Knowledge in operation

by T. Bollinger
U. Pletat

The L_{LILOG} knowledge representation language and an inference engine to interpret it have been developed as part of the LILOG project, where new concepts for understanding natural-language texts were investigated. L_{LILOG} is a typed predicate logic whose type system has adopted the concepts of KL-One-like languages. Further language constructs allow the formulation of default and control knowledge. The inference engine for L_{LILOG} was designed as an experimental theorem prover, allowing us to investigate the behavior of various inference calculi as well as a number of search strategies. Processing with L_{LILOG} is not restricted to a propositional reasoner for logical formulas; we are also able to delegate special kinds of inferences to external deductive components. Currently, one such external reasoner for processing spatial information on the basis of analog representation is attached to the inference engine.

1. Introduction

With the realization of the first knowledge based systems, the paradigm of rule-based programming (programming by stating logical relationships instead of describing algorithms using procedural programming languages) has emerged; see, e.g., [1]. Rule-based programming plays an important role in the knowledge based systems framework primarily because of the high degree of programming productivity it makes possible.

Although knowledge based systems still use some conventional software, which is often programmed in a procedural language, a rule-based representation of the knowledge provided to a knowledge based system has become an accepted programming style. This is not surprising because the knowledge underlying a knowledge based system is the "intelligent" part of the software system, and it is often very difficult to describe this part of a knowledge based system in terms of procedural languages. Thus, abstract ways of describing knowledge are of high importance. This paper deals with the pure form of rule-based programming of the knowledge of a knowledge based system: to represent the knowledge in terms of predicate logic. In particular, we describe the logic-based knowledge representation language L_{LILOG}, together with the inference engine we developed to make the knowledge operational.

The L_{LILOG} language and the inference engine interpreting it are the outcome of research on knowledge representation and processing which was carried out within the LILOG project,[1] where new concepts for understanding natural-language tools were investigated. The monograph [2] is the final report on the LILOG project. It contains numerous articles documenting the progress of the linguistic part of the project.

Although the requirements for L_{LILOG} stem mainly from the natural-understanding context from which it emerged, the concepts offered by the language are general enough to make it applicable in other contexts of knowledge based systems as well:

[1] The LILOG project at the IBM Germany Institute for Knowledge Based Systems had as its primary goal the use and development of advanced LInguistic and LOGical methods for the understanding of German through a knowledge based programming system.

- Conceptionally L_{LILOG} is a typed predicate logic. It offers a KL-One-like type system and supports the rule-based formulation of logical axioms with the expressiveness of full first-order predicate logic.
- The language offers a means for controlling the inference engine executing the knowledge. This allows for the selection of axioms for forward- and backward-chaining tasks and the delegation of inferences to external deductive devices.
- L_{LILOG} also supports the handling of default information. As a by-product, this enables us to offer both classical negation and negation by failure.

These elements make L_{LILOG} an advanced knowledge representation language supporting the natural modeling of the knowledge underlying a knowledge based system. Its expressiveness is at the upper end of the scale of typed logics and can be best compared with the KRYPTON approach (see [3]) or Frisch's general framework for typed logics for knowledge representation (cf. [4]).

Besides developing new concepts for natural-language understanding, the LILOG project was intended to develop an experimental software environment in which the practical applicability of theoretical approaches could be evaluated. This has led to the implementation of the natural-language understanding system LEU/2 (LILOG Experimentier Umgebung, Version 2, or LILOG Experimental Environment, Version 2), of which the inference engine interpreting L_{LILOG} is an essential part.

Since LEU/2 is experimental, the L_{LILOG} inference engine was designed accordingly, to achieve a high degree of flexibility rather than high performance. This high flexibility makes it easy to exchange modules which have great influence on the inference behavior. Also, the adaptability of the inference engine to new tasks has been considered in the design.

In particular, the LILOG inference engine has the following features:

- The inference calculus determining the behavior of the inference processes has been designed as a separate module. Therefore, exchanging the calculus involves only replacing one implementation of the calculus module with another.
- The same process applies for the search strategies. This enables us to experiment with the interaction between inference calculi and strategies for traversing the search space.
- External deductive components can be connected to the inference engine to delegate special inference tasks to more efficient reasoners. In the framework of LEU/2 one such external reasoner, the depiction module processing spatial information on the basis of analog representations, has already been attached. Because we consider the invocation of external reasoners to be the application of a specific calculus rule, other external reasoners can easily be attached, e.g., a deductive database system.
- A number of parameters permit us to tune the inference behavior whenever a knowledge base of considerable size must be processed by the inference engine.

The knowledge to be processed by the LILOG inference engine is stored in the LILOG database system, and a special interface module allows the inference engine to store in and retrieve from the database system the knowledge it must process, cf. [5]. The L_{LILOG} compiler translates the L_{LILOG} source code into an internal representation on which the inference engine operates. Together with the knowledge engineering environment supporting the development of L_{LILOG} knowledge bases, the above three components form a complete knowledge representation system based on typed predicate logic.

The structure of this paper is as follows: The next section describes the L_{LILOG} knowledge representation language by explaining the language constructs offered to the knowledge engineer. We then describe the LILOG inference engine in detail. Finally, we report on our practical experience with the entire knowledge representation system. We assume that the reader has some basic understanding of predicate logic or rule-based programming and is able to recognize the benefits of type systems for programming and knowledge representation.

2. The L_{LILOG} knowledge representation language

The development of a language to represent the semantic knowledge of the LEU/2 natural-language understanding system was one of the central activities of the LILOG project from its beginning. Creating a new language became necessary because we found no suitable formalism in the scientific literature to satisfy our multiple requirements for a knowledge representation language:

- To offer a rich expressiveness for capturing a wide range of natural-language phenomena.
- To be a communication medium between the linguistic and the logical parts of the project.
- To provide a formal semantics as the basis for the LILOG inference engine.

Most formalisms suggested in the knowledge representation literature (see [6] for a compact overview) address specific aspects of knowledge representation. This reflects a situation in which formalisms emerge from investigations of particular problems aimed at understanding various specific concepts in the representation of knowledge.

In our search for an adequate knowledge representation formalism for a wide-spectrum natural-language understanding project such as LILOG, we found that the individual well-established approaches dealing with specific aspects of knowledge representation did not cover the range of features required for our purposes. What seemed necessary was to integrate various specific approaches to knowledge representation within one language. To achieve such an integration was the major impetus for the development of L_{LILOG}. In the long run, only the integration of several research directions can lead to the knowledge representation formalisms that are required in the framework of natural-language understanding and other applications of realistic size of knowledge based systems.

Against the background of the above discussion of the state of the art in knowledge representation, the definition and implementation of L_{LILOG} can be seen as one of the first attempts to integrate different streams of knowledge representation languages within one formalism.

• *The basic ideas behind L_{LILOG}*

The starting point for L_{LILOG} was the decision to create a logic-based knowledge representation formalism, since this seemed to be the most promising way to fulfill our requirement of defining a language in terms of both a formal syntax and a formal semantics.

To be more specific, we considered order-sorted predicate logic as the basis for the development of L_{LILOG} because it offers a type concept, and type systems are an essential part of modern programming languages [7–9]. Also, in formal software specification, type concepts play an important role (see [10] or [11]). And finally, types have also made their way into a particular class of AI languages: the attributive set-description languages we know (for example) from computational linguistics [12, 13], or from the KL-One family of languages [14, 15].

The role of type systems is to describe the object classes of a program or, as in our setting, of a knowledge base. Since we used the attribute-based set-description language STUF (cf. [16, 17]) in the linguistic part of the LILOG project, we decided to develop a type system in the STUF or KL-One style for L_{LILOG}. These approaches to type definitions fit nicely into the framework of an order-sorted predicate logic (cf. [18, 19]) since they also deal with a set of sorts together with a partial ordering between them, reflecting the subset relationship between the sets interpreting the sorts.

This concept of sets ordered by inclusion and common to both the semantic and the linguistic knowledge representation formalisms was the reason for integrating (parts of) STUF as the type system of the first version of L_{LILOG} [20]. Since then we have gained an improved understanding of the feature-term description languages and how to combine them with predicate logic. This enabled us to integrate concepts of languages such as KL-One [14] and Feature Logic [21] into the sort descriptions offered by L_{LILOG} [22].

The integration of attribute-based set descriptions with the framework of order-sorted predicate logic is also a step toward integrating two philosophies of knowledge representation: classical logic-based approaches and typical object-oriented approaches having their roots in artificial intelligence [3].

The world of logic has not only created formalisms for the pure mathematical way of reasoning (of which order-sorted predicate logic is a representative), but during the last ten years it has also enabled more flexible ways of human reasoning to be formalized. One of the more elaborate of these enrichments of logic to provide better models of human reasoning is nonmonotonic reasoning [23]. The basic idea of nonmonotonic reasoning may be characterized as a logical framework which offers quantifications less rigid than the usual universal ones, with the aim of arriving at a logic which still works well in situations where exceptions to general rules occur. Particularly in the framework of natural-language understanding, the required reasoning mechanisms are strongly influenced by the common sense humans apply. In this situation, standard logics are inadequate, because they were introduced to describe the reasoning processes in the precise and even formalized field of mathematics. In contrast, formal reasoning with respect to information in natural language must mimic the imprecise reasoning humans apply in everyday situations. Thus, the results that have been achieved in the field of nonmonotonic reasoning are very welcome in the framework of a project such as LILOG, and have therefore found their place in our knowledge representation language.

The features of L_{LILOG} discussed thus far have described improved facilities for representing knowledge. But knowledge is not only of the static nature suggested by the term *representation.* Knowledge that is only representable but not processable is of little practical use (at least from a computer scientist's point of view). Processing knowledge formulated within a certain logic is still a complex task which often reaches the limits of available computational power. This is because the inferential processes we use for putting knowledge in operation are still search procedures traversing immense search spaces in an uncontrolled way. The advantages of conventional programming languages over most knowledge representation languages stem mainly from the effort which must be spent on the explicit control of program execution.

In L_{LILOG} we have made some first attempts to provide control information for the theorem prover implementing the language. This comprises the possibilities of selecting logical axioms for forward chaining or backward chaining, or both reasoning modes, in order to exclude certain rules

of a knowledge base from the reasoning process. Moreover, we can delegate deductive tasks to external inference systems, which are assumed to process certain requests in a more efficient way than the main theorem prover interpreting L_{LILOG}. In the context of LEU/2, the depictional reasoner [24] for processing spatial information can be activated to perform reasoning tasks on the basis of analog representation of spatial knowledge.

In addition to these basic concepts for *expressing* knowledge, we have developed a concept for *structuring* knowledge into "knowledge packets" [25]. This permits us to separate large knowledge bases into different components, each of which deals with specific aspects of the domain of discourse to be modeled.

Following this general overview of L_{LILOG}, we provide a more detailed discussion of the particular language constructs in the next section. The style of the discussion remains informal, since formal definitions of the concepts available in L_{LILOG} may be found elsewhere [22, 26, 27].

• *Representing knowledge in L_{LILOG}*

A L_{LILOG} knowledge base is the formalization of a particular application domain over which a knowledge based system is supposed to reason. Within the LILOG project we used L_{LILOG} to model the semantic background knowledge of the LEU/2 natural-language understanding system. The knowledge developed for LEU/2 deals with tourist information about the city of Düsseldorf, such as one would expect from a tourist guide, along with general knowledge about places of interest to a visitor to a city. A particular description of the knowledge that has been modeled for LEU/2 can be found in [28].

In this section we provide an informal description of the language constructs offered by L_{LILOG}, introducing the concrete syntax of the language by means of examples. We explain what can be described by the various constructs of L_{LILOG} for which a formal definition of the semantics has been developed. The corresponding formalizations are provided elsewhere; we refer to the respective background papers whenever further information on the language may be of interest to the reader.

Knowledge items

Because L_{LILOG} is a knowledge representation language based on typed logic, type definitions and logical axioms play a major role in a L_{LILOG} knowledge base. Axioms state the logical properties of functions and predicates; thus, the declarations of these predicate and function symbols occurring in the logical formulas of a knowledge base are further knowledge entities.

Sort declarations Sort declarations introduce the object classes of a knowledge base. In order-sorted predicate logic the sort declarations are of a very simple nature, consisting simply of the name of the sort to be declared plus the embedding of the sort into the hierarchy of sorts of a knowledge base [4, 19, 29].

As an example, we have the following sort declarations recalling what can be expressed in order-sorted logic:

sort *person.*
sort *woman* < *person.*
sort *man* < *person.*

These sort declarations introduce the data domains *person*, *woman*, and *man*, and state that *woman* and *man* are contained in *person*.

L_{LILOG} offers a richer sort concept than order-sorted predicate logic does. This results from the integration into L_{LILOG} of concepts from feature-term-description languages such as STUF [17], Feature Logic [21], and KL-One [14]. Since all these languages pursue the paradigm of modeling semantic knowledge by defining sets, they offer sophisticated means for describing object classes. Thus, in contrast to order-sorted logic, where sets can only be described in terms of sort names, L_{LILOG} allows for complex descriptions of sets by means of sort expressions. These sort expressions are constructed over a collection of operators defining, for example, the intersection, the union, or the complement of object classes.

Stating subset relationships between object classes is useful not only for splitting the data domain of a knowledge base into several different object classes (generalizing one-sorted logic to many-sorted logic), but also for modeling knowledge about the world (generalizing many-sorted logic to order-sorted logic). Thus, a modern knowledge representation language should support the explicit positioning of sorts within the lattice of sort expressions (with respect to the subsumption ordering, see below) of a knowledge base. In L_{LILOG} this is achieved by means of so-called sort constraints that may be part of a sort declaration.

Requiring the subset relationship to hold between two sorts (*woman* and *person* as well as *man* and *person* in the above example) is not the only way of constraining the interpretation of sorts; we may also express that one sort is disjoint from another. A sort declaration such as

sort *person.*
sort *woman* < *person.*
sort *man* < *person*;
  disjoint *woman.*

in a L_{LILOG} knowledge base introduces the sort *man* in such a way that *man* and *woman* are disjoint subsets of *person*. The disjointness of the sorts *man* and *woman* could also be expressed as a declaration of the sort *man* by

sort *person.*
sort *woman* < *person.*
sort *man* = **and**(*person*, **not**(*woman*)).

where the sort expression **and**(*person,* **not**(*woman*)) stands for the intersection of *person* with the complement of the sort *woman*. (Note that the two ways of introducing the sort *man* lead to sort hierarchies which are not completely equivalent.) A third alternative for defining these three sorts is the following collection of sort declarations using the union operator:

sort *woman*.

sort *man*;
    disjoint *woman*.

sort *person* = **or**(*man*, *woman*).

In L_{LILOG} the equations, inequalities, and disjointness conditions between a sort name and a sort expression are called *sort constraints*, since they constrain the interpretation of the sort name being declared to the same set, a subset, or a set disjoint from the set interpreting the sort expression, respectively. For the first example above, this means that both *man* and *woman* are contained in *person*. Moreover, *man* and *woman* do not have any object in common.

The sort constraints are a means for influencing the structure of the sort hierarchy induced by the subsumption relation ≪ between the sort expressions of a knowledge base:

se ≪ **se'** *iff in any interpretation the denotation of* **se** *is a subset of the set interpreting* **se'**.

This subsumption relation has been studied for various feature-term languages [3, 13, 21].

Besides positioning a sort within the lattice of sort expressions, we can also introduce the attributes, i.e., features and roles, for a sort as part of its declaration. In the more detailed declaration of the sort *person* below, we introduce the two features *age* and *sex*, with their corresponding ranges *integer* and *sexes*:

sort *person*;
    features *age : integer*,
        sex : sexes.

These attributes allow us to speak about the age and the sex of a person. Features are functional attributes; i.e., they have a unique value for each object to which they can be applied. In certain situations it is convenient to have relational attributes available as well. This holds, e.g., for the parenthood of a man or a woman, respectively, since a person can be the father or mother of several (or possibly no) children:

sort *man* < *person*;
    roles *father_of :: person*.

sort *mother* < *person*;
    roles *mother_of :: person*.

Semantically, features stand for one-place total functions, while roles are interpreted as two-place relations [22].

In many situations we wish to introduce some specific objects of a sort when declaring it. The objects that can be introduced for a sort in its declaration are called *atoms*. Referring back to our sort *person*, we must still declare the target sort *sexes* of the feature *sex*; a reasonable definition could be the following:

sort *sexes*
    atoms *female, male*.

The atoms which can be declared for a sort are objects of that sort for which we impose a unique names assumption. For the sort *sexes*, this means that it contains (at least) the two different elements *male* and *female*.

Atoms can be used to form another kind of sort expression: intervals of integers and enumerations which appear in the definition of the sort *vehicle* below. Let us first say what kinds of vehicles we might want to consider:

sort *vehicle-type*;
    atoms *bike*, *boat*, *car*, *plane*, *train*.

Then we define vehicles as

sort *vehicle*
    features *wheels :* [*0 . . . 256*],
        doors : [*0 . . . 256*],
        type : vehicle-type,
        driver : person
        owner : person.
    roles *user :: person*.

The declaration of the sort *vehicle* makes use of intervals of integers. The intervals are sort expressions such as

$[\, 0 \cdots 256 \,]$,

which is a short form of enumerating a set of integers. In general, enumerations have the form

$\{ a_1, \cdots, a_n \}$,

where the a_i are atoms and they define the set consisting exactly of those elements interpreting the atoms mentioned in the enumeration.

Sort expressions of the form *f : se*, where *f* is a feature and *se* is some sort expression, define the subset of all those elements of the domain of the feature *f* which are mapped to the data domain defined by *se*. This mechanism is typically called feature value restriction [14]. An expression such as

type : { *car* }

stands for all vehicles for which the *type*-feature has the value *car*, and we could use this expression to define the sort of *cars* by

sort *car* = **and**(*vehicle*, *wheels* : {4}, *doors* : {2, 3, 4, 5}, *type* : {*car*});
features *body* : *body-type*.

where the body types of a car may be defined as follows:

sort *body-type*;
atoms *cabrio*, *coupe*, *hatchback*, *sedan*.

According to this definition, cars are vehicles having four wheels and two to five doors, depending on the body type. Intuitively, any car should be a vehicle, and that is exactly what the formal semantics of L_{LILOG} establishes [22]; i.e., the subsumption relationship *car* $\ll$ *vehicle* holds for these two sorts. Value restrictions involving roles may also be formulated; here, L_{LILOG} offers two variants. The sort expression

all *user teenager*

describes the sort of all vehicles whose users are teenagers, provided that any such vehicle is in use, i.e., that there is a person driving the vehicle. In many modeling situations we wish to express that, with reference to the above example, there is indeed a user of the vehicle. In order to avoid notational overhead for describing this with the **some** operator discussed below, we introduce the following useful syntactic device in the sort description features of L_{LILOG}: The sort expression

driver :: *teenager*

is a shorthand for

and(**all** *driver teenager*, **some** *driver*).

The sort declarations

sort *father* = **some** *father_of*.
sort *mother* = **some** *mother_of*.

define *fathers* and *mothers* by stating that a *father* or a *mother* is a *man* or a *woman*, respectively, such that there is at least one person of whom he or she is a parent. In other words, the sort expression **some** *mother_of* stands for all objects m of sort *person* such that there is an object c of sort *person* for which *mother_of*(m, c) holds.

The operator **agree** allows us to form the set of all objects for which two feature paths (i.e., sequences of features) have the same value. Thus, the sort expression

and(*vehicle*, **agree**(*owner*, *driver*))

characterizes vehicles owned by people who do not allow others to drive their cars. The counterpart of the agreement operator is the disagreement operator **disagree**. It defines the set of all objects such that the two feature paths involved in the sort expression have different values.

Finally, we wish to introduce sort expressions which allow us to speak about sets containing finite subsets over some base set as their elements. The sort declaration

sort *group-of-tourists* < *person*$^+$.

defines the sort of *group-of-tourists* as a set containing finite sets of *persons* as elements. In several modeling situations it is convenient to consider a base set and finite subsets within one set. Such classes of objects could be described by the union **or**(*person*, *person*$^+$), if we take the sort *person* as the base set. This union can be abbreviated by another operator, *; thus, for the sort *person* we obtain *person** = **or**(*person*, *person*$^+$).

In summary, the sort descriptions of L_{LILOG} have integrated concepts from order-sorted predicate logic and KL-One-like languages. This supports an object-oriented description of the data domain of a knowledge base in the sense that we are able to speak about objects and their attributes. The formal semantics of these language constructs interprets sort expressions as sets (of a certain structure according to the set-forming operators used in the expression), features as total functions, and roles as relations [22], allowing us to deal with subsumption relationships between sort expressions in a natural way. An interesting, although very natural, phenomenon of the semantics for the sort descriptions of L_{LILOG} is that we can use an important property of object-oriented languages at no cost: the concept of multiple inheritance. The inheritance mechanism works in two directions: Features and attributes are inherited downward in the lattice of sort expressions, while objects of a sort are inherited upward. This results from two simple mathematical properties: A function $F : D \rightarrow R$ can also be applied to any subset $D' \subseteq D$ of its domain; thus we say that F is inherited down to any subset D' of D. In the context of the L_{LILOG} knowledge terms introduced so far, this means, e.g., that the *age*-feature of a *person* may also be evaluated for any *mother*, since *mother* $\ll$ *person*; i.e., *mother* is subsumed by *person*, formalizing that any mother is (of course) also a person. On the other hand, the subset relation between the sets interpreting the sorts causes any object of a sort s to be also an element of the sort s', if s is subsumed by s'. In this sense we obtain the upward inheritance of atoms (and also other objects we describe below) of a sort. All this is straightforward: We simply exploit the natural properties of subset relations among sets and the element relation between objects and sets.

Function declarations While the sort declarations of a knowledge base introduce its sort hierarchy, or (to use a different terminology) the taxonomy of its data domains, the formulation of the logical axioms of a knowledge base

requires further declarations: We need to know the relations to which the axioms refer. To introduce these relations between the data classes of the knowledge base is the purpose of the function and predicate declarations to be discussed next. We have already introduced simple kinds of function and predicate declarations: the features and roles attached to a sort. Their simplicity results from the fact that they can relate only two sorts. In general, both functions and predicates of arbitrary arity are desirable.

A function declaration

function *F (argname : se_1, ⋯ , $argname_n$: se_n) → se.*

in L_{LILOG} states that in the domain modeled by the knowledge base, the object classes represented by the sort expressions se_i are functionally related to the set *se*, and that this relationship bears the name *F*.

Within the domain of traveling functions is a convenient means for modeling knowledge. A typical functional relationship is that of seat allocation, assigning a seat within a vehicle to each of the passengers. Assuming for simplicity that any seat can be identified by an integer number, we obtain the following function declaration as part of our knowledge base:

function *seat-allocation (passenger : person, carrier : vehicle) → integer.*

This declaration exhibits another feature of the concrete syntax of L_{LILOG}: The arguments of a function (and also those of a predicate) need not be identified by their position in the argument list; instead, we support the more flexible method of explicit naming of argument positions.

Choosing a function to represent seat allocation is a good means for expressing that each passenger in a vehicle has a unique seat. Unfortunately, seats may be overbooked; this can also be captured adequately by modeling the seat allocation with a function, since functions are in general not injective. Thus, it may happen that both Mr. Miller and Mr. Smith are allotted to the same seat No. 15 in a Boeing 737; i.e., we have

seat-allocation(passenger : Mister_Miller, carrier : Boeing_737) = 15

and

seat-allocation(carrier : Boeing_737, passenger : Mister_Smith) = 15.

Some functions have no arguments. Such functions are considered as elements of their target sort. The syntax of L_{LILOG} offers such nullary functions as *constants* or *reference objects*, so we may have further declarations such as

constant *Mister_Miller : man.*

and

refo *Mister_Smith : man.*

and

refo *Boeing_737 : type :* **and**(*vehicle, { plane }*).

as part of a knowledge base. Logically, there is no difference between constants and reference objects. However, in the natural-language understanding context of the LILOG project, from which L_{LILOG} emerged, reference objects (refos, for short) are distinguished as constants. The type of a constant or a reference object may be given by an arbitrary sort expression. Thus, we want to speak about two distinguished sportscars such as

constant *Porsche_911 :*
and(*car, doors : { 2 }, body-type : { cabrio }*).

and

refo *ferrari_dino :*
and(*car, doors : { 2 }, body-type : { coupe }*).

L_{LILOG} offers several built-in functions for the arithmetic operations. Thus, the following functions are part of any knowledge base:

function *(integer + integer) → integer.*

function *(integer − integer) → integer.*

function *(integer * integer) → integer.*

function *(integer / integer) → integer.*

function *− (integer) → integer.*

Since in L_{LILOG} we may have data domains containing sets over a certain base set as elements, we would like to be able to form unions of these sets or to create the set containing two elements of the base set. This is the idea behind the following built-in function:

function *(top* & top*) → top^+.*

The sort expression *top** abbreviates the union **or**(*top*, top^+), which is the top element of the entire lattice of sort expressions over a knowledge base. For example, the function & can be used to form the set *john & mary*, which is a set-object of the type $person^+$ and contains the two elements *john* and *mary*.

The ability to declare function symbols and define them via equations allows us to use L_{LILOG} like the kernel of a functional programming language standing in the tradition of abstract data type specification; see [10, 30].

Predicate declarations A predicate declaration

predicate *R (argname$_1$: se_1, ⋯ , argname$_n$: se_n).*

appearing in a L_{LILOG} knowledge base tells us that we have an arbitrary relation named *R* between the sets interpreting the sort expressions se_i; i.e., there is no functional dependency between the arguments of the predicates.

An important predicate that should occur in any knowledge base on traveling is the following:

predicate *travel (who : person*, fr : location, to : location, with : vehicle).*

The declaration of the predicate *travel* states that a relation *travel* exists among the sorts *person**, *location*, and *vehicle* which can tell us who is traveling from where to where using which vehicle. Choosing the sort expression *person** expresses that we wish to speak about traveling groups of people and individual travelers in a uniform way.

Several built-in predicates are available in any knowledge base, e.g., the comparison predicates between integers and an equality predicate.

To test the subset or membership relation between objects of the sort *top** we have the predicate

predicate *(top** **in** *top*).*

made available as an L_{LILOG} built-in. Using the constants *john* and *mary* of sort *person* and the union operator &, we observe that

mary **in** *john & mary*

is true, since *mary* is an element of the set *john & mary*. Also,

mary & john **in** *john & mary*

holds, because *john & mary* and *mary & john* denote the same set-object.

Finally, we have an additional collection of useful predicates which are available as built-ins of L_{LILOG}: the so-called sort-predicates, which could be declared as follows:

predicate *{se} (top*).*

for each sort expression *se*; i.e., we may use an arbitrary sort expression as a unary predicate.

Using sort expressions as predicates, we are able to state sort memberships that cannot be expressed by the declaration of constants, refos, or functions. This holds, for example, for conditional memberships such as the following: If somebody travels more than 50 000 miles per year, he is a globetrotter, where *globetrotter* is a subsort of *person*:

sort *globetrotter < person.*
sort *year.*
predicate *travels (who : person, distance : int, period : year).*

axiom *globetrotters*
  forall *D : distance, P : person, Y : year;*
  travels(period : Y, distance : D, who : P)
  and *D ≥ 50.000*
    →
  { globetrotter } (P).

Having introduced the first logical axioms, we complete the discussion of function and predicate declarations and take a more detailed look at the logical axioms that may occur in a knowledge base.

Axioms of a knowledge base The declarations of sorts, functions, and predicates we have discussed thus far have introduced the basic building blocks for formulating the logical axioms of a L_{LILOG} knowledge base. As in any logic, the logical axioms express which objects of which sort are related by the relations (i.e., functions and predicates) declared within the knowledge base.

The expressive power of the axioms that may occur within a L_{LILOG} knowledge base is that of full first-order predicate logic. Because of the knowledge based system context for which L_{LILOG} was designed, we have chosen a rule-oriented notation for the logical axioms of the following form:

axiom ⟨ *axiom-id* ⟩ ⟨ *quantifications* ⟩*;*
  ⟨ *premise* ⟩ → ⟨ *conclusion* ⟩

Thus, every axiom has a name; next we have the quantifications introducing the variables occurring in the axiom, together with the quantifier to which the respective variable is bound. Both universal and existential quantifiers are offered; moreover, further default quantifiers are possible as well (see below for the discussion of how to handle defaults in L_{LILOG}). Then we have the body of an axiom, which is basically an implication where the premise may be a conjunction of disjunctions of literals, while the conclusion may be a disjunction of conjunctions of literals. Compared to the clausal form for logical formulas, which is often used in resolution-style theorem proving [31] and also for logic programming [32], this is a moderate generalization which has the following advantages: 1) various quantifiers are available to the knowledge engineer; 2) writing the axioms as implications supports the "what follows from what" intuition which is often used in the framework of operational logic; and 3) no normalizations (but skolemization) are necessary for processing these formulas as long as we use the model elimination calculus for generalized clauses, which is described in [33, 34].

This general structure of the logical axioms allows us to formulate ground facts such as

axiom *john-and-mary-travel*
  travel(who : john & mary, fr : Los_Angeles,
    to : San_Francisco, with : Porsche_911).

expressing that both John and Mary travel from Los Angeles to San Francisco with their Porsche, and rules such as

axiom *group-member :*
  forall *M : person, G : person*, F, T : location, V : vehicle;*

travel(who : G, fr : F, to : T, with : V) **and**
M in *G*
→ *travel(who : M, fr : F, to : T, with : V)*

expressing that if a group of people travels with some vehicle, then any member of the group uses the same vehicle.

While the above two rules involved either no or only universal quantification, the concrete syntax of L_{LILOG} also supports the use of the existential quantifier, enabling us to formulate axioms such as

axiom *lonely-traveler :*
exist *P : person*;
travel(who : P, fr : San_Francisco,
to : Los_Angeles, with Porsche_911).

expressing that somebody is traveling from San Francisco to Los Angeles.

Defaults in L_{LILOG} The quantifications appearing in the axioms of a knowledge base are also the means for formulating default knowledge. The basic idea behind this approach is that a default proposition can be seen as a special form of universally quantified formula, stating that the proposition does not strictly hold for every object of the sort of the quantified variable, but rather permits exceptions.

Assuming a predicate

predicate *uses(who : person, which : road).*

and reference objects

refo *Highway_1 : road.*

refo *Highway_101 : road.*

we can state that people typically use either Highway 1 or Highway 101 when going from San Francisco to Los Angeles by the following axiom involving a default quantification:

axiom *typical-route :*
o_default *P : person*, **forall** *V : vehicle;*
travel(who : P, fr : San_Francisco,
to : Los_Angeles, with : V)
→ *uses(who : P, which : Highway_1)* **or**
uses(who : P, which : Highway_101)

The quantification **o_default** *P : person* states that the axiom *typical-route* involves what we call an optimistic default; i.e., applying an axiom containing optimistic default quantification does not trigger any consistency checking of the conclusions (of the applied rules involving the default) immediately after the inference process. The more standard situation of performing the consistency check before the application of a rule can be specified by the so-called pessimistic defaults, for which we foresee quantifications such as **p_default** *P : person*. The effect of this differentiation is that the LILOG inference engine uses rules involving an optimistic default as if the corresponding variable had been universally quantified. However, when it becomes necessary to revise knowledge relying on default information, formulas that have been derived using optimistic defaults may be withdrawn, while this is not the case for formulas relying on hard information only (see [35] for a description of the truth-maintenance system which handles this default reasoning approach in the LILOG inference engine. The pessimistic defaults behave like normal defaults by applying default rules only in situations causing no contradictions [36]. From an operational point of view, this means that we must verify that no contradiction occurs when performing an inference step involving a pessimistic default. Furthermore, the default mechanism of L_{LILOG} offers the concept of graded qualities of default knowledge [26, 35]. While in the early approaches to nonmonotonic reasoning, the quality scale for knowledge relying on default information consisted only of the two degrees *hard* and *default* (cf. [36], [37]), more recent approaches have implicitly suggested the use of an arbitrary partial ordering as the quality scale [38]. In our setting of an order-sorted logic, it seemed to be quite natural to use the partial ordering of the sort expressions as the scale for measuring the quality of default information described in L_{LILOG} knowledge bases. The basic idea is that information involving smaller sorts is of higher quality than information stated for larger sorts.

The default reasoning concept of L_{LILOG} allows for elegant formulations of some typical puzzles which have become rather famous in the meantime: the question whether *tweety* can fly, and the question whether *nixon* is a pacifist.

Let us treat *tweety* first and assume the following knowledge entities in our knowledge base:

sort *bird.*
sort *large-bird < bird.*
predicate *fly(who:bird).*

refo *tweety : large-bird.*
axiom *flying-birds*
p_default *B : bird;*
fly(who:B).
axiom *non-flying-birds*
p_default *LB : large-bird;*
not *fly(who:LB).*

Then our knowledge implies both

fly(tweety)

and

¬*fly(tweety)*

but since we have *large-bird* < *bird*, the information that *tweety* does not fly is of better quality than the information asserting that *tweety* flies. (This analysis of the quality of default information is done by the truth-maintenance system [35].) Therefore, we do not encounter an inconsistency, because the TMS discards the worse information that tweety flies.

Of course, most real reasoning problems are not that easy, because the sort hierarchy of a L_{LILOG} knowledge base is not a total but a partial ordering. Taking a partial ordering as the quality scale for default information leaves us with the problem of what to do with inconsistent statements for which the quality of the conflicting results cannot be effectively compared with respect to the underlying sort hierarchy. This is discussed in the framework of the Nixon diamond formulated in L_{LILOG} below:

sort *quaker.*

sort *republican.*

sort *pacifist.*

refo *nixon :* **and**(*quaker, republican*).

axiom *peaceful quakers*
p_default *Q : quaker;* {*pacifist*}(*Q*).

axiom *brave republicans :*
p_default *Q : republican;* **not** {*pacifist*}(*Q*).

Since *nixon* is both a *quaker* and a *republican*, we obtain both *pacifist(nixon)* and ¬*pacifist(nixon)*. Because the quality of the default information leading to this contradiction cannot be compared, we take a skeptical approach and believe neither in *nixon* being a pacifist nor in *nixon* not being a pacifist.

The availability of defaults allows the knowledge engineer to make use of both classical negation (since we deal with full first-order predicate logic) and negation by failure (since we can express that "something does not hold unless stated otherwise" by means of a default axiom).

For a more detailed discussion of the concepts of nonmonotonicity as part of L_{LILOG}, see [26, 35].

Controlling inferences in L_{LILOG} Since L_{LILOG} is implemented by an inference engine interpreting the language, we (as any theorem prover does) face the problem of having to traverse large search spaces. In the natural-language understanding context of LEU/2, the size of these search spaces does not result particularly from the depth of the proofs to be performed, but from the fact that we must deal with rather large knowledge bases: The background knowledge base of LEU/2 consists, for example, of about 600 sort declarations and about 300 facts and rules. Thus, means for excluding parts of the axiomatic knowledge for certain inference tasks are welcome.

The application of L_{LILOG} in the natural-language understanding framework suggests both forward- and backward-chaining tasks over a knowledge base. This is due to the two operation modes of the LEU/2 system: When texts have to be understood by the system, we want to draw conclusions from the information contained in the texts with respect to the background knowledge provided to LEU/2. This is a typical situation in which the forward-chaining mode of the inference engine is the main inference task to be used. In many situations we know in advance that only specific axioms of the knowledge base need be used for these forward-chaining tasks. Analogously, we can say the same about the backward-chaining reasoning mode, i.e., that there are certain axioms in the knowledge base for which the knowledge engineer knows in advance that they need only be used in the problem-solving mode, which is the basic inference task when questions about the contents of texts must be answered by LEU/2. This gives us two classes of axioms within a knowledge base:

1. Axioms which are applied for knowledge base extension, i.e., in the forward-chaining mode of the inference engine.
2. Axioms which are applied for problem solving, i.e., in the backward-chaining mode of the inference engine.

Of course, these two classes of axioms need not be disjoint. The technical concept for qualifying axioms to be relevant for backward or forward inferences is to use *entry points*, which can be attached to the literals occurring in an axiom. To make an axiom available to forward-chaining tasks, one of the literals in the premise of the axiom must be tagged with an entry point. The backward-chaining tasks may use all axioms in which either no entry point occurs or an entry point is provided for one of the literals in the conclusion. Axioms that are to be used for both forward- and backward-chaining tasks must have literals tagged with entry points both in their premise and in their conclusion.

To illustrate the effect of marking literals by entry points, we return to our axiom *group-member* from above:

axiom *group-member-bwc*
forall *M : person, G : person*, F, T : location, V : vehicle; travel(who : G, fr : F, to : T, with : V)* **and**
M in *G*
→
EP travel(who : M, fr : F, to : T, with : V).

The entry point specified for the literal

travel(who : M, fr : F, to : T, with : V)

qualifies the axiom *group-member-bwc* as one to be used for backward inferences only.

By placing an entry point on a literal in the premise of the axiom, we can qualify it to be usable in forward inferences:

axiom *group-member-fwc*
forall *M : person, G : person*, F, T : location, V : vehicle;*
EP travel(who : G, fr : F, to : T, with : V) **and**
M in *G*
→
travel(who : M, fr : F, to : T, with : V).

The entry point specified for the literal

travel(who : G, fr : F, to : T, with : V)

makes the axiom available for forward inferences which have been triggered by a formula containing a literal that unifies with the labeled literal. Thus, we may use the axiom

axiom *john-and-mary-travel*
travel(who : john & mary, fr : Los_Angeles,
to : San_Francisco, with : Porsche_911).

as a trigger formula and derive that both John and Mary travel from Los Angeles to San Francisco by means of forward inferences. Note that the formula *group-member-bwc* is blocked for application in forward-chaining mode.

The second means for controlling the inference engine is to delegate inference tasks to an external deductive component. In particular, this means that in L_{LILOG} we are able to state that certain literals must be evaluated outside the theorem prover that interprets L_{LILOG}. Currently only one such external reasoner is supported: the depiction module for processing spatial information on the basis of cell matrices [24, 39]. The depiction module is a special evaluator for predicates (appearing in literals) such as *close-to* defining when some object is located close to another one. A typical invocation of the depiction module would be

depic *close-to(what-is : church, close-to-what : bicycle)*

indicating that the analysis should switch from the theorem prover to the depiction module and process the above literal there. A literal can be processed externally in two ways, corresponding to the basic inference modes of knowledge base extension and problem solving:

1. In the imagination mode, the depiction module is provided with some literal and extends its internal knowledge base with the information given by the literal. This corresponds to the knowledge base extension; whenever a depic-literal such as the one above appears in the conclusion of an axiom, the imagination mode of the depiction module is triggered with this literal.

   This general idea about the imagination mode of the depictional reasoner is illustrated by the following axiom:

   axiom *bike-and-church*
   forall *B : bicycle, C : church*
   EP distance(between-object1 : B, and-object2 : C) ≤ *1m*
   → **depic** *close-to(what-is : B, close-to-what : C).*

   When the axiom is to be processed by the inference engine, it causes the depiction module to store the fact that a bicycle is close to a church whenever it is possible to prove that the distance between the bicycle and the church is less than one meter. We can see from this axiom that its application makes sense only in a forward-reasoning mode, which is why we qualify the rule as applicable only in forward-chaining tasks by placing an entry point on the premise of the axiom.
2. The inspection mode of the depiction module corresponds to the problem-solving inference task. When the inference engine must process a depic-literal in backward-chaining mode, it does not search the set of axioms for a complementary literal in order to make a "classical" resolution step over that literal. Instead, the literal is passed to the depiction module in order to verify it and to return substitutions as the solution for that literal to the theorem prover.

   For the goal formula below, this means that the inference engine will delegate the finding of a solution to the depictional reasoner and not try to solve the goal according to the knowledge in the propositional knowledge base:

   goal *what-is-close-to-what*
   exists *OB1 : object, OB2 : object;*
   *?-***depic** *close-to(what-is : OB1, close-to-what : OB2).*

The mechanisms of imagination and inspection of the depictional reasoner for processing spatial knowledge must be considered special instances of increasing the amount of knowledge of an external reasoner and accessing the knowledge of such a special-purpose knowledge processor. The integration of the spatial reasoner could be taken as the design for including other external inference engines, such as relational databases or a logic programming system, in the LILOG knowledge representation approach.

Summarizing, the axiomatic part of a L_{LILOG} knowledge base allows us to formulate the logical relationships between the objects of a knowledge base in a rather sophisticated way. The generalized clausal form in which axioms may be presented supports the "what follows from

what" intuition that has proved very successful in languages such as Prolog and other rule-based programming languages for knowledge based systems. Moreover, the default logic and the means for controlling the inference processes can be used with little notational overhead and exhibit a clear underlying intuition.

3. The LILOG inference engine

The development of an inference engine making the L_{LILOG} knowledge representation language operational has been strongly influenced by the natural-language understanding context of the LILOG project. However, in the architecture of the inference engine we have anticipated the use of its logical kernel for a wider spectrum of applications as well.

• *Design objectives for the inference engine*

In the framework of the LEU/2 natural-language understanding system, knowledge represented in L_{LILOG} may stem from two sources:

- Our knowledge engineers have developed the background knowledge of LEU/2 about touring through a city in which we find streets, museums, restaurants, and other things of interest. Moreover, this background knowledge contains specific information about the city of Düsseldorf.
- The LEU/2 system interprets German texts describing, for example, specific museums, or the location of department stores in Düsseldorf, by constructing L_{LILOG} representations of the information provided in such texts.

The natural-language understanding capabilities of LEU/2 require that the knowledge provided to the system be made operational by an inference engine for various purposes. Knowledge extracted from German texts must be combined with the background knowledge by means of forward inferences (this corresponds to the text-input mode of LEU/2). Of course, we can also query the knowledge available to LEU/2 in natural language and obtain natural-language answers to our question (this corresponds to the query mode of LEU/2).

Apart from these two overall requirements, the linguistic capabilities of a natural-language understanding system can be improved by supporting, e.g., the linguistic analysis with background knowledge about the application domain. Typical tasks that benefit from the evaluation of background knowledge are the disambiguation of different readings of a natural-language sentence, or the resolution of anaphoric references between a pronoun and the explicitly mentioned referent to which it refers. Another component which can benefit from the services of the inference engine is the dialog component of the LEU/2 system. Here the inference engine supports the generation of cooperative natural-language answers to queries posed against the knowledge of LEU/2.

The above discussion shows that in the natural-language understanding framework of LEU/2, the inference engine must offer a variety of *application-dependent inference tasks.*

On the other hand, L_{LILOG} is a typed logic that does not implicitly recognize the applications for which it is being used. However, any logic suggests certain inference tasks that can be performed over the knowledge represented in this logic. Let us call these inference tasks *application-independent*. Since L_{LILOG} is a typed logic, the following three logical inference tasks are realized by the inference engine no matter what application it is supposed to support:

1. The classical logical inference task of solving the problem of whether a goal formula follows from the formulas given in a knowledge base.
2. In addition to the classical backward-chaining inference task, a forward chainer (with consistency checking capabilities) is of considerable interest for combining new pieces of knowledge with existing knowledge.
3. The sort language of L_{LILOG} constitutes a (sub)logic in itself, suggesting inference tasks such as testing the validity of the subsumption relation for two sort expressions or computing their greatest lower bound. Thus, special sort-processing capabilities are available as a logical inference task.

These application-independent inference tasks can be considered to be a second level of inferential capability (realizing the application-dependent inference tasks) offered by the inference engine of LEU/2.

Logics are typically made operational by means of theorem provers. Classical theorem provers implement pure first-order predicate logic in terms of a proof procedure for testing whether a set of clauses is inconsistent [40–42]. Very often they are optimized toward performing this basic inference task efficiently [43–45]. In considering the variety of tasks the LEU/2 inference engine must deal with, it is clear that the functionality of such basic theorem provers is not sufficient for our purposes. Of course, the inference tasks we have in mind also require the basic theorem-proving function of showing that a set of clauses is inconsistent. According to the application context given by the LEU/2 natural-language understanding system, these tasks are executed in specific contexts and may require different settings for the basic proof procedure. Therefore, we aim at a parameterizable theorem prover serving as the basic inference algorithm, which is adaptable to the various specific inference tasks it is supposed to solve. The *general proof procedure* meets

these requirements and can be considered as the "heart" of the inference engine. The second basic reasoning algorithm is an *inconsistency checker* for sort expressions, which processes the sort information of a L_{LILOG} knowledge base. **Figure 1** illustrates the vertical structure of the LILOG inference engine outlined thus far.

The discussion above has shown the different levels of abstraction at which knowledge formulated in L_{LILOG} can be made operational within LEU/2. The basic idea of introducing the vertical structuring of the inference engine is to make a clear distinction between application-dependent inference tasks and the logical inference tasks that are supported independently of the application. These logical inference tasks themselves form a higher level of inferential capability than the basic inference algorithms represented by the general proof procedure and the inconsistency checker.

In addition to the vertical structure of the LILOG inference engine, we have imposed a horizontal structure on the logical level of the inference engine, which has been strongly influenced by further overall objectives for LEU/2. One objective for the development of LEU/2 was to create an experimental environment offering basic processing modules needed for natural-language understanding in such a way that extensions of these modules and the integration of new components into LEU/2 are easily possible. This decision was made in order to provide an experimental system for testing the feasibility of theoretical solutions in a given software environment. Of course, this overall objective for LEU/2 applies to the knowledge representation and processing activities of the LILOG project as well.

For processing L_{LILOG} (and knowledge processing in general), several interesting questions do need practical experience, requiring an implementation of the knowledge representation language:

- To what extent does the inference behavior of the various logical inference tasks depend on the inference calculus being used for the inference task?
- Can specialized external reasoners improve the inferential capacities of the inference engine?
- How do inference calculi and strategies for traversing the search space interact? Which search strategy best fits which calculus?
- Does one inference calculus fit some specific knowledge base better than another?
- Can we tailor the behavior of the inference engine by setting certain parameters and options so that inference behavior improves for a specific knowledge base?

To experiment with the inference behavior of our inference engine and answer such general questions, we had to establish a very clean module structure for the

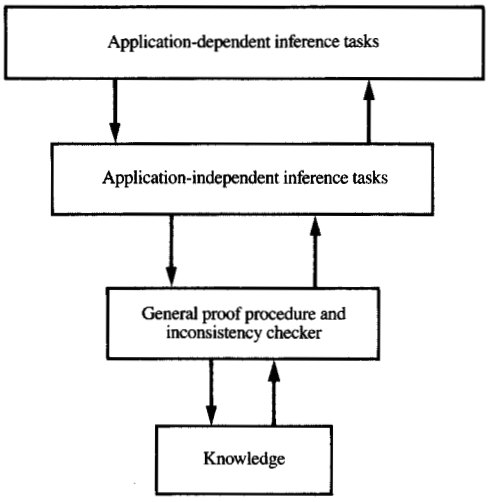

Figure 1

Vertical structure of the LILOG inference engine.

inference engine as a whole and the general proof procedure in particular. This led to an architecture in which, for example, the general proof procedure is able to make explicit reference to the inference calculus and search strategy it is supposed to use. Thus, the inference calculus and the search strategy became separate modules of which different realizations can be interchanged easily in order to impose different inference behaviors on the general proof procedure. The high degree of flexibility that can be achieved by such an architecture has its price in terms of lower efficiency, but for a system serving as an environment for experimenting with inference calculi, search strategies, etc., we are willing to pay this price.

The implementation of LEU/2 has already shown the practical usefulness of the modular architecture of the logical level of the inference engine. The user is already able to choose among several search strategies such as depth-first or breadth-first, and the reasoning capacities can be enhanced by adding specialized calculus rules, e.g., for processing sort literals, when the contents of the knowledge base require this.

The rest of this section discusses the three layers of the inference engine. First we address the application-dependent layer: the user interface. Then the logic interface, representing the application-independent inference tasks, is discussed; and finally we give a more detailed view of the low-level inference mechanisms

represented by the general proof procedure and the sort-processing algorithms.

• *User interface*

The user interface level of the inference engine offers the application-dependent inference tasks. Since some of the logical inference tasks provided by the inference engine are also of interest to some of its users within LEU/2, the user interface is the point at which to access them. Below we describe typical situations in which various modules of the LEU/2 system will access the inference engine.

Inference tasks supporting text understanding Having constructed the semantic representation of a sentence, we must integrate the new knowledge into the previously acquired text knowledge and background knowledge available to LEU/2. This is done by checking whether the new knowledge is consistent with the existing knowledge and by performing forward inferences in order to deduce implicit knowledge. Consistency checks and the execution of forward inferences are part of the functionality offered by the logic machine; they are described in more detail in the next subsection.

The resolution of anaphoric references is one part of the interpretation process that constructs L_{LILOG} representations for German texts. It must determine whether there is an antecedent in the text to which a pronoun may refer. One way of making this decision is to compare the sort of the object representing the antecedent with the sort attached to the object representing the pronoun. The linguistic algorithms will discard any of the candidate antecedents whose sort is incompatible with the sort of the pronoun; i.e., the greatest lower bound of the sort of the antecedent and that of the pronoun is the empty set. Computing the greatest lower bound is an inference task the logic machine offers to its users within LEU/2.

Inference tasks supporting question answering In the question-answering mode of LEU/2, we are able to have a natural-language dialog with the system about the knowledge it has acquired during previous text-understanding phases. One objective here is to achieve a dialog behavior which can be called *cooperative*. That is to say, instead of verbalizing the logical answers to the questions posed to the system in a straightforward way, LEU/2 is expected to provide more informative, natural-language answers.

Basically there are two question types whose logical representations must be handled by the inference engine: *yes/no-* and *wh-questions* (questions beginning with *who*, *what*, *which*, and *where*). For yes/no-questions, it is sufficient to try to prove the goal formula representing the question, and, if the proof does not succeed, the negated goal as well in order to provide the natural-language answers *Yes*, *No*, and *I don't know* instead of being able to say only *Yes* or *I don't know*. For wh-questions such as "*Which museum is open at 11 o'clock*?" one is additionally interested in those instantiations of a specific variable which represent the logical answers to a question.

To achieve cooperative behavior, concepts such as over-answering and presupposition handling must be realized as part of the dialog component, avoiding simple "*Yes*" or "*No*" answers to a question. For example, the answer "*No*" to the question "*Does the museum open at 9 o'clock?*" is not very helpful; a better answer would be "*No, the museum opens at 10 o'clock.*"

To support such features with the inference engine, some extensions of the general proof procedure have been made, leading to more elegant solutions than would *ad hoc* ideas based on standard approaches. The standard solution for producing the answer "*No, the museum opens at 10 o'clock*" is first to submit the original goal representing the yes/no-question. Then, if this proof fails, the "9 o'clock term" in the question goal can be replaced with a variable, and the more general goal resubmitted to the inference engine.

We are able to realize this within one query by immediately replacing the 9 o'clock term with a variable in the question goal and by telling the inference engine to prefer solutions which instantiate the variable with terms that are compatible with the 9 o'clock term. If the compatibility constraint set up by the 9 o'clock term cannot be satisfied, we drop it and search for further solutions to our modified goal; if we find the 10 o'clock term, it becomes the result of the inference process. We give a more detailed description of this method in the subsection on the general proof procedure.

Detecting violated presuppositions is another means for generating cooperative answers. Assuming that no Picasso museum exists in Düsseldorf, a good answer to the question "*Is the Picasso museum open on Monday?*" would be "*There is no Picasso museum*," instead of "*No*," or "*I don't know*." This is because the latter answers presuppose that there is a Picasso museum which, however, is closed on Mondays. The semantic representation of the original question may be split into a conjunction of two subgoals corresponding to the questions *Q1:* "*Is there a Picasso museum?*" and *Q2:* "*Is it open on Monday?*" This conjunction of subgoals is submitted to the inference engine, and it is asked to prove first Q1 and then Q2. The inference engine cannot find a proof of Q1 and reports this to the dialog component, which is now able to give the desired cooperative answer.

In general, to produce the logical answers to such questions, the subgoals of the question goal must be processed in a given order—the presupposition order. If the proof of the entire question fails, those subgoals that

could be proved, as well as the first nonproven subgoal, are returned to the dialog component, enabling it to give its cooperative answer.

From an abstract logical point of view, the answers to wh-questions are terms instantiating a variable of the question goal. One task of natural-language generation is to verbalize these terms by finding an appropriate succinct description distinguishing it from other objects of the same category (i.e., sort). Thus, the inference engine may be asked to compute all attributes for which a certain object has distinct values compared to other objects of the same sort.

For example, if we intend to speak about a specific pencil among a number of others, it may happen that the one we are interested in is the only red one. If there is a *color*-feature, its value is *red* for the red pencil and different from red for the other pencils; i.e., *color* is a discriminative attribute for the red pencil, and "*red pencil*" is an appropriate verbalization of it. We refer to [46], which describes the language-generation component of the LEU/2 system.

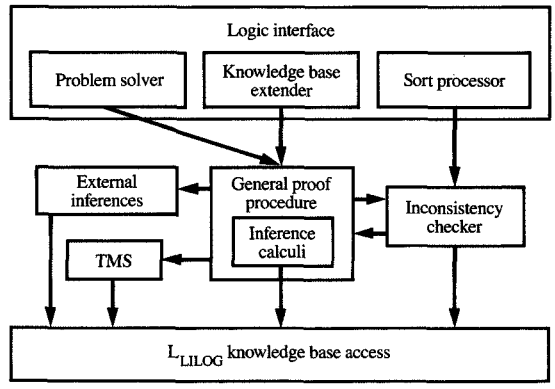

Figure 2

Architecture of the LILOG logic machine.

- *Logic machine*

The logic machine of the LILOG inference engine performs the logical, i.e., application-independent, inference tasks, upon which the more complex, application-dependent inference tasks of the user interface are realized. Here, then, we find the inference services one would expect from an implementation of L_{LILOG} as a stand-alone knowledge representation language. The general structure of the logic machine is shown in **Figure 2.**

The three subcomponents for the execution of logical inference tasks are the problem solver, the knowledge base extender, and the sort processor. The first two subcomponents use the general proof procedure, since their major concern is to process the knowledge provided in terms of the axioms of a L_{LILOG} knowledge base. The general proof procedure contains the inference calculus (together with the search strategy) as a fundamental submodule.

Since the truth-maintenance system and the external deductive component (a reasoner performing special deductions) are triggered by special calculus rules, the general proof procedure is also the point at which these components are attached to the inference engine.

The sort processor offers inference tasks that evaluate the knowledge represented in the sort declarations of a knowledge base. These inference tasks are realized in terms of an inconsistency checker for sort expressions.

Access to the knowledge bases is realized by a special interface module connecting the inference engine with the database system that stores the compiled L_{LILOG} code on which the inference engine operates.

Logic interface

The logic interface is, so to speak, the entrance through which one can access the logical inference tasks offered by the inference engine. Its three doors are described below.

Problem solver The problem solver can be considered as a generalized theorem prover whose essential task is to prove L_{LILOG} goal formulas, given in terms of conjunctions of disjunctions of literals.

The problem solver itself essentially performs some initialization tasks comprising the negation of the goal formula and the setting of various parameters directing the search within the general proof procedure. The negated goal is then passed to the general proof procedure for finding a proof of the goal. Certain features distinguish the problem solver (with the general proof procedure) from ordinary theorem provers. First, there are many options available, allowing us to set various parameters for the execution of an inference task. In addition to options for specifying the search depth and the number of results (such as 1 for a yes/no-question or "all" for wh-questions), we may, for example, specify how to represent the result of an inference process: The simplest form is to return just a success or failure message. Alternatively, one may request the inference result to be presented as instantiations of the goal formula, or in terms of valid substitutions for the variables in the goal. An option more specific to the LILOG inference engine allows us to focus the search on particular instantiations of certain variables; this is useful for realizing some kinds of over-answering (see the subsection on the user interface). Another special

fc option deals with handling implicit presuppositions when asking natural-language questions. For this, the problem solver receives a list of L_{LILOG} formulas, which is interpreted as a conjunction and represents the presupposition order. The task is to prove a maximal number of these L_{LILOG} formulas in the given (presupposition) order. The formulas for which the proof fails represent the violated presuppositions.

Knowledge base extender The knowledge base extender offers the inference tasks of executing forward inferences in connection with performing consistency checks. To check the consistency of a formula with respect to a given knowledge base, it is sufficient to pass the formula itself to the general proof procedure, which (being a refutation procedure) detects an inconsistency by deducing the empty clause.

The execution of forward inferences is triggered by the input facts of the knowledge base extender. These facts are applied to forward (-chaining) rules by trying to resolve them with the premises that are marked by an entry point. The results are passed to the general proof procedure for "resolving away" any remaining premise literals. If this succeeds, the instances of the conclusions constitute valid facts deduced by a forward inference step. In the axiom

axiom *fw-axiom1*
forall *x, y, z : top*
EP P(x, y) **and** *Q(y, z)*
→
R(x, z).

the first premise is marked by an entry point. Thus, a forward inference step can be initiated by the fact $P(a, b)$; if the second literal of the forward rule can be proven by $Q(b, c)$, the literal $R(a, c)$ has been deduced by a forward inference step. This fact can be the input of another forward-inferencing cycle.

Since the inferential closure is infinite in general, the forward search depth, i.e., the number of iterations for using deduced facts as new triggering formulas for further forward inference steps, must be limited by a special parameter of the knowledge base extender.

Sort processor The sort processor solves logical inference tasks which refer to the sort hierarchy defined by the sort declarations of a L_{LILOG} knowledge base. Typically, the inference tasks that can be submitted to the sort processor are those that test whether the subsumption relationship holds between two L_{LILOG} sort expressions, compute the greatest lower bound for two sort expressions, or ask whether a sort expression stands for the empty set.

This kind of inferencing is offered on the one hand to users of the inference engine. On the other hand, we also use the sort processor within the inference engine, e.g., during the unification process, when the compatibility between the sort of a variable and the sort of a term that is to be substituted for the variable must be checked. Processing the sort literals (those literals whose predicate is a sort expression) also requires us to invoke the sort processor to check whether a sort literal can be eliminated [47].

The sort processor can be realized by an inconsistency checker for sort expressions, since all inference tasks concerning sort expressions can be reduced to the question of whether a sort expression is inconsistent, i.e., whether it stands for the empty set. If we have to test whether the subsumption relation $se \ll se'$ holds, this is equivalent to the question of whether **and**(*se*, **not**(*se'*)) stands for the empty set. The computation of the greatest lower bound of two sort expressions can be represented in terms of subsumption tests and an inconsistency test.

General proof procedure

The general proof procedure (GPP) can be regarded as the heart of the inference engine. It is the basic inference algorithm upon which the problem solver and the knowledge base extender offered at the logic interface are realized through special calls to the GPP.

From an abstract point of view, the GPP can be viewed as a theorem prover, because its essential task is to prove goals formulated by L_{LILOG}. Typical theorem provers that have been developed in the recent past, such as the Prolog Technology Theorem Prover [44] or SETHEO [45], focus on the efficient implementation of pure predicate logic. Their efficiency results from choosing one specific inference calculus as the basis of the prover and then implementing it in "lower"-level languages such as C by using special implementation techniques such as abstract machines.

All of these provers have been designed as stand-alone implementations of predicate logic, and efficiency has often been their major design objective. In contrast to that, the inference engine for L_{LILOG} has been designed with a concrete application in mind (to be the processor of the semantic knowledge of a natural-language understanding system). Moreover, the idea of being able to experiment with inference calculi and search strategies has had a major influence on the architecture of our proof procedure as well. Thus, our main objective was to construct an inference engine able to deal with a broad range of different tasks and easily adaptable to new applications. To achieve these objectives, efficiency had to be sacrificed.

Considering the spectrum of possible objectives one may set up for developing a theorem prover, the LILOG inference engine is a bit out of the mainstream when

efficiency is the major concern. Thus, making direct comparisons between rigid high-speed provers for pure predicate logic and a flexible prover for a powerful sorted logic is difficult.

Characteristics of the general proof procedure The general proof procedure is realized as a search procedure for traversing or constructing proof graphs. Compared to standard resolution theorem provers such as the ones mentioned above, it is distinguished from them by several special features:

- *Generalized clauses* [33] are used instead of normal clauses. These generalized clauses are disjunctions of *complex literals*, each in itself consisting of a conjunction of (simple) literals (see the subsection on axioms). This representation avoids certain disadvantages of the classical clausal normal form. In particular, rules with conjunctive conclusions or disjunctive premises need not be split into several clauses. Also, to represent the negation of L_{LILOG} goals, one generalized clause is sufficient, since the negation of the goal formula yields a disjunction of conjunctions, i.e., again a clause in our generalized normal form. Using generalized clauses, the proofs become shorter in general. Furthermore, due to a method for generating lemmas, multiple proofs of identical subgoals can be avoided.
- A straightforward extension of the resolution calculus enables us to process these generalized normal forms: two complex literals $L_1 \wedge \cdots \wedge L_m$ and $L'_1 \wedge \cdots \wedge L'_n$ are complementary iff two of the simple literals contained in the complex ones are complementary.
- The GPP can run under different *inference calculi* as well as different *search strategies*. Therefore the "naked" GPP (i.e., without a calculus and search strategy) can be considered to be a theorem prover shell. For the GPP, an inference calculus is a set of calculus rules. In the current implementation the following rules are realized:
  - The *model-elimination extension* rule [41], adapted to generalized clauses (cf. [33]).
  - The *model-elimination reduction* rule, also adapted to the generalized clauses.
  - The *sort-elimination* rule [48] for proving sort formulas.
  - The *execute-built-in* rule for evaluating built-in predicates, features, and built-in arithmetic operators that occur in equations and inequations.
  - The *tms-lookup* rule for consulting the truth-maintenance system (see the subsection on the TMS).
  - The *depic-inspection* rule for letting a subgoal be proved by the depictional component (see the subsection on external reasoners), the only external reasoning component connected to the GPP at the moment.

  The GPP can run with any subset of these rules; thus, it is easy to configure the inference engine. If, for example, one is not interested in using the TMS or the depictional component, one need only delete the corresponding inference rule from the calculus. Adding a new rule is also not complicated. One need only write the Prolog code that implements the rule, without having to modify existing code.
- A part of the search space can be temporarily disregarded during a proof. This is realized by introducing a new kind of node in the search tree or graph—the *suspended* node; this is explained in more detail below.
- The conditions for terminating a proof can be quite complex. In particular, it is not always enough to deduce the empty clause as in classical theorem-proving applications, because the GPP must support the realization of a variety of other inference tasks.

Finding a proof as a general search problem The task of finding a proof of a logical formula can be formulated in terms of a tree/graph-search problem (cf. [49]); i.e., given a set of nodes with specific initial and terminal nodes, plus a relation that determines the successors for each node, the problem is to find a path from an initial node to a terminal node.

For resolution-based calculi, the nodes in such a tree represent clauses from the initial clause set (also called the input set) as well as clauses created during the proof. The set of initial nodes depends on the resolution refinement[2] used. For the set-of-support strategy, for example, the initial nodes of the proof graph are the clauses of the support set; for linear resolution and model elimination, one must choose one of the input clauses such that the set of the remaining clauses is satisfiable.

The expansion of a node (i.e., the calculation of its successor clauses) consists of applying the calculus rules to this selected node, possibly with additional arguments. For the resolution rule, expanding the selected node means performing all of the resolution steps which are valid according to the used resolution refinement.

In the set-of-support strategy, for example, only clauses in the current search tree and the initial clause are candidates for performing a resolution step with the selected clause. This restriction can be imposed because one assumes that the clauses stemming from the support set are not contradictory. Thus, a contradiction can be found only by resolving either two clauses from the proof tree, or a clause from the proof tree with one of the clauses of the support set. If linear resolution or model elimination are used, the relevant clauses for performing

[2] We assume here that different refinements of the resolution rule constitute different resolution calculi. For the most common resolution refinements, see [40, 41].

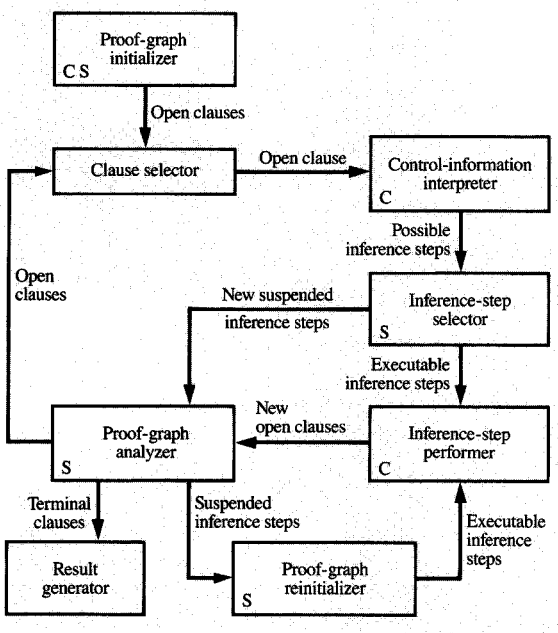

Figure 3

General proof procedure.

resolution steps with the selected node are those of the input set, but in the case of linear resolution, the predecessors of the clause to be expanded must be considered as well.

A terminal node in a normal theorem-proving application corresponds to the empty clause. But since we use the inference engine for a variety of user-specified inference tasks, the conditions for a terminal node can be more complex, and in only a few situations does the empty clause happen to be a terminal clause. For example, when the GPP is used to execute forward inferences, any clause containing only literals originating from the conclusion of an applied forward inference rule qualifies as a terminal clause. This is because, in forward-chaining mode, we are resolving away the premises of some L_{LILOG} axioms and take the instances of the conclusions as inferred formulas.

The search space is traversed according to a certain strategy. In the context of tree or graph search there are two kinds of nodes: open and closed. Open nodes have not yet been expanded, whereas for closed nodes the successors have already been determined. The role of the strategy is therefore to select the next open node to be expanded; in the depth-first search, this is the most recently created open node. (For an apparently more intelligent strategy, this could be the node with the shortest clause.)

The distinction between closed and open nodes is the simplest one. For our proof procedure we have introduced a third class of nodes, called *suspended nodes*, in order to realize more complex search strategies. Suspended nodes are nodes which are temporarily disregarded by the search procedure. In situations where no terminal node can be reached, suspended nodes are "reactivated" for the search. Various search strategies can be modeled using the concept of suspended nodes—for example, iterative deepening, where a node is suspended, if its depth is equal to the current maximal search depth, or the combination of an efficient, incomplete search strategy with a less efficient but complete one, where the suspended nodes play the role of a reserve to be used if the efficient search strategy fails. This technique also allows us to focus the search on particular solutions, since nodes that lead to incompatible solutions are simply suspended. We explain this in more detail in the section on the proof graph analyzer.

After this general overview we describe the proof procedure in more detail by walking through one inference cycle of the inference engine. **Figure 3** shows the main parts of the general proof procedure. The indices in the boxes indicate whether the corresponding subprocedures are parameterized by the inference calculi (C) or by the search strategies (S).

Proof graph initializer Because of our model elimination calculus for generalized clauses, the initialization of the proof graph becomes trivial. We suppose that background and text knowledge bases, from which the goal is derived, are consistent [40].

According to the model elimination calculus, the goal is a logical consequence of these knowledge bases, if the proof search leads to a terminal clause when starting from one of the clauses from the negated goal. The usual choice of the starting clause can be avoided for our generalized clauses, since the negated goal can always be represented by exactly one generalized clause. This is only one advantage of our generalized normal form, as the example below illustrates. Using the classical clausal normal form, the negation of the goal $P \vee Q$ yields $\neg P \wedge \neg Q$, which must be represented by the two clauses $\neg P$ and $\neg Q$ while it is a single clause (consisting of one complex literal) in our normal form.

Additionally, a temporary proof knowledge base for the clauses of the proof graph is created in which the initial goal clause is stored. Including this clause in the input clause set is necessary for obtaining indefinite solutions; see also [41].

Clause selector When a new inference cycle is entered, the first activity is to select a clause to be expanded next. The set of open clauses in the proof graph is always ordered according to the criteria imposed by the search strategy. Therefore, the clause selector has only the trivial task of taking the first open clause as the clause to be expanded.

Control information interpreter Given the selected clause, the control information interpreter determines the possible inference steps that can be applied to this selected open clause. In some sense, the actual calculus rules themselves make this choice, because a calculus rule not only performs the actions for its execution, but also determines its potential inference steps. For a resolution-style calculus rule such as the model elimination extension rule, this means that a calculus rule selects only those clauses to which it can be applied to perform a possible resolution step.

This allows us to conceive different calculus rules that perform the same actions but interpret the control knowledge in different ways; e.g., one rule considers only L_{LILOG} rules whose conclusion is marked by an entry point, whereas another calculus rule may look at all clauses in the knowledge bases.

For resolution-like inference steps, the unifier can be established when computing the possible inference steps. This means that we select for the possible inference steps only clauses which pass the "full unification filter." Because of the complex sort information which must be considered during the unification process as well, computing the unifiers as part of the clause selection may be too costly. As an alternative, one may specify special calculus rules for which the cheaper unsorted unification is done by the control information interpreter and the more expensive sort checks are executed by the inference step performer.

This shows that we interpret the term "calculus rule" in a very broad sense, and also that a part of the inference strategy is transferred into the calculus rules. However, since one of our main goals was to be able to have an experimental environment for testing different calculi with different strategies, this general concept of calculus rules is an important means of achieving the amount of flexibility we require.

Calculus rules may require specific control operators in order to be applied. The switch to the external reasoners is realized by such an operator. If a literal of the goal is to be proven by such a component, it must be marked by the corresponding control information. The calculus rule for invoking an external reasoner checks whether the respective control information is attached to the goal literal, and only if this is the case does it generate the corresponding inference step specification consisting of the calculus rule with its arguments.

The same holds for the rules-processing literals that involve specific predicates. The execute-built-in rule, for instance, checks whether the literal to be proven is an equation or has a built-in predicate. Analogously, the sort-elimination rule tests whether the literal to be processed is a sort literal.

Inference-step selector The output of the control information interpreter is a set of inference-step specifications containing the necessary information for the execution of an inference step. This set of possible resolution steps is divided according to a strategy by the inference-step selector into two disjoint subsets: the set of executable inference steps and the set of the suspended inference steps.

Inference-step performer The inference step performer executes the selected inference steps. For resolution steps, this means in general that the two parent clauses, instantiated by the calculated unifier, are concatenated without the resolved literals. Other inference steps need more processing, e.g., if an external reasoner is consulted.

Proof graph analyzer The result of the inference-step performer is a set of new open clauses. This set must be integrated into the existing proof graph, which consists of the actual set of open clauses, the set of closed clauses, and the set of terminal clauses. More precisely, the new terminal clauses are determined. The remaining new open clauses are integrated into the set of open clauses according to the search strategy; e.g., if we have breadth-first search, the set of open clauses is represented by a list, and the new open clauses are added at the end of that list.

It may happen that the set of open clauses and the set of terminal clauses are not disjoint. This is useful for realizing certain options in an elegant way. According to one option, one may request the forward chainer to deduce the most specific consequences that can be obtained from the trigger formula. For example, assume that the disjunction $D_1 \vee D_2$ qualifies as a terminal clause such that $\neg D_1$ can be proven (or, in other words: the literals in D_1 can be "resolved away"); then, D_2 should be a consequence derived by forward inferences. For specializing this disjunction, it is necessary to try additional inference steps so that $D_1 \vee D_2$ must also remain in the set of open clauses.

This possibility of allowing a clause to be both a terminal and an open clause is also exploited for handling the presupposition order of the subgoals in the representation of a question. According to this option, the subgoals must be processed in a certain order until the proof of one subgoal fails. The results of an inference task

handling the presuppositions contains both the proven and the nonproven subgoals of the original question representation. A simple way to achieve this is to include open clauses whose literals originate from the initial question representation in the set of terminal clauses as well. After the proof search, the literals in the shortest terminal clause represent the minimal set of nonproven subgoals.

The proof graph analyzer is also the part of the GPP responsible for focusing the search on specific solutions. Suppose that for a query

goal *goal-1*
exists $X : S_X$, $Y : S_Y$;
?-Q(X, Y).

the preferred solutions for X are a_1 or a_2, and those for Y are b_1, b_2, or b_3. The proof graph analyzer determines for every new open clause whether the instantiations of the variables X and Y are still compatible with the preferred solutions. If X is replaced by a term t_X, this means that t_X must be unifiable with either a_1 or a_2; i.e., if t_X is a variable, the sort of t_X must be a supersort of the sort of a_1 or a_2. If t_X is a ground term, it must be either a_1 or a_2. Open clauses whose instantiations of X and Y are incompatible with the preferred solutions are suspended; i.e., in our search tree we have two classes of suspended nodes: *suspended inference steps* determined according to a search strategy and *suspend clauses* for realizing the preferred solutions option. If no open clauses are left in the search tree, the proof graph analyzer must drop one of the constraints for the instantiations of the variables. In our example, these may be the constraints on the variable X. Then all suspended clauses are reactivated (i.e., reconsidered as open clauses, which do not violate the remaining constraints), and the proof search continues as usual.

When the new sets of open, suspended, and terminal clauses have been determined, the proof graph analyzer must decide whether the proof is finished. This depends on the option specifying the number of desired solutions. If only one solution is to be computed, the proof terminates when the first (real) terminal clause has been generated. If all solutions are desired, the search continues until the set of open clauses becomes empty.

Result generator The result generator must select the relevant terminal clauses that contribute to the results. As shown by the example of the most special facts deduced by forward inferences, there may be terminal clauses that can be subsumed by other terminal clauses. From the selected terminal clauses, the results are determined according to the "kind-of-result" options. With these options one can specify whether valid instantiations of the goal, valid substitutions, or simply success/failure should be the reported result.

Proof graph reinitializer If the set of open clauses is empty and there are still some suspended nodes not satisfying the termination conditions, we can reactivate such suspended inference steps as possible usable ones and select one of them according to the search strategy for the reinitialization of the search. The selected inference steps are executed, and the generated clauses constitute the new set of open clauses.

Inconsistency checker for sort expressions The basic inference algorithm for processing the knowledge given in terms of the sort declarations of a L_{LILOG} knowledge base is an inconsistency checker for sort expressions. Its task is to decide whether a sort expression stands for the empty set. Because of the richness of our sort description language, other inference tasks concerning the sort information of a knowledge base can be reduced to the question whether a sort expression is inconsistent; see also the subsection on the logic interface.

The basic steps of the inconsistency-checking algorithm are the ones described below (see also [50] for a comprehensive overview of such algorithms):

1. Fully expand the given sort expression according to the sort constraints in the knowledge base. This replaces every occurrence of a sort name with a sort expression corresponding to the constraints for that sort name. Apply this expansion process recursively until no further expansion of user-defined sort names is possible. The recursive nature of the expansion process requires that the sorts of a knowledge base must not be defined cyclically. This is checked by an algorithm for detecting cyclic sorts within a knowledge base.
2. Normalize the expanded sort expression by transforming it according to the following rules:
   - Push the negation symbols inward as far as possible; after this only sort names and enumerations can be negated.
   - Apply the law of associativity by flattening nested intersections and nested unions.
   - Contract sort expressions *f:se* and *f:se'* appearing in an intersection to *f:***and***(se, se')* until there is at most one such expression per feature within an intersection.
   - Contract sort expressions *r::se* and *r::se'* appearing in an intersection to *r::***and***(se, se')* until there is at most one such expression per role within an intersection.
   - Contract the enumerations (and negated enumerations) in intersections until there remains

exactly one enumeration per intersection or the intersection is replaced by the empty set.

3. Test whether the consistency of the normalized sort expression can be ensured; i.e., the sort expression cannot be inconsistent. This is the termination condition for delivering a negative result.
4. Test whether the inconsistency of the normalized sort expression can be ensured. This is the termination condition for delivering a positive result.
5. If the termination conditions are not satisfied, reduce the question of whether the given sort expression is inconsistent to certain subgoals, i.e., to the inconsistency of other sort expressions, according to the inconsistency-checking calculus for L_{LILOG} sort expressions.

Besides computing whether a sort expression stands for the empty set, the sort processor stores the results it has computed by filling a table of greatest lower bounds of sort expressions, in order to avoid repeated computations during the lifetime of a knowledge base.

Truth-maintenance system

The truth-maintenance system (TMS) maintains the assumptions and conclusions generated by the problem solver and puts in order the dependencies among them. Thus, it is able to determine the current set of valid conclusions (*beliefs*) and to name the assumptions supporting those conclusions. If an inconsistency arises, the TMS follows the dependencies back to the assumptions which caused the inconsistency. It can remove these assumptions and all dependent conclusions from the set of current beliefs.

Besides its main purpose of supporting belief revision, a TMS can serve as a *cache* for inferred problem-solver data. Since it regulates the inference process, it can also generate simple explanations. The best-known TMSs are those of [51, 52].

The TMS integrated into the LILOG inference engine is based on the argument-based default logic developed in [26]. Conflicts between contradicting conclusions are resolved by comparing the strength of their arguments, which are the rules and facts used for their deduction. Their default strength is identified with a vector of sort expression, where a vector of sort expression which is subsumed[3] by another one represents the stronger argument. Currently, our TMS is used primarily as a cache. For consulting the TMS during problem solving, we have a calculus rule called *tms-lookup* that determines whether a fact is valid. A detailed presentation of this TMS is in [35].

[3] We use the componentwise subsumption relationship between the sort expressions in two such vectors as the subsumption relation between vectors of sort expressions.

External reasoners

In the current implementation there is one external reasoner attached to the inference engine: the depictional component [39], which reasons over spatial information on the basis of analog representations. The two main processes of this component are imagination and inspection. The *imagination* process generates the analog representation for a certain spatial relationship within the internal knowledge base of the depictional reasoner. It is activated when the knowledge base extender derives a literal which is marked for processing by the depiction module. The literal, instead of being added to the propositional knowledge base, is handed to the depiction module to be memorized there. *Inspection* is the dual process; it determines whether a certain spatial relationship holds. The *depic-inspection* calculus rule transfers a literal that is to be processed outside the theorem prover to the depiction component. If the depiction module is able to prove this subgoal, it returns an answer substitution to the general proof procedure, which can then resolve away this literal.

These inference rules show how a principal solution for attaching an external reasoning device could be attached to a theorem prover without having to leave the logical framework.

Access to knowledge

The knowledge base access system allows us to retrieve the knowledge entities of a L_{LILOG} knowledge base which we need for the inferential processes. In the corresponding queries, syntactic properties can be specified as search patterns. Thus, we may search for the declarations of certain predicates, the roles and features of specific sorts, or axioms of a certain type (such as facts or the backward-chaining rule) containing certain literals (e.g., with a specific predicate and polarity).

This knowledge base access system is realized in terms of an interface module to the LILOG database system [5] that has been developed as an advanced background storage mechanism satisfying special requirements from the knowledge based systems context.

4. Summary

The LILOG inference engine is the basis for the LILOG knowledge representation system. This working environment for knowledge engineers comprises tools such as a graphical browser for sort lattices defined in terms of L_{LILOG} sort declarations. The L_{LILOG} compiler translates the knowledge bases into a processible representation on the basis of which the LILOG inference engine executes the knowledge.

The LILOG knowledge representation system has been realized in Quintus Prolog™ and is running under AIX® on PS/2® and RS/6000™ workstations. This system comprises about 45 000 lines of Prolog code.

In the context of the LILOG project on natural-language understanding, the knowledge representation environment has been used for modeling the background knowledge of the LEU/2 text-understanding system. The inference engine processes this background knowledge together with those knowledge bases containing the L_{LILOG} representations of natural-language texts. The background knowledge base of LEU/2 contains knowledge about touristic activities and attractions in general, plus specific knowledge about the city of Düsseldorf. In addition to that, the background knowledge contains the description of numerous temporal and spatial relationships as well. This constitutes some 600 sort declarations and some 300 logical axioms in that knowledge base.

Although the LILOG knowledge representation system has been developed as part of the LEU/2 text-understanding system, we have employed it in nonlinguistic contexts as well.

The KBSSM (Knowledge-Based System for Security Management) system [53] supports system security managers in their task of defining and controlling the access rights to computer systems. Much of the information concerning security management can be described as logical rules about the accessibility of computer systems. Thus, it appeared natural to us to investigate a knowledge based approach in such a context and to formulate the rules relevant for the access to data or devices in L_{LILOG}. The inference engine could then be used to check whether the set of rules defining the access rights was consistent and whether certain actions violated the security specifications. Thus, the knowledge based approach led to an immediate implementation of the security rules, whereas for conventional systems security rules must be implemented in a procedural language. This has shown the adequateness of L_{LILOG} and its underlying inference mechanism as a language for writing executable specifications of software systems.

The inference services offered at the user interface level of our deductive engine as described above covered nearly all the requirements of the new application context. Only one additional inference task had to be added to check whether a certain property holds for all members of a certain sort [54]. However, this was quite easy because of the modular architecture of the system.

During the project, several extensions have been investigated. Reference [55] presents a graph-based extension of the model elimination calculus that has been realized for allowing a more directed proof search. A learning method has been developed for acquiring proof plans from proofs of similar goals [56]. Furthermore, the prototype of a control language for specifying declaratively the sequence of actions to be undertaken during the search for a proof [57, 58] has been implemented.

We have learned the following lessons from the development of the LILOG knowledge representation system:

- Building a knowledge representation system should start with a clear design of the language to be processed by such a system. A clear design should be rooted in a well-established technical framework in which the syntax and formal semantics of a language can be defined. In our case this was the context of order-sorted predicate logic, KL-One-like type descriptions plus some fields of nonmonotonic reasoning. It turned out that the attachment of external reasoners can be handled easily through corresponding calculus rules (see below).
- The development of the inference engine for the L_{LILOG} language has benefitted considerably from the clearly defined notion of logical inference. This has been a good guideline for clarifying what must be implemented in order to put the language into operation: an inference engine that offers enough inference rules for processing the various syntactic kinds of logical axioms that may occur in a knowledge base. Thus, the inference calculus determines the deductive power of the engine, and the formal semantics defined for L_{LILOG} is the yardstick for measuring the potential of the calculus in terms of correctness and completeness.
- Handling the inference calculus (as well as the search strategy) as a separate module of the inference engine has turned out to be a good means for adjusting the computational power and efficiency of our implementation of L_{LILOG} with little effort. For example, the easy attachment of external reasoners to our inference engine has become possible because we can interact with such a device simply by adding a new inference rule to the calculus under which the inference engine runs. However, the flexibility we have gained through this approach has its price in terms of overall performance.
- L_{LILOG} allows the "permissive" use of sorts, features, roles, and atoms in both sort expressions and logical axioms. This approach was motivated by the experimental nature of the whole system. By allowing the same information to be expressed in different ways in L_{LILOG} (e.g., feature values being specified in sort expressions or by equations in axioms), we achieved a high flexibility for the modeling of knowledge. However, this had the drawback that the knowledge engineers were uncertain how to express certain facts or relationships. L_{LILOG} imposes no modeling style. Such a style was influenced to a large degree by the inferential capacities of the inference engine. Language constructs that could

be processed efficiently were preferred to those where this has not been the case.

- Finally, the implemented calculus for processing L_{LILOG} does not cover all possible valid inferences that can be drawn semantically; i.e., even though the calculus is sound, it is not complete. The completeness gaps result mainly from the dual nature of sorts, roles, and features, the use of equality in axioms, and the use of disjunctions in sort expressions. It would be possible to close these gaps by employing calculi such as paramodulation for handling equality or those described in [47, 59] to account for sorts being used as predicates. Their implementation, however, would lead in general to a prohibitive explosion of the search space, since they enable inference steps that are not possible in the order-sorted case. We realized, therefore, mainly only those inferential capacities that were imposed by the requirements of the applications.

In summary, the experiences we gained in the LILOG project led to a deeper theoretical understanding of logic-based knowledge representation, and should enable us to develop a knowledge representation system

- That constitutes a reasonable comprise between the expressiveness of the representation language and the complexity and the efficiency of the calculus processing it.
- Whose inference engine is tuned versus efficiency by, e.g., choosing an abstract machine architecture for the inference engine.
- That offers powerful tools for developing and inspecting knowledge bases.

Such a system could form an advanced programming environment for knowledge based systems.

Acknowledgment

Many people have contributed to the development of the LILOG inference engine and the knowledge representation system as a whole. We would like to thank Stephan Bayerl, Karl Hans Bläsius, Uli Hedtstück, Kai von Luck, and Karl Schlechta for their conceptual support. Roland Seiffert implemented the L_{LILOG} compiler, Thomas Ludwig was responsible for the knowledge base access, and Jan Wilms provided the knowledge engineering tools. And finally, Josef Gemander, Gerd Kortüm, Thomas Link, Martin Müller, and Karin Neuhold helped us to implement the inference engine.

References

1. *Rule-Based Expert Systems*, B. G. Buchanan and E. H. Shortliffe, Eds., Addison-Wesley Publishing Co., Reading, MA, 1985.
2. *Textunderstanding in LILOG*, O. Herzog and C.-R. Rollinger, Eds., *Lecture Notes in Artificial Intelligence*, Volume 546, Springer-Verlag, New York, 1991.
3. R. J. Brachman, V. P. Gilbert, and H. J. Levesque, "An Essential Hybrid Reasoning System: Knowledge and Symbol Level Accounts of KRYPTON," *Proceedings of IJCAI-85*, 1985, pp. 532–539.
4. A. M. Frisch, "The Substitutional Framework for Sorted Deduction: Fundamental Results on Hybrid Reasoning," *Artif. Intell.* **49,** 161–198 (1991).
5. Thomas Ludwig, "A Brief Overview of LILOG-DB," *Proceedings of the 1990 Conference on Data Engineering*, M. Liu, Ed., Los Angeles, 1990.
6. *Readings in Knowledge Representation*, Ronald J. Brachman and Hector J. Levesque, Eds., Morgan Kaufmann, Los Altos, CA, 1985.
7. K. Jensen and N. Wirth, *PASCAL: User Manual and Report*, Springer-Verlag, New York, 1975.
8. N. Wirth, *Programming in Modula 2*, Springer-Verlag, New York, 1985.
9. M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF*, *Lecture Notes in Computer Science*, Volume 78, Springer-Verlag, New York, 1978.
10. Joseph A. Goguen, James W. Thatcher, and Eric G. Wagner, "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types," *Current Trends in Programming Methodology IV: Data and Structuring*, R. Yeh, Ed., Prentice-Hall, Inc., Englewood Cliffs, NJ., 1978, pp. 80–144.
11. *The Vienna Development Method: The Meta-Language*, D. Björner and C. B. Jones, Eds., *Lecture Notes in Computer Science*, Volume 61, Springer-Verlag, New York, 1978.
12. F. Pereira and S. M. Shieber, "The Semantics of Grammar Formalisms Seen as Computer Languages," *Proceedings of the 10th International Conference on Computational Linguistics*, Stanford, 1984, pp. 123–129.
13. R. T. Kasper and W. C. Rounds, "A Logical Semantics for Feature Structures," *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, Columbia University, New York, 1986, pp. 257–265.
14. R. J. Brachman and J. G. Schmolze, "An Overview of the KL-ONE Knowledge Representation System," *Cogn. Sci.* **9,** 171–216 (1985).
15. W. A. Woods and J. G. Schmolze, "The KL-ONE Family," *Report No. TR-20-90*, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1990.
16. Gosse Bouma, Esther König, and Hans Uszkoreit, "A Flexible Graph-Unification Formalism and Its Application to Natural-Language Processing," *IBM J. Res. Develop.* **32,** 170–184 (1988).
17. J. Dörre and R. Seiffert, "A Formalism for Natural Language—STUF," *Textunderstanding in LILOG*, O. Herzog and C.-R. Rollinger, Eds., *Lecture Notes in Artificial Intelligence*, Volume 546, Springer-Verlag, New York, 1991.
18. A. Oberschelp, "Untersuchungen zur mehrsortigen Quantorenlogik," *Mathematische Ann.* **145,** 297–333 (1962).
19. C. Walther, "A Many-Sorted Calculus Based on Resolution and Paramodulation," *Research Notes in Artificial Intelligence*, Morgan Kaufmann, Los Altos, CA, 1987.
20. C. Beierle, J. Dörre, U. Pletat, C.-R. Rollinger, and R. Studer, "The Knowledge Representation Language L-LILOG," *CSL'88—2nd Workshop on Computer Science*

Logic, E. Börger, H. Kleine Büning, and M. M. Richter, Eds., *Lecture Notes in Computer Science*, Volume 385, Springer-Verlag, New York, 1989, pp. 14–51.
21. G. Smolka, "A Feature Logic with Subsorts," *Lilog-Report 33*, IBM Deutschland GmbH, Stuttgart, 1988.
22. U. Pletat and K. v. Luck, "Knowledge Representation in LILOG," *Sorts and Types for Artificial Intelligence*, K. H. Bläsius, U. Hedtstück, and C.-R. Rollinger, Eds., *Lecture Notes in Artificial Intelligence*, Volume 449, Springer-Verlag, New York, 1990.
23. *Readings in Nonmonotonic Reasoning*, Matthew L. Ginsberg, Ed., Morgan Kaufmann, Los Altos, CA, 1987.
24. M. N. Khenkhar, "Object-Oriented Representation of Depictions on the Basis of Cell Matrices," *Text Understanding in LILOG: Integrating Computational Linguistics and Artificial Intelligence, Lecture Notes in Artificial Intelligence*, Volume 546, Springer-Verlag, New York, 1991.
25. I. Wachsmuth, "Zur intelligenten Organisation von Wissensbeständen in künstlichen Systemen," *IWBS-Report 91*, IBM Deutschland, Scientific Center, 1989.
26. K. Schlechta, "Defeasible Inheritance: Coherence Properties and Semantics," *SNS-Bericht 89–47*, Seminar für natürlich-sprachliche Systeme, Michael Morreau, Ed., Universität Tübingen, Germany, 1989.
27. U. Pletat, "Modularizing Knowledge in L_{LILOG}," *IWBS-Report 173*, IBM Deutschland, Scientific Center, 1991.
28. G. Klose and K. von Luck, "The Background Knowledge of the LILOG System," *Textunderstanding in LILOG*, O. Herzog and C.-R. Rollinger, Eds., *Lecture Notes in Artificial Intelligence*, Volume 546, Springer-Verlag, New York, 1991.
29. A. G. Cohn, "A More Expressive Formulation of Many-Sorted Logic," *J. Automated Reasoning* **3,** 113–200 (1987).
30. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1—Equations and Initial Semantics, EATCS Monographs on Theoretical Computer Science*, Volume 6, Springer-Verlag, New York, 1985.
31. *Deduktions-Systeme*, K. H. Bläsius and H.-J. Bürckert, Eds., Oldenbourg, München, 1987.
32. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
33. T. Bollinger, "A Model Elimination Calculus for Generalized Clauses," *Proceedings of IJCAI-91*, Sydney, Australia, Morgan Kaufmann, Los Altos, CA, 1991, pp. 126–131.
34. T. Bollinger, S. Lorenz, and U. Pletat, "The LILOG Inference Engine," *Text Understanding in LILOG: Integrating Computational Linguistics and Artificial Intelligence, Lecture Notes in Artificial Intelligence*, Volume 546, Springer-Verlag, New York, 1991.
35. S. Lorenz, "Nichtmonotones Schließen mit ordnungssortierten Defaults," *IWBS-Report 100*, IBM Deutschland, Scientific Center, January 1990.
36. Raymond Reiter, "A Logic for Default Reasoning," *Artif. Intell.* **13,** 81–132 (1980).
37. John McCarthy, "Circumscription—A Form of Non-Monotonic Reasoning," *Artif. Intell.* **13,** 27–39 (1980).
38. J. F. Horty, R. H. Thomason, and D. S. Touretzky, "A Sceptical Theory of Inheritance in Semantic Networks," *Proceedings of the 6th National Conference of the American Association for Artificial Intelligence*, 1987.
39. M. N. Khenkhar, "DEPIC-2D: Eine Komponente zur depiktionalen Repräsentation und Verarbeitung räumlichen Wissens," *GWAI-89, 13th German Workshop on Artificial Intelligence*, D. Metzing, Ed., Springer-Verlag, New York, 1989, p. 318–322.
40. C. L. Chang and R. C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Computer Science and Applied Mathematics Series, Academic Press, Inc., New York, 1973.
41. D. Loveland, *Automated Theorem Proving: A Logical Basis*, Volume 6 of *Fundamental Studies in Computer Science*, North-Holland Publishing Co., New York, 1978.
42. W. Bibel, *Automated Theorem Proving*, Vieweg, Berlin, 1982.
43. W. McCune, "OTTER 2.0 User's Guide," *Report No. ANL-90/9*, Argonne National Laboratory, Argonne, IL, 1990.
44. Mark E. Stickel, "A Prolog Technology Theorem Prover," *New Generation Computing* **2,** 371–383 (1984).
45. R. Letz, S. Bayerl, J. Schuhmann, and W. Bibel, "SETHEO—A High-Performance Theorem Prover," *J. Automated Reasoning* **8,** 186–212 (1992).
46. H.-J. Novak, "Integrating a Generation Component in a Natural Language Understanding System," *Textunderstanding in LILOG*, O. Herzog and C.-R. Rollinger, Eds., *Lecture Notes in Artificial Intelligence*, Volume 546, Springer-Verlag, New York, 1991.
47. C. Beierle, U. Hedtstück, U. Pletat, J. Siekmann, and P. H. Schmitt, "An Order-Sorted Predicate Logic for Knowledge Representation Systems," *Artif. Intell.* **55,** 149–191 (1992).
48. U. Hedtstück and P. H. Schmitt, "A Calculus for Order-Sorted Predicate Logic with Sort Literals," *Sorts and Types for Artificial Intelligence*, K. H. Bläsius, U. Hedtstück, and C.-R. Rollinger, Eds., *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1990.
49. Nils J. Nilsson, *Principles of Artificial Intelligence*, Tioga Press, Palo Alto, CA, 1980.
50. B. Hollunder, W. Nutt, and M. Schmidt-Schauss, "Subsumption Algorithms for Concept Description Languages," *Proceedings of ECAI-90*, 1990.
51. Jon Doyle, "A Truth Maintenance System," *Artif. Intell.* **12** (1979).
52. Johan DeKleer,"An Assumption-Based Truth Maintenance System," *Artif. Intell.* **28,** 127–162 (1986).
53. C. Lingenfelder and Astrid Schmücker-Schend, "Using Knowledge-Based Methods to Administrate an Access Control System," *IWBS-Report 222*, IBM Deutschland, Scientific Center, June 1992.
54. S. Decker and C. Lingenfelder, "Universally Quantified Queries in Languages with Order-Sorted Logics," *Proceedings of GWAI-92, Lecture Notes in Artificial Intelligence*, Springer-Verlag, New York, in press.
55. S. Bayerel and K.-H. Bläsius, "Graph Based Extension of the LILOG Inference Engine," *IWBS-Report 229*, IBM Deutschland, Scientific Center, July 1992.
56. K. Neuhold, "Ein Lernverfahren für den automatischen Erwerb von Kontrollwissen," *IWBS-Report 207*, IBM Deutschland, Scientific Center, January 1992.
57. K. Klabunde, "Erweiterungen der Wissensrepräsentationssprache L_{LILOG} um Konstrukte zur Spezifikation von Kontrollinformation," *IWBS-Report 92*, IBM Deutschland, Scientific Center, 1989.
58. M. Müller, "Implementierung und Integration von Verfahren zur wissensbasierten Steuerung der LILOG-Inferenzmaschine," Diploma Thesis, FH Dortmund, 1991.
59. T. Bollinger and U. Pletat, "An Order-Sorted Logic with Sort Literals and Disjointness Constraints," *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, MA, Morgan Kaufmann, Los Altos, CA, 1992, pp. 413–424.

Received October 24, 1990; accepted for publication October 9, 1992

Toni Bollinger *IBM Germany, GSDL Software Architectures and Technologies, Hans-Klemm-Str. 45, D-7030 Boeblingen, Germany (bollinger@vnet.ibm.com).* Dr. Bollinger studied computer science at the Universities of Bonn and Paris-Sud in Orsay, from which he received diploma degrees in 1984 and 1983, respectively. In 1986 he obtained a "Doctorat de 3e cycle" from the University of Paris-Sud with a thesis concerning generalization in machine learning. In 1987 he joined IBM to work on knowledge-processing for the LILOG project.

Udo Pletat *IBM Germany, GSDL Software Architectures and Technologies, Hans-Klemm-Str. 45, D-7030 Boeblingen, Germany (pletat@vnet.ibm.com).* Dr. Pletat studied computer science and mathematics, receiving his master's degree in 1980 from the University of Dortmund. He then worked as a research and teaching assistant at the University of Stuttgart, where he received his Ph.D. in computer science in 1984 on the subject of formal models for the software design process. After being a Visiting Scientist at the Computer Science Department of the Technical University of Denmark, he joined IBM in 1986. He has worked on the PROTOS project, and since 1987 has been project leader of the LILOG project, first for implementation of the first LILOG prototype and then for the knowledge representation and processing subproject. His current research interests include logic programming, logic-based knowledge representation, and formal methods for software design.