1 C_2)	$\lambda x.\, [\![C_1]\!](x) \wedge [\![C_2]\!](x)$
(**:and** R_1 R_2)	$\lambda xy.\, [\![R_1]\!](x,y) \wedge [\![R_2]\!](x,y)$
(**:at-least** 1 R)	$\lambda x.\, \exists y.\, [\![R]\!](x,y)$
(**:exactly** 1 R)	$\lambda x.\, \exists y.\, [\![R]\!](x,y) \wedge \forall yz.\, ([\![R]\!](x,y) \wedge [\![R]\!](x,z)) \rightarrow y = z$
(**:all** R C)	$\lambda x.\, \forall y.\, [\![R]\!](x,y) \rightarrow [\![C]\!](y)$
(**:domain** C)	$\lambda xy.\, [\![C]\!](x)$
(**:range** C)	$\lambda xy.\, [\![C]\!](y)$
(defconcept C ... (**:implies** C_1 C_2))	$\forall x.\, [\![C]\!](x) \rightarrow ([\![C_1]\!](x) \wedge [\![C_2]\!](x))$
(defrelation R ... (**:implies** R_1 R_2))	$\forall xy.\, [\![R]\!](x,y) \rightarrow ([\![R_1]\!](x,y) \wedge [\![R_2]\!](x,y))$

Figure 3: Semantics of Some Term-Forming Expressions

performs the service of computing all subsumption relationships between a given concept and all other concepts in a concept network. This procedure is made efficient by organizing all concepts into a partial lattice in which more-general concepts (subsumers) are placed above less-general concepts (subsumees). The tree shown in Figure 2 shows the result of classifying each of the concept definitions from Figure 1.[5]

The search algorithm employed by the LOOM classifier for applying subsumption tests was developed by Tom Lipkis for the NIKL (see [Moser, 1983]) classifier. Here we list three properties which serve as a partial characterization of the Lipkis algorithm. We will use the word *test* to mean "perform a subsumption test on". Let N stand for the concept being classified:

1. Don't test a concept C if a descendant (subsumee) of C is known to subsume N;
2. Don't test a concept C unless at least one parent (immediate subsumer) of C is known to subsume N;
3. Don't test primitive concepts.

When classifying large networks, the observed effect of these properties is that only a small percentage of all classified concepts are tested during the classification of a new concept.

Within the classification-based paradigm, a concept is treated *as if* it were a pattern. We will say an individual I *matches* a concept C if I is an instance of C. It follows that we equate the process of finding all concepts which I matches with the process of computing the type of I. A complete pattern-matching system requires the following facility—after every change to the database, we want to compute matches for *all* database individuals against *all* concepts, i.e., we need to *continuously* maintain the type of each individual. The program that accomplishes this task is called a *recognizer*.[6]

[5] Note: The LOOM classifier also builds a partial lattice reflecting all subsumption relationships computed between (binary) relations. These are not shown in Figure 2.

[6] The term *realizer* is sometimes used in place of the term recognizer.

We will call two database individuals I_1 and I_2 *adjacent* if for some relation r, the predication $r(I_1, I_2)$ has been asserted. Three types of database modifications can cause the type of a database individual I to change:

i) asserting or retracting a unary predicate on I;

ii) changing the value of one of I's roles;

iii) changing the type of an individual adjacent to I.

Applying the procedure *adjust-individual-type* to an individual I will

1. recompute I's type (if necessary), and
2. call *adjust-individual-type* recursively, if I's type has changed, to adjust the types of individuals adjacent to I.

The job of the recognizer is to apply the procedure *adjust-individual-type* after each database update to any individuals affected by a class (i) or class (ii) change.

A concept expression which is matched by an instance I is called an *abstraction* of I. The procedure *adjust-individual-type* employs a strategy called abstraction/classification (A/C):

> "to find those concepts which are matched by an individual I, we form an abstraction A of I, and then classify A. I necessarily matches all concepts which subsume A."

The straightforward application of the A/C strategy represents an elegant but impractical method for computing the type of an individual: if the abstraction A chosen for I is not sufficiently complete, then only *some* of the concepts matching I will be found by classifying A. However, the abstraction-generating schemes used in KL-TWO and BACK are designed to match only a subset of the possible concept expressions, because the cost of generating a sufficiently-detailed abstraction is prohibitive.[7]

The solution is to abandon a purely forward-chaining A/C strategy in favor of one in which the classifier, while classifying an abstraction, can ask questions about the individual behind the abstraction being classified—the questions represent backward chaining. In the LOOM scheme,

[7] In particular, the abstractions they generate do not include the *role-relating* concepts defined in section 2.

Step	Encoded Type(s)
A1. (tell (Married Fred))	Fred: (Married)
A2. (tell (child Fred Suzy))	Suzy: (Person)
A3. (tell (Male Fred))	Fred: (Father Married)
A4. (tell (Female Suzy))	Suzy: (Female)
A5. (tell (Married Suzy))	Fred: (Successful-Father Married); Suzy: (Married Female)
A6. (forget (Married Suzy))	Fred: (Father Married); Suzy: (Female)
A7. (tell (Successful-Father Fred))	Fred: (Successful-Father Married); Suzy: (Married Female)

Figure 4: Database Assertions and Retractions

the classifier, while classifying an abstraction A of I, can interrogate I directly about details missing in the abstraction A. Rather than generating a "complete" abstraction to begin with, a sufficiently-detailed abstraction for I is built-up incrementally during the A/C process.

The existing implementations of abstraction/classification pattern-matchers are relatively inefficient: they recompute a database object's type each time it is modified, i.e., a classification occurs once per database update. Also, there may be facts about an individual which get abstracted, but are not used during the classification step—these components of the abstraction represent wasted computation. The LOOM scheme avoids these problems:

Initially, the abstraction of an individual consists only of the conjunction of all unary predicates (concepts) asserted for that individual. While computing the type of a database object, three lists are attached to the object: TYPE is a list of the most-specific concepts matched so far; HITS is a list of questions (phrased as algebraic expressions) which received positive answers; MISSES contains questions which received non-positive answers. Whenever the value of a role R of a database individual is modified, the expressions in the individual's HITS and MISSES lists are inspected. If the answers to the HITS expressions are still positive, while the answers to MISSES questions are not positive, then no recomputation of the type is necessary. Each augmentation of the HITS list becomes an augmentation of the abstraction as well.

Summarizing, the LOOM pattern-matcher embodies three ideas which distinguish it from the earlier abstraction/classification pattern matchers

1. The algorithm generates an abstraction incrementally rather than all at once; this is possible because
2. The classification step mixes backward chaining (the questions) with forward chaining (the normal mode of a classifier);
3. The addition of the HITS and MISSES lists significantly reduces the type-computation overhead, i.e., it reduces the frequency of classification.

Here we trace the list activity as `Fred`'s type is maintained in the presence of the assertions in Figure 4. This trace illustrates the points just made.

A1. "`Fred` is Married"

The initial abstraction of `Fred` is

`(defconcept Fred (:and Married :primitive))`.

The classifier makes no subsumption tests at all. Thus, the classifier does not match `Fred` against the "role-bearing" concepts `Father` and `Successful-Father`, and hence no questions about role values were posed. The resulting state of `Fred` is:

TYPE: `(Married)` HITS: () MISSES: ().

A2. "`Suzy` is `Fred`'s `child`"

`Fred`'s `child` role has been modified—it is now a set containing `Suzy`—but there were still no questions in HITS and MISSES pertaining to the role `child`, so `Fred`'s type is not recomputed.

A3. "`Fred` is `Male`"

The abstraction for `Fred` now becomes

```
(defconcept Fred (:and Male Married
                       :primitive)).
```

The abstraction is tested against the concept `Father`; the classifier asks `Fred` the question `(:at-least 1 child)`, i.e., "Does `Fred` have at least one child?" The answer comes back `TRUE`, the test succeeds, and we test next against `Successful-Father`. The classifier asks `(:all daughter Married)`. `Fred`'s child `Suzy` may or may not be `Female`, and may or may not by `Married`, so the answer is `UNKNOWN`. Fred's state is now:

TYPE: `(Father Married)`
HITS: `((:at-least 1 child))`
MISSES: `((:all daughter Married))`

A4. "`Suzy` is `Female`"

`Suzy`'s type changes from `(Person)` to `(Female)`. `Suzy` notifies all adjacent database objects, including `Fred`, that its type has changed. `Fred` determines that `Suzy` is now his `daughter` as well as his `child`. However, the answer to `(:all daughter Married)` is still `UNKNOWN`, so `Fred`'s type is not recomputed.

A5. "`Suzy` is `Married`"

`Suzy` recomputes its type, and again notifies `Fred`. This time the answer to `(:all daughter Married)` is `TRUE`. The abstraction

```
(defconcept Fred (:and Male Married :primitive
                  (:all daughter Married)))
```

is classified; The resulting state is

TYPE: `(Successful-Father Married)`
HITS:
`((:at-least 1 child) (:all daughter Married))`
MISSES: ()

A6. Retract "`Suzy` is `Married`"

Retracting the assertion `(Married Suzy)` causes `Suzy`'s type to revert back to Female. `Suzy` once more notifies `Fred` that it's type changed. `Fred` checks its HITS and MISSES lists and discovers that `(:all daughter Married)` is no longer true; `Fred`'s type is recomputed from an abstraction built from those members of HITS which are still true:

```
(defconcept Fred
(:and Male Married :primitive
                    (:at-least 1 child)))
```

`Fred`'s state now matches the previous state after the assertion A4.

A7. "`Fred` is a `Successful-Father`"

Here we see an example of *forward* deduction rather than backward deduction: The assertion `(Successful-Father Fred)` not only causes `Fred` to recompute its type; `Fred` broadcasts to all fillers of the role `daughter` (in this case `Suzy`) that they now satisfy the predicate `Married`. This causes `Suzy` to revise its status. Hence, the types for `Fred` and `Suzy` now match the state after the assertion A5. However, because `Successful-Father` was asserted directly, and because `Successful-Father` implies `Father`, during this pass the classifier did not ask `Fred` any questions about the roles `child` and `daughter`, and hence the HITS list is different then it was after assertion A5. `Fred`'s state is now

TYPE: `(Successful-Father Married)`
HITS: `((:at-least 1 child))`
MISSES: ()

A couple of points are worth noting: First, the kind of inference observed after assertion A7 that we have called *forward* inference is not performed by most pattern matchers. To achieve this type of reasoning necessitates (i) that the pattern-defining language is capable of, i.e., rich-enough, to express such a logical dependency (most aren't); and (ii) that the pattern-matcher exhibit a facility for *deductive inference* as well as ordinary matching.

Second, the communication which takes place between database objects accomplishes the task of truth-maintenance—this is discussed further in section 7.

5 Implications

KL-ONE-style languages draw a distinction between *terminological*, or term-defining knowledge, and all other knowledge (called *assertional* knowledge, see [Brachman *et al.*, 1983]). A classifier computes the subsumption relationship between a pair of concepts solely on the basis of the terminologically-specified definitions of the two concepts. Previous classification-based pattern matchers have equated the language used to express patterns with the terminological language used to define concepts. LOOM breaks this habit by permitting some classes of assertional knowledge to contribute to the pattern-definitions: implications, covering relationships, and disjointness relationships. This section provides an illustration of the additional deductive power that implications of the form "$\forall x. C(x) \rightarrow D(x)$" bring to our pattern matcher.[8]

[8] No collective agreement has been arrived at as to exactly where the boundary between terminological and assertional knowledge lies, but there seems to be general agreement that implications represent assertional knowledge.

```
L1. (defconcept List :primitive)
L2. (defconcept Null :primitive (:implies List))
L3. (defconcept Cons
              (:and :primitive (:exactly 1 car)
                               (:exactly 1
cdr)))
L4. (defconcept Cons-List
              (:and Cons (:all cdr List))
          (:implies List))
```

Figure 5: A Lisp List

To date, none of the terminological languages implemented for a KL-ONE-style system permit one to define recursive or self-referential concepts, such as the concept of a Lisp list. LOOM, however, is able to augment a terminological definition of the concept `List` with implications which permit its pattern matcher to recognize a `List` when one occurs in the database. Figure 5 illustrates how this can be done. The implication in line L2 provides the basis step, $\forall x.\, \texttt{Null}(x) \rightarrow \texttt{List}(x)$. The implication in line L4 supplies the inductive step, $\forall x.\, (\texttt{Cons}(x) \wedge \forall y.\, (\texttt{cdr}(x,y) \rightarrow \texttt{List}(y))) \rightarrow \texttt{List}(x)$.

Suppose we assert

```
(tell (Cons c1) (Cons c2)
      (cdr c1 c2) (cdr c2 nil) (Null nil))
```

First, the object `nil` will classify as an instance of `Null`; next the implication L2 will enable the deduction `List(nil)`, so the type of `nil` is computed to be `(Null List)`; next, the type of `c2` will be computed as `(Cons-List List)`, using the implication L4; finally, the type of `c1` will become `(Cons-List List)`. Hence, we have inferred `(List c1)`.

To produce the inferences just illustrated, we added two extensions to the LOOM classifier. First, for each concept C in the classification hierarchy, LOOM computes a second concept representing the intersection of all concepts *implied by* C. The intersection includes concepts found by inheriting implies relationships from subsumers of C, and concepts found by computing the reflexive-transitive-closure of the implies relationship at C. The second extension builds on the first—after computing the type of a database individual, the classifier intersects that type with all concepts implied by that type, and returns the intersection concept as the new type of the individual.

6 Production Rules

Although the LOOM architecture does not currently provide a production rule facility, it is designed for that possibility—[Yen *et al.*, 1988] describes a production-rule language being built on top of LOOM.

The scheme is straight-forward: Whenever the type of a database object changes, the old and new types are compared. Each concept missing in the old type but present in the new type corresponds to a newly-matched pattern. LOOM instantiates the database object with all production rules that have that pattern as a pre-condition (left-hand-side).

7 Discussion

In this section, we discuss some of the advantages, and one disadvantage, of LOOM's pattern-matching architecture. In particular, we contrast it with OPS5-style [Forgy, 1981] pattern matchers.

Expressiveness of the Pattern Language

LOOM's pattern language is both more and less expressive than the typical OPS5-style pattern language. The forte of KL-ONE-based languages is in representing and reasoning about roles—they provide a rich set of operators for describing set-valued roles, chains of roles, and relationships between roles. In addition to the subsumption lattice for concepts, they define a separate lattice for recording subsumption relationships between relations, which allows one to state, for example, that the role `daughter` is a specialization of the role `child`. LOOM's ability to express implication relationships between concepts permits the definition of recursive patterns—this was illustrated in section 5.

On the other hand, LOOM patterns correspond to logical expressions containing a single free variable, while OPS patterns can have multiple free variables. This feature permits the OPS pattern matcher to function as the sole control-mechanism in an OPS program. The LOOM matcher is intended to be used in conjunction with a second programming language (as in, for example, [Yen *et al.*, 1988]). The design of such a language (we currently use Lisp) is a topic for future research.

Truth Maintenance

LOOM's maintenance of types for all database objects effectively maintains *truth-values* for all unary predicates applied to all objects. To achieve this behavior, the pattern-matcher is augmented by a specialized truth-maintenance subsystem.

8 Conclusions

We have described the pattern-forming language and discussed the architecture of a classification-based deductive pattern matcher, one which performs deductive inferences during the course of the pattern matching process.

Because in this scheme, patterns are concept-based, they are not isolated entities; instead, patterns are connected to other patterns via direct reference, or by implication or other logical relationships—this results in a very rich pattern-forming language.

The pattern-matcher designed for the LOOM system represents an improvement over earlier matchers that have adopted the abstraction/classification approach. Most importantly, LOOM widens the scope of what kinds of patterns can be matched by (i) incorporating backward chaining into the abstraction/classification strategy, and (ii) inferring additional matches justified by reference to non-terminological knowledge, e.g., assertions of implications between concepts. In addition, the LOOM matcher is expected to be more efficient than previous classification-based matchers because (i) its strategy of incrementally building-up an abstraction avoids abstracting features which won't be referenced by any patterns, and (ii) it eliminates the necessity for recomputing the type of a database individual each time that that individual's attributes change.

Acknowledgements

In designing the classification and pattern-matching algorithms, I benefited greatly from discussions with Tom Lipkis, Ray Bates, Bernhard Nebel, Marc Vilain, and Kai Von Luck. John Yen and Norm Sondheimer made significant contributions to the design of the LOOM language. I would like to thank Stuart Shapiro for his criticisms of an earlier draft of this paper, and Dave Brill for his help in preparing this paper.

References

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

[Brachman *et al.*, 1983] Ronald Brachman, Richard Fikes, and Hector Levesque. KRYPTON: a functional approach to knowledge representation. *IEEE Computer*, September 1983.

[Forgy, 1981] Charles L. Forgy. *OPS5 Users Manual*. Technical Report CMU-CS-81-135, Carnegie Mellon University, Pittsburg, PA, 1981.

[Luck *et al.*, 1987] K. von Luck, B. Nebel, C. Peltason, and A. Schmiedel. *The Anatomy of the BACK System*. Technical Report KIT Report 41, Technische Universitat Berlin, January 1987.

[MacGregor and Bates, 1987] Robert MacGregor and Raymond Bates. *The* LOOM *Knowledge Representation Language*. Technical Report ISI/RS-87-188, USC/Information Sciences Institute, 1987.

[Moser, 1983] M. G. Moser. An overview of NIKL, the new implementation of KL-ONE. In *Research in Natural Language Understanding*, Bolt, Beranek, and Newman, Inc., Cambridge, MA, 1983. BBN Technical Report 5421.

[Nebel and Luck, 1987] B. Nebel and K. von Luck. Issues of integration and balancing in hybrid knowledge representation systems. In K. Morik, editor, *GWAI-87*, pages 114–123, Springer, Berlin (Germany), 1987.

[Schmolze and Lipkis, 1983] James Schmolze and Thomas Lipkis. Classification in the KL-ONE knowledge representation system. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, IJCAI, 1983.

[Vilain, 1985] Marc Vilain. The restricted language architecture of a hybrid representation system. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 547–551, Los Angeles, CA, August 1985.

[Yen *et al.*, 1988] John Yen, Robert Neches, and Robert MacGregor. *Classification-based Programming: A Deep Integration of Frames and Rules*. Technical Report ISI/RR-88-213, USC/Information Sciences Institute, March 1988.