The SIMS Manual
Version 1.0*

José-Luis Ambite
Yigal Arens
Naveen Ashish
Chin Y. Chee
Chun-Nan Hsu
Craig A. Knoblock
Wei-Min Shen
Sheila Tejada
Information Sciences Institute
University of Southern California
4676 Admiralty Way,
Marina del Rey, CA 90292, U.S.A.

July 13, 1995

Abstract

SIMS provides intelligent access to heterogeneous, distributed information sources, while insulating human users and application programs from the need to be aware of the location of the sources, their query languages, organization, size, etc.

This manual explains how to bring up a SIMS information server in a new application domain. After providing a short overview of relevant features of the SIMS system, it describes the modeling and programming work that has to be performed to support the extension of SIMS to a given collection of information sources in the domain. To aid a user inexperienced with the technological infrastructure underlying SIMS, the manual contains examples structured as a tutorial that can be followed to actually produce a working SIMS system.

For general discussion, information and announcements concerning SIMS, please send mail to *sims-forum-request@isi.edu* and ask to be added to the **SIMS-Forum** mailing list. For bug reports, please send mail to *sims-bug-report@isi.edu*.

*The research reported here was supported in part by Rome Laboratory of the Air Force Systems Command and the Advanced Research Projects Agency under Contract Number F30602-94-C-0210, in part by a Technology Reinvestment Program award funded by the Advanced Research Projects Agency under Contract Number MDA972-94-2-0010 and by the State of California under Contract Number C94-0031, in part by the National Science Foundation under Grant Number IRI-9313993, and in part by an Augmentation Award for Science and Engineering Research Training funded by the Advanced Research Projects Agency under Contract Number F49620-93-1-0594. The views and conclusions contained in this paper are those of the author and should not be interpreted as representing the official opinion or policy of RL, ARPA, CA, NSF, the U.S. Government, or any person or agency connected with them.

Contents

1 Introduction

The overall goal of the SIMS project is to provide intelligent access to heterogeneous, distributed information sources (databases, knowledge bases, flat files, programs, etc.), while insulating human users and application programs from the need to be aware of the location of the sources, their query languages, organization, size, etc.

The standard approach to this problem has been to construct a global schema that relates all the information in the different sources and to have the user pose queries against this global schema or various views of it. The problem with this approach is that integrating the schemas is typically very difficult, and any changes to existing data sources or the addition of new ones requires a substantial, if not complete, repetition of the schema integration process. In addition, this standard approach is not suitable for including information sources that are not databases.

SIMS provides an alternative approach. A model of the application domain is created, using a knowledge representation system to establish a fixed vocabulary describing objects in the domain, their attributes and relationships among them. For each information source a model is constructed that indicates the data-model used, query language, network location, size estimates, etc., and describes the contents of its fields in relation to the domain model. SIMS' models of different information sources are independent, greatly easing the process of extending the system.

Queries to SIMS are written in the high-level uniform language of the domain model, a language independent of the specifics of the information sources. Queries need not contain information describing which sources are relevant to finding their answers or where they are located. Queries do not need to state how information obtained from different sources should be joined or otherwise combined or manipulated.

SIMS uses a planning system to determine how to retrieve and integrate the data necessary to answer a query. The planner first selects information sources to be used in answering a query. It then orders sub-queries to the appropriate information sources, selects the location for processing intermediate data, determines which sub-queries can be executed in parallel, and then initiates execution.

Changes to information sources are handled by changing models only. The changes will be considered by the planner in producing future plans that utilize information from the modified sources. This greatly facilitates extensibility.

The rest of this section presents an overview of SIMS and its architecture. In Section 2 we show the format of the queries that a user would input to SIMS and the output that should be expected. Then, we consider in more detail the specification of domain models, in Section 3, and information sources models, in Section 4. Section 5 gives a brief introduction on how to construct a wrapper for a new information source and how to communicate with the wrapper. Section 6 explains how to run SIMS both through its graphical user interface and its functional interface. Section 7 describes how to test and debug a new SIMS application. Section 8 presents the installation and system requirements. Finally, in Section 9 we show the code that would implement the example that is discussed throughout the manual. Section 10 contains a reading list of relevant papers.

1.1 Architecture and Background

A visual representation of the components of SIMS is provided in Figure 1.

SIMS addresses the problems that arise when one tries to provide a user familiar only with the general domain with access to a system composed of numerous separate data- and knowledge-bases.

Specifically, SIMS does the following:

- **Modeling:** It provides a uniform way to describe information sources to the system, so that data in them is accessible.
- **Information Source Selection:** Given a query, it
  - Determines which information sources contain the data relevant to answering the query.
  - For those concepts mentioned in the query which appear to have no matching information source, it determines if any knowledge encoded in the domain model (such as relationship to other con-

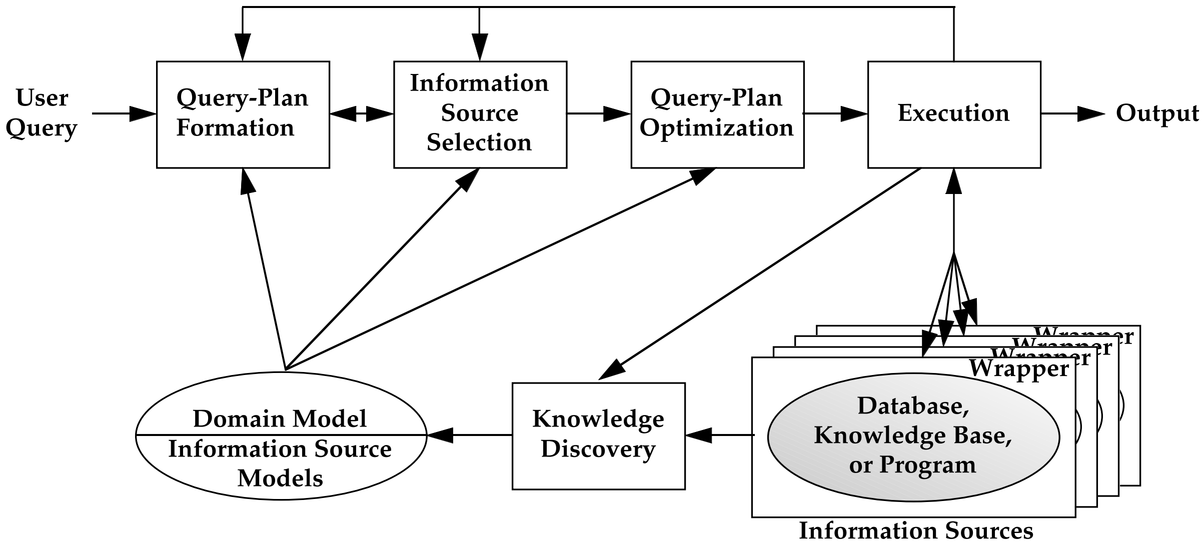

Figure 1: SIMS Overview Diagram.

cepts) permits reformulation of the query in a way that will enable suitable information sources to be identified.

- **Access Planning:** It creates a plan, a sequence of subqueries and other forms of data-manipulation that when executed will yield the desired information.
- **Query-Plan Optimization:** It uses knowledge about databases — their sizes, semantic rules concerning their contents, indexes — to optimize the plan.
- **Discovering Database Semantic Rules:** By querying databases and other information sources and analyzing the returned information, it discovers semantic rules characterizing their contents. This learned knowledge is used to modify SIMS' domain and information source models and ultimately to improve query plans. This module has not been released yet.
- **Execution:** It executes the reformulated query plan; establishing network connections with the appropriate information sources, transmitting queries to them and obtaining the results for further processing. During the execution process SIMS may detect that certain information sources are not available, or respond erroneously. In such cases, the relevant portion of the query plan will be replanned.

Each information source is accessed through a *wrapper*, a module that can translate queries to that information source from the SIMS query language to the query language understood by that source. The wrapper then takes the data returned by the information source and sends it on to SIMS in the form SIMS expects.

1.2 Information Sources Supported

In order for SIMS to support an information source there must be an information source model for it, and there must exist a wrapper for that type of source. While each information source needs to be modeled individually, only one wrapper is required for any type of information source. For example, LIM (see below) serves as the wrapper for any Oracle database.

Wrappers for Loom knowledge bases, Oracle relational databases, and MUMPS-based network databases have already been written for SIMS. To add a new database of any of these types requires, therefore, only to create an information source model for it. In order to add an information source of a new type one would

```
(sims-retrieve (?last-name)
  (:and (patient ?patient)
        (doctor-name ?patient "GONZALEZ")
        (last-name ?patient ?last-name)))
```

Figure 2: Example SIMS/Loom Query

have to obtain, or write, a new wrapper for it. In the case of another database using SQL as its query language, this would be only a simple modification of LIM.

1.3 Technological Infrastructure

This subsection is provided for readers who may not be familiar with the systems underlying SIMS. A general understanding of Loom, LIM and KQML is assumed in the rest of this manual. A list of relevant papers is provided in Section 10.

1.3.1 Loom

Loom serves as the knowledge representation system SIMS uses to describe the domain model and the contents of the information sources. In addition, Loom is used to define a knowledge base that itself serves as an information source. Loom provides both a language and an environment for constructing intelligent applications. It combines features of both frame-based and semantic network languages, and provides some reasoning facilities. As a knowledge representation language it is a descendant of the KL-ONE [1] system.

The heart of Loom is a powerful knowledge representation system, which is used to provide deductive support for the declarative portion of the Loom language. Declarative knowledge in Loom consists of definitions, rules, facts, and default rules. A deductive engine called a *classifier* utilizes forward-chaining, semantic unification and object-oriented truth maintenance technologies in order to compile the declarative knowledge into a network designed to efficiently support on-line deductive query processing. For a more detailed description of Loom, see [3, 4].

To illustrate both Loom and the form of SIMS' queries, consider Figure 2, which contains a simple semantic query to SIMS. This query requests the last names of the patients of a doctor by the name of Gonzalez. The three subclauses of the query specify, respectively, that the variable `?patient` describes a member of the model concept `patient`, that the relation `doctor-name` holds between the values of `?patient` and the string `GONZALEZ`, and that the relation `last-name` holds between the values of `?patient` and the respective values of the variable `?last-name`.

As we will see later in Figure 4, `patient` is a subconcept of `person` therefore, by inheritance, it too has `person`'s attributes, among them `last-name`.

The query specifies that the value of the variable `?last-name` be returned. A query to SIMS need not necessarily correspond to a single database query, since there may not exist one database that contains all the information requested.

1.3.2 LIM

The Loom Interface Module (LIM) [5] has been developed by researchers at Unisys to mediate between Loom and databases. It currently acts as a SIMS wrapper to relational Oracle databases, using the SQL language. LIM reads an external database's schema and uses it to build a Loom representation of the database. The Loom user can then treat concepts whose instances are stored in a database as though they contained "real" Loom instances. Given a Loom query for information about instances of that concept, LIM automatically generates an SQL query to the database that contains the information, and returns the results as though they were Loom instances. LIM focuses primarily on the issues involved in mapping a SIMS query to a single database. After SIMS has planned a query and formed subqueries, each grounded in a single database, it hands the subqueries to LIM for the actual data retrieval. SIMS handles direct queries to the Loom knowledge base on its own.

1.3.3 KQML

To simplify and modularize its interaction with databases, SIMS is designed as an intelligent agent that communicates with other information agents, which can be simple information sources or other SIMS agents. The former are regular databases which are, however, enclosed in software *wrappers* — systems that accept queries to the database in the Loom language and translate them into queries that the local DBMS can handle.

The Knowledge Query Manipulation Language (KQML) [2] is an agent communcation language that supports such messages. We have adopted KQML for our agent-to-agent communication. KQML has been designed to support knowledge-based communication. Using it, agents can transmit structured objects along with the contents of the objects, so they can transmit model fragments together with queries using terms from those models. The KQML language handles the interface protocols for transmitting queries, returning the appropriate information, and building the necessary structures.

2 The SIMS Language

The SIMS language supports three different types of commands for retrieving, modifying, and monitoring information. The command `sims-retrieve` is used for querying, while the commands `sims-insert`, `sims-delete`, and `sims-update` are transaction commands. The commands `sims-begin-notify` and `sims-end-notify` handle the monitoring of data items. In the following sections each type of command will be discussed in further detail.

2.1 Sims-Retrieve Query Command

SIMS takes a `sims-retrieve` query as input and outputs the data satisfying the constraints specified in the query. Currently, SIMS supports most features of the Loom query language. These features should be enough for most database applications. From now on, we call this subset of the Loom query language as the SIMS query language. The output format of SIMS is a (LISP) list which can contain constants, Loom objects or tuples.

```
<query> ::= (sims-retrieve ({<variable>}+) <query-expr>)
<query-expr> ::=
     ({:and | :or} {<query-expr>}+) |
     (:not <query-expr>) |
     (:collect ({<variable>}+) <query-expr>) |
     (:implies <query-expr> <query-expr>) |
     ({:for-some | :for-all} ({<variable>}+) <query-expr>) | <clause>
<clause> ::=
     <concept-exp> | <relation-exp> | <assignment-exp> | <comparison-exp> |
     <aggregation-exp>
<concept-exp> ::= (<concept-name> <variable>)
<relation-exp> ::=
    (<relation-name> {<bound-variable> | <function-exp>} {<term> | <function-exp>})
<assignment-exp> ::= (= <unbound-variable> {<arith-exp> | <set-exp>})
<comparison-exp> ::= <member-comparison> | <arithematic-comparison>
<function-exp> ::= (<relation-name> {<bound-variable> |<constant> |<symbol> |<function-exp>}) |
     (<aggregation-function> <set-exp>)
<set-exp> ::= ({<constant>}+) |(:collect ({<variable>}+)<query-expr>) |<function-exp> |
     <bound-variable>
<aggregation-exp> ::=
     (<aggregation-function> <set-exp> {<clause> |<variable>})
<member-comparison> ::=
     (member <bound-variable> <set-exp>)) | (members <set-exp> <bound-variable>)
<arithematic-comparison> ::=
     (<comparison-op> {<arith-exp> | <function-exp>} {<arith-exp> | <function-exp>})
<arith-exp> ::= <number> | <bound-variable> |
     (<arith-op> {<arith-exp> | <function-exp>} {<arith-exp> | <function-exp>})
<arith-op> ::= + | - | * | /
<comparison-op> ::= = | > | < | >= | <= | !=
<aggregation-function> ::= count | avg | sum | min | max
<concept-name> ::= <symbol>
<relation-name> ::= <symbol>
<term> ::= <constant> | <variable>
<variable> ::= <bound-variable> | <unbound-variable>
<bound-variable>[1] ::= ?<symbol>
<unbound-variable> ::= ?<symbol>
<constant> ::= <number> | <string>
```

Figure 3: BNF for the SIMS Query Language

The BNF syntax for the SIMS query language is shown in Figure 3. We next provide a more in-depth description of the language. The following is the basic form of a SIMS query:

```
(sims-retrieve (?v1 ... ?vn) <query-expr>)
```

The variables listed after the `sims-retrieve` command, $?v_1 \ldots ?v_n$, are considered output variables. This means that the values of these variables are returned as the output of the query. All variables must be named with the prefix '?'. The query expression is composed of clauses and constructors. Clauses determine the values of the variables by binding the variables to specific types of values. In other words, clauses constrain the values of the variables. There are five types of clauses supported by the SIMS language which will be described in the next section. Clauses can be grouped by constructors into queries. Currently, the set of constructors provided is `:and, :or, :not, :for-all, :for-some,` and `:collect`.

A SIMS query returns as output a list of instantiations of the output variables which satisfy the bindings of the clauses in the query body. The following shows an example of output from a SIMS query.

```
(sims-retrieve (?patient ?lname)  (:and (Patient ?patient)
                                        (last-name ?patient ?lname)))
==> ((|i|PATIENTS145443 "Richardson")
     (|i|PATIENTS145437 "Brown")
     (|i|PATIENTS145441 "Kumar") ...)
```

In this query the output variable `?patient` is bound to the concept `Patient` and the variable `?lname` is bound to the values of the role `last-name`, respectively. SIMS returns a set of tuples which contains Loom objects and strings corresponding to the instances of `Patient` and `last-name`. Loom objects are displayed with the prefix `|i|`, which is a Loom internal identification symbol.

2.1.1 Clauses

Clauses are expression that constrain the values which can be bound to a variable. A clause is satisfied when there exists values which satisfy the constraints on the variables in that clause. The following are the five types of clauses:

- Concept expressions:

  ```
  (<concept-name> <variable>)
  ```

  where <concept-name> is the name of a concept, the variable is bound to an instance of the concept <concept-name>. An example of a concept expression is:

  ```
  (Patient ?patient)
  ```

  This constrains the variable `?patient` to only the instances of the concept `Patient`.

- User-defined relation expressions:

  ```
  (<relation-name> <bound-variable> <term>)
  (<relation-name> <bound-variable> <function-exp>)
  (<relation-name> <function-exp> <term>)
  (<relation-name> <function-exp> <function-exp>)
  ```

  where <relation-name> is the name of a relation, <bound-variable> is a variable while <term> can be either a variable or a constant (a number or a string). The first clause states that there is a binary relation <relation-name> between <bound-variable> and <term>. The following are examples of this type of relation expression:

  ```
  (last-name ?patient ?lname)
  (first-name ?patient "Ann")
  ```

  The first expression is only satisfied if the value for `?lname` is the last name of `?patient`. The second expression is only satisfied if "Ann" is the first name of `?patient`.

  The other types of relation clauses contain function expressions, <function-exp>. A function expression is basically the same as a relation expression, except that the second argument of the relation is returned as the result. The expression <function-exp> returns a constant, in this case. A <function-exp> is defined as either of the following:

[1] A variable is considered to be bound if it appears earlier in the query

```
(<relation-name> <term>)
(<relation-name> <symbol>)
(<relation-name> <function-exp>)
(<aggregation-function> <set-exp>)
```

Some examples of function expressions are:

```
(last-name ?patient)
(last-name (used-by-patient ?room))
```

The function `last-name` returns the the last name of the patient. In the second example the function `used-by-patient` returns the patient that is in `?room` and the function `last-name` returns the the last name of that patient.

The following examples are relation expressions which contain these function expressions:

```
(doctor-name ?patient (last-name ?patient))
(last-name (used-by-patient ?room) "Lee")
```

The first expression is satisfied only when the patient and the doctor have the same last name. The second expression is satisfied when the last name of the patient in the `?room` is "Lee".

- Assignment expressions:

```
(= <unbound-variable> <arith-expr>)
(= <unbound-variable> <set-exp>)
```

The first clause assigns to the unbound variable the computed result of <arith-expr>. In the second clause the variable is assigned to a <set-exp> which is defined as a set of constants or any expression which returns a set of constants.

For the following example, suppose we have a concept `patient` and relations on patients and their names (`last-name` and `first-name`). The role `medi-care-fee` states the relation between patients and their medical care charge. The role `insurance-deduct` defines the insurance deduction of a patient. The following query will return a list of patients' names and total balances:

```
(sims-retrieve (?lname ?fname ?total)
          (:and (patient ?patient)
                (last-name ?patient ?lname)
                (first-name ?patient ?fname)
                (medi-care-fee ?patient ?mfee)
                (insurance-deduct ?patient ?insd)
                (= ?total (- ?mfee ?insd))))
   ==> (("Okumura" "Ben" 15000)
        ("DeSpain" "Ann" 20000)
        ("Kumar" "Barbara" 18500)
        ("Hamilton" "Sheila" 27500)
        ("Lee" "Kayano" 26500)
        ⋮
```

- Comparison expressions are used to express a constraint on variables. The following are forms of member comparisons:

```
(member <bound-variable> <set-exp>)
(members <set-exp> <bound-variable>)
```

where a <set-exp> is defined as a set of constants or any expression which returns a set of constants. The first clause is satisfied if the variable is bound to one of the constants (i.e., strings or numbers) in the <set-exp>. In the second clause the order of the arguments are reversed.

The following are examples of member comparisons:

```
(member ?lname ("Smith" "Hamilton" "DeSpain" "Datta"))
(members (doctor-name ?patient) ?doctor)
```

The first expression is only satisfied if the value for `?lname` matches one of the four strings in the set. The second expression is only satisfied if the value for `?doctor` matches one of the strings in the set returned by the function `doctor-name`. The function `doctor-name` returns the names of the doctors for `?patient`.

Another type of comparison expression uses the arithmetic comparison operators: `=`, `>`, `<`, `>=`, `<=`, `!=`. In this case, the expression <function-exp> returns a constant.

```
(<comparison-op> <arith-expr> <arith-expr>)
(<comparison-op> <arith-expr> <function-exp>)
(<comparison-op> <function-exp> <arith-expr>)
(<comparison-op> <function-exp> <function-exp>)
```

The following are examples of the arithmetic comparison:

```
(!= (last-name ?patient) "Kumar")
(= (insurance-deduct ?patient) 300)
(> (medi-care-fee ?patient) (insurance-deduct ?patient))
```

The first example checks that the last name of the patient is not "Kumar". The `=` operator in the second expression tests whether the insurance deduction of the patient is equal to 300. The last example compares the medi-care fee of the patient to the insurance deduction.

- Aggregate expression:

  ```
  (<aggregate-function> <set-exp> <term>)
  ```

  A set of values is required as the first argument for an aggregate function. The value of the operation performed on the set of values is then bounded to <term>, as shown in the following example:

  ```
  (count (doctor-name ?patient) ?count)
  ```

  This counts the set of values returned by the function `doctor-name` and binds the result to the variable `?count`.

2.1.2 Query Expression Constructors

This section provides detailed descriptions for each of the expression constructors supported by SIMS. The examples refer to the models defined in later sections.

`(:and` $expr_1 \ldots expr_n$`)` — CONJUNCTION

This returns the values for which each of the expressions $expr_j$ is satisfied.

Example: `(:and (Office ?x) (hospital-room ?x))`
This expression is satisfied if `?x` is both an `Office` and a `hospital-room`.

`(:or` $expr_1 \ldots expr_n$`)` — DISJUNCTION

This returns the values for which at least one of the expressions $expr_j$ is satisfied.

Example: `(:or (Doctor ?x) (Patient ?x))`
This expression is satisfied if `?x` is either a `Doctor` or a `Patient`.

`(:for-some` $(?v_1 \ldots ?v_n)$ $expr$`)` — EXISTENTIAL QUANTIFICATION

This returns the values for which there exist values for the variables $?v_1$ through $?v_n$ that satisfy the expression $expr$.

Example:

```
(:for-some (?d1 ?d2)
    (:and (patient-of ?patient ?d1) (patient-of ?patient ?d2)
          (!= (doctor-id ?d1)
              (doctor-id ?d2))))
```

This expression is satisfied if there exist a `patient` which has more than one doctor.

`(:for-all` $(?v_1 \ldots ?v_n)$ `(:implies` $expr_1$ $expr_2$`))` — UNIVERSAL QUANTIFICATION

This returns the values of all sets of bindings of the variables $?v_1$ through $?v_n$ that satisfy the expression $expr_1$ and the expression $expr_2$.

Example:
```
(:for-all (?doctor)
    (:implies (patient-of ?patient ?doctor)
        (:not (doctor-Id ?doctor 135))))
```
This expressions is satisfied if all of the doctors of `?patient` do not have the identification number 135.

All of the universally quantified variables $?v_j$ must appear within $expr_1$. This is necessary in order to bind the variables to specific types of values. The bindings for the expression $expr_2$ can then be generated, because the types of the variables have already been determined.

`(:not` *expr*`)` — Negation

This returns the values for which expression *expr* can not be proved satisfiable.

Example:
```
(:and (Patient ?p) (:not (elderly-patient ?p)))
```
This expression is satisfied if `?p` is a `Patient` and `?p` is not known to be an `elderly-patient`.

In the negated expression *expr* variables must be bound when *expr* is evaluated. The following query is not legal because the variable $?p$ is not bound yet.

```
(retrieve ?p (:not (Patient ?p)))
```

`(:collect` $?v$ *expr*`)` — Collect Satisfying Values (Computed Set)

This returns the list of values bound to the variable $?v$ such that *expr* is satisfied.

Example:
```
(:collect ?id
    (:for-some ?patient
        (:and (Patient ?patient)
            (patient-id ?patient ?id))))
```
Returns the list of identification numbers of all the patients.

2.2 SIMS Transaction Commands

SIMS supports transactions of insert, delete, and update under the following assumptions:

- Every transaction must have roles that can be used to uniquely identify one and only one instance of a domain concept.
- Can only add/update/delete one instance per transaction.
- The current release assumes all sub-transactions generated by SIMS can be executed successfully. In the next release, we shall have the complete two-phase commit in place.

2.2.1 Sims-Insert, Sims-Update, and Sims-Delete

- **sims-insert**: for creating a new instance of a concept with given roles/values. This function returns T for success, NIL for a failure.
- **sims-delete**: for deleting an existing instance of a concept. This function returns T for success, NIL for a failure.
- **sims-update**: for changing values of roles for an existing instance of a concept. This function returns T for success, NIL for a failure.

2.2.2 Examples

Assume that we have a domain concept called `patient`, and it has three roles: `patient-id` (key), `sex`, and `religion`. Furthermore, assume that there are two information sources: DB1 and DB2. DB1 contains patient's id and sex, and DB2 contains patient's id and religion. Then the following is a trace of transactions illustrate how to use sims-insert, sims-delete, and sims-update.

Verify there is no patient with ID 111

```
(sims-retrieve (?p)
   (:and (patient ?p)
         (patient-id ?p 111)))  ⟹ NIL
```

Create a new patient with ID 111. Note that this will trigger creations of new instances in both DB1 and DB2.

```
(sims-insert ()
   (:and (patient ?p)
         (sex ?p "M")
         (religion ?p "WHO KNOWS")
         (patient-id ?p 111)))      ⟹  T
```

Query this new patient

```
(sims-retrieve (?p ?s)
   (:and (patient ?p)
         (patient-id ?p 111)
         (sex ?p ?s)
         (religion ?p ?r)))   ⟹  (("M"  "WHO KNOWS"))
```

Change values of this patient

```
(sims-update ()
   (:and (patient ?p)
         (patient-id ?p 111)
         (sex ?p ?sex)
         (= ?sex "F")
         (religion ?p ?religion)
         (= ?religion "EVERY ONE KNOWS"))) ⟹ T
```

Query for the new values

```
(sims-retrieve (?p ?s)
   (:and (patient ?p)
         (patient-id ?p 111)
         (sex ?p ?s)
         (religion ?p ?r)))   ⟹  (("F"  "EVERY ONE KNOWS"))
```

Delete a patient with ID 111

```
(sims-delete ()
   (:and (patient ?p)
         (patient-id ?p 111)))  ⟹ T
```

2.3 SIMS Active Notifications

Clients can ask SIMS to monitor certain data items with specified conditions and actions. SIMS will execute these actions and send a notification via TCP/IP to the client whenever the data items experience changes that satisfy the specified conditions.

2.3.1 Sims-Begin-Notify and Sims-End-Notify

sims-begin-notify

```
(sims-begin-notify
   :concept 'CNAME          ;; The name of the concept to be monitored
   :when    Q1              ;; A SIMS query used as the trigger condition
   :do      #'FUN           ;; Any lisp function that takes two arguments;
```

```
                          ;; the first argument is bound to the concept name,
                          ;; and the second the transaction itself.
   :host    Address       ;; a string for the LISTENER's machine address
   :port    2020)         ;; a port number of the LISTENER

   ===> NOTIFICATION-ID ;; the function returns a notification id.
```

sims-end-notify

```
(sims-end-notify NOTIFICATION-ID)  ;; this function informs SIMS to stop
                                   ;; monitor activities specified in the
                                   ;; notification with the NOTIFICATION-ID.
```

2.3.2 Examples

Assume that there is a machine xxx.isi.edu that has a port 2020 open for incoming messages. Here is how you ask SIMS to start a notification on the concept "finding" where you are interested in knowing any transactions that involve a patient who has an injury at "left front chest":

```
(sims-begin-notify
  :concept 'finding
  :when '(sims-retrieve (?f ?id)
             (:and (finding ?f)
                   (finding-location ?f ?l)
                   (= ?l "left front chest")))
  :do #'show-transaction
  :host "xxx.isi.edu"
  :port 2020)          ⟹  NOTIFICATION-12
```

where `show-transaction` is a function that prints out its second argument. Suppose now, someone else has successfully created a new instance of "finding" as follows:

```
(sims-insert ()
      (:and (finding ?f)
            (finding-id ?f 3)
            (patient-name ?f "James Wilkie")
            (finding-location ?f "left front chest")))
```

Then SIMS will send the following message to the port 2020 on xxx.isi.edu:

```
SIMS NOTIFY: (INSERT ()
             (:AND (FINDING ?F)
                   (FINDING-ID ?F 3)
                   (PATIENT-NAME ?F "James Wilkie")
                   (FINDING-LOCATION ?F "left front chest")))
```

You can stop this notification any time you send SIMS the following message:

```
(sims-end-notify 'NOTIFICATION-12)
```

You will receive a symbol "T" if the cancellation is successful.

3 The Domain Model

A domain model provides the general terminology for a particular application domain. This model is used to unify the various information sources that are available and provide the terminology for accessing those information sources. Throughout this section we use a simple example from a medical domain that involves maintaining patient records and hospital room assignment. The example is very simple in order to provide a complete, but short, description of the model for the example.

The model is described in the Loom language, which is a member of the KL-ONE family of KR systems. In Loom, objects in the world are grouped into "classes" with a set of "roles" defined on each class. See Figure 4 for a small fragment of the domain model. Classes are indicated with circles, roles with thin arrows, and subclass relations with thick arrows. Roles are inherited down to subclasses.

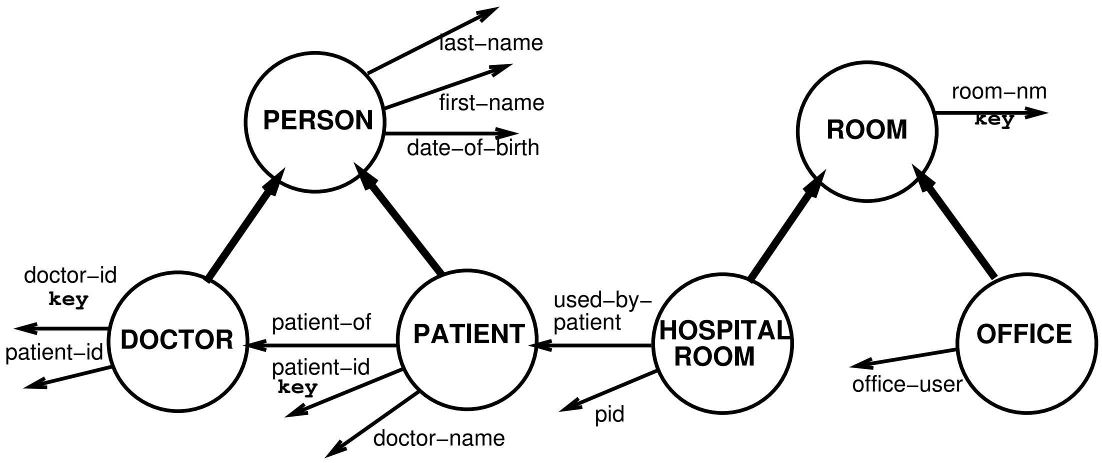

Figure 4: Domain Model Fragment

For example, there is a node in the model representing the class of `person`, a node representing the subclass of `patient`, and a `patient-of` relation specified between `patient` and `doctor`. The definition of the `patient` class is shown in Figure 5.

```
(defconcept Patient
  :is-primitive
  (:and Person
        (:all patient-id String)
        (:all patient-of Doctor)
        (:all doctor-name String))
  :annotations ((key (patient-id))))

(defrelation patient-id
  :domain Patient
  :range String)

(defrelation patient-of
  :domain Patient
  :range Doctor)

(defrelation doctor-name
  :domain Patient
  :range String)
```

Figure 5: Domain-level definition of the `Patient` class and corresponding roles

Other facts about the domain that are represented in the model include which roles on a class (if any) constitute **key roles**. These are roles that uniquely identify instances of the class that forms the domain of the relation. For example, the `patient-id` relation is a key for the class `patient`, as shown in the annotation field of the definition of patient (Figure 5). Key roles are important and powerful, because they help SIMS determine how joins can be performed. The model indicates that the patient-id uniquely identifies a patient, and thus *any* two related patient classes that both have a `patient-id` role can be joined over the patient ID (provided, of course, that they are rendered identically, such as using the same capitalization). See Section 4 for more relevant details.

The entities included in the domain model are not necessarily meant to correspond directly to objects described in any particular information source. The domain model is intended to be a description of the application domain from the point of view of someone who needs to perform real-world tasks in that domain and/or to obtain information about it.

For example, the class of elderly patients, which are patients that are 65 or older, might be particularly important for a given application, yet there may be no information source that contains only this class of patients. Nevertheless, we can define this class in terms of other classes for which information is available. The Loom definition of this class is shown in Figure 6. Notice that the definition of elderly patient requires defining another class for older-than-65, which in turn includes a simple Lisp function for computing the age of a patient.

```
(defconcept Elderly-Patient
  :is (:and Patient
    (:satisfies (?p)
                 (:for-some (?dob)
                            (:and (Patient ?p)
                                  (date-of-birth ?p ?dob)
                                  (older-than-65 ?dob))))))

(defconstant *YEAR* 95)

(defconcept older-than-65 :is
  (:and Number
        (:predicate (dob)
                    (<= 65 (- *YEAR*
                              (- dob
                                 (* 100
                                    (truncate (/ dob 100)))))))))
```

Figure 6: Example of a Defined Class

When viewing model fragments such as that in Figure 4, one must remember that every fact about the domain must either be available in some information source or it must be explicitly represented. For example, consider the relation `used-by-patient`. It is tempting to believe that it stands for a mapping of hospital-rooms to the patients in the those rooms. However, all the figure itself expresses is that it is a mapping between hospital-rooms and patients and that its name is `used-by-patient`. To define the relationship, the model definition includes the following Loom statement shown in Figure 7 (which is not in the original figure because it is difficult to express graphically). It states how the `used-by-patient` relation is related to another one, `patient-id`, which relates `hospital-room` to the patients in that room. This latter relation is determined by its interpretation in some database table, as we will see later. This knowledge will, naturally, be of use in the course of processing this query.

The domain model classes are used as the basis for the **query language** that enables the user to query the information source. It is also the language in which one describes the contents of a new information source to SIMS. This is done by describing how the terms in the information source model relate to the terms in the domain model (see next section for details). In order to submit a query to SIMS, the user composes a Loom statement, using terms and roles in the domain model to describe the precise class of objects that are of interest. If the user happens to be familiar with particular information sources and their representation,

```
(defrelation used-by-patient
  :is (:satisfies (?room ?patient)
                  (:for-some (?pid)
                             (:and (Hospital-Room ?room)
                                   (Patient ?patient)
                                   (pid ?room ?pid)
                                   (patient-id ?patient ?pid)))))
```

Figure 7: Definition of the `used-by-patient` Relation

those classes and roles may be used as well. But such knowledge is not required. SIMS is designed *precisely* to allow users to query it without such specific knowledge of the data's structure and distribution.

The next section describes how information sources are described in SIMS and how their relationship to higher-level domain model classes and roles is specified.

4 Information Source Models

4.1 Modeling the Contents of an Information Source

Each information source is incorporated into SIMS by modeling the information sources and relating those models to the domain model. Appropriate classes in the domain model are linked to representations of the classes of instances contained in the database, or other information source. These mappings include the information SIMS needs to make decisions about when and whether to retrieve data from the information source in order to satisfy a user's query.

To illustrate the principles involved in representing an information source within SIMS, let us consider a table in a relational database. In SIMS, we "translate" the table into the Loom model as follows (see Figure 8).

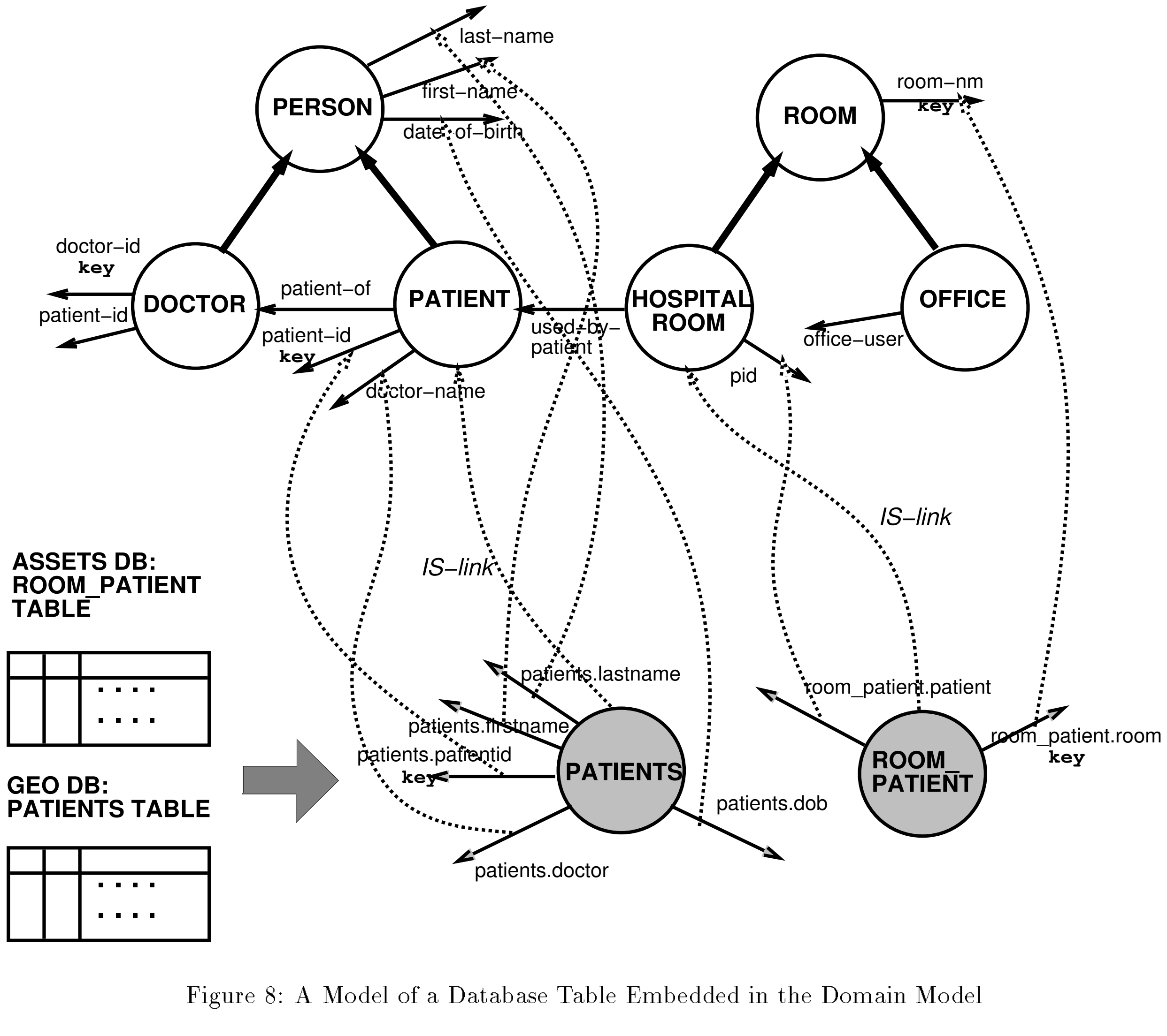

Figure 8: A Model of a Database Table Embedded in the Domain Model

The table is represented by a class that stands for the rows of the table. One may view each instance of that class as corresponding to one of the data items conceptually underlying the table in question. For example, for the `Patients` table in the `GEO` database we will create the Loom class `Patients` whose instances stand for the patients described in that table. The definition of this class is shown in Figure 9. The new class is marked as an *information source class* by specifying the "Info-Source" annotation, which indicates the source containing the corresponding data. In the figure above we have indicated that by filling in the class in question in grey.

```
(defconcept Patients
  :is-primitive
  (:and
       (:the Patients.patientid String)
       (:the Patients.lastname String)
       (:the Patients.firstname String)
       (:the Patients.dob Number)
       (:the Patients.doctor String))
  :mixin-classes (info-source-class)
  :annotations ((Info-Source geo)))
```

Figure 9: Example Source-Level Class Definition

Each column in the table is represented in Loom as a role whose domain is the class standing for the table, and whose range is the class from which the values in the column in question are drawn. For example, the `patientid` column in the `Patients` table of the `GEO` database is represented as the Loom role `Patients.patientid`, as shown in Figure 10.

```
(defrelation Patients.patientid
  :domain Patients
  :range Number)
```

Figure 10: Example Source-Level Role Definition

Finally, each new source class must be correctly related to a class in the domain model. This is done by defining an `IS-link` between the new class and one (or more) in the domain model. An `IS-link` is the way of making explicit the semantics of the information in a given information source. In general, the name of a class is not sufficient to define the semantics of that class. A class may contain names, but the significance of those names is not self-evident. Are they indeed the names of the individual patients? Or, are they the names of the closest relative of the patient? Are they the names of the doctor? The possibilities are endless, and the schema alone is not sufficient to choose one. In order to choose to use this data at the right time, SIMS must know the precise relationship — and an IS-link to a previously defined domain model-level class establishes it. Figure 11 shows the IS-links for the Patients class. These definitions specify that Patients.patientid maps to the patient-id of the patient class, Patients.lastname maps to the last-name of the patient class, and so on.

```
(def-IS-links Patients Patient
    ((Patients.patientid patient-id)
     (Patients.lastname last-name)
     (Patients.firstname first-name)
     (Patients.dob date-of-birth)
     (Patients.doctor doctor-name)))
```

Figure 11: Example IS-links for the Patients class

To summarize, below is the complete list of tasks that need to be performed in the process of creating an information source model describing a database table:

- Create an "information source class" representing the table.
- Create "information source roles" for each column in the table.
- Create the IS-links between the new classes and roles and the domain model.

4.2 Accessing an Information Source

In addition to specifying the contents of an information source, the system also needs to know what information sources are currently available and how to access them. This section first describes the basic commands for declaring information sources and then describes the protocol for communicating with them.

To make an information source available to the system, the name, host, and access function must be declared in advance. This provides the information required for accessing an information source. The template for declaring an information source is shown in Figure 12. The <unique identifier> provides a term for referring to a specific information source. The <name> of an information source might not be unique since you may have multiple instances of the same information source running on different hosts. The <host> is the name of the machine where the information source is running. The <initialization function> and <termination function> are optional arguments that specific functions for starting and stopping information sources (if SIMS has control over them). The <transaction function> is the function for sending a command to the information source and will be passed the command that the information source is supposed to process.

```
(define-information-source <unique identifier>
      :name '<information source name>
      :host '<information source host>
      :init-fn <initialization function (optional)>
      :term-fn <termination function (optional)>
      :transaction-fn <transaction function>)
```

Figure 12: Template for Defining an Information Source

Figure 13 shows an instantiated declaration for a local loom knowledge base.

```
(define-information-source patient-kb
      :name 'geo
      :host 'kb1
      :init-fn #'(lambda ()(format t "local-geo-kb initialized"))
      :term-fn #'(lambda ()(format t "local-geo-kb terminated"))
      :transaction-fn #'(lambda (trans)
                            (info-source-op #'execute-loom-trans trans
                                            :kb "MANUAL-KB")))
```

Figure 13: Example Definition of a Loom Knowledge Base

Figure 14 shows an instantiated declaration for an relational database that is accessed through a LIM wrapper.

```
(define-information-source patient-db
      :name 'geo
      :host 'isd10.isi.edu
      :init-fn #'(lambda ()(lim-open-db :db-name 'geo))
      :term-fn #'(lambda ()(lim-close-db 'geo))
      :transaction-fn #'(lambda (trans)
                            (info-source-op #'execute-lim-trans trans
                                            :server-name "ISI-GEO-SERVER")))
```

Figure 14: Example Definition of a Oracle Database using the LIM Wrapper

5 Information-Source Wrappers and Communication

Once the SIMS planner has selected the desired sources for a user's query and devised a plan for obtaining (or updating) the required information, it must communicate with the individual information sources. In order to modularize this process and cleanly separate query planning from communication issues, SIMS requires that for each type of information source there exist a wrapper with which it will communicate. The wrapper must be capable of translating between the SIMS query language and the information source's query language, as well as between the data output format of the information source and a format appropriate for SIMS.

This section explains how wrappers are used by SIMS. The communication flow and protocols used will be described as well.

5.1 Information Source Wrappers

An information source's wrapper will receive as input a restricted form of the SIMS query language (described in Section 2). The restiction is that all concepts and roles used in the query will be drawn *only* from that information source's model. Note that at the time when such communication takes place SIMS has already determined that the query being sent to the information source can be processed in its entirety by that source alone.

The wrapper converts a SIMS query into a query in information source's query language. It submits it to the source and returns to SIMS a list of tuples corresponding to the variable parameters used in the submitted query.

To standardize the use of information source wrappers, SIMS contains a function, `info-source-op` that makes the programmatic interface more standardized and performs the actions necessary to contact a server. This function creates Loom instances when that will be required for subsequent SIMS processing. It issues the remote KQML request if the information source is a remote server. Figure 15 illustrates the relationship between SIMS, `info-source-op`, the wrappers and the information sources.

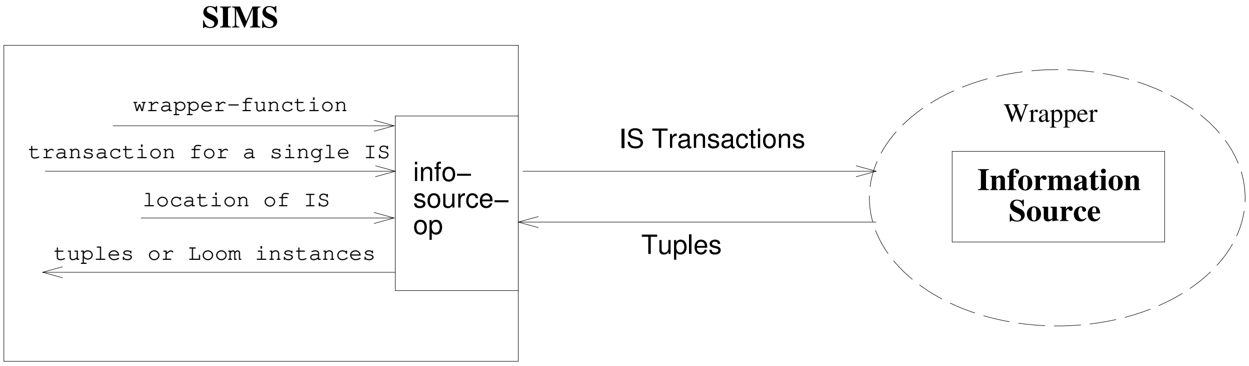

Figure 15: Communication Between SIMS and Information Sources

`info-source-op` is called with the following arguments:

```
(info-source-op wrapper-fn transaction &key server-name kb)
```

`wrapper-fn` A function that will be called with `query` as an argument, to produce an appropriate statement in the information system's query language and satisfy the transaction. This is the actual wrapper for the information source.

`query` The query. An expression of the form `(<type> <output arg> <query body>)` where `<type>` may be one of the following: `retrieve`, `insert`, `delete`, or `update`.

`server-name` If supplied, this is the name of the remote server, a string. If absent, the query will be executed on a local information source.

`kb` If supplied, the name of a knowledge base in which the query should be processed. If absent, the query will be processed in the current knowledge base.

The function will return a list of tuples which may contain Loom instances if so specified in the `<output args>` (see below). `info-source-op` should be used in the information source declaration for specifying the `transaction-fn` function (see Section 4.2).

For example, consider the simple SIMS query:

```
(sims-retrieve (?id ?fname ?lname)
               (:and (Patient ?patient)
                     (Patient-id ?patient ?id)
                     (First-name ?patient ?fname)
                     (Last-name ?patient ?lname)))
```

Assuming that this information is available from a single LIM information source, it will ultimately generate the information source query:

```
(info-source-op #'execute-lim-trans
                 '(retrieve (?id ?fname ?lname)
                      (:and (Patients ?patient)
                            (Patients.patientid ?patient ?id)
                            (Patients.firstname ?patient ?fname)
                            (Patients.lastname ?patient ?lname)))
                 :server-name "ISI-GEO-SERVER")
```

This is the level at which SIMS interacts with the information source server. It is the task of the information source's wrapper (in this case `#'execute-lim-trans`) to process the query beyond this point.

5.1.1 Returning Loom Instances

An added complication may arise when the SIMS query specifies that a *Loom instance* must be returned, and not simple data. Loom instances are sometimes required for further SIMS processing. Instances need to be returned when one of the output arguments in the query is a variable bound to a model concept. That is the case, for example, for the variable `?patient` in the following query:

```
(sims-retrieve (?patient ?id ?fname ?lname)
               (:and (Patient ?patient)
                     (Patient-id ?patient ?id)
                     (First-name ?patient ?fname)
                     (Last-name ?patient ?lname)))
```

As there is no need to build such a capability into each and every wrapper, we have chosen to include this functionality in `info-source-op`. This function strips off the concept variables from the output args list of the query, passes only the reformulated query to the information source wrapper, and later creates appropriate Loom instances and adds them to the data returned from the wrapper.

5.2 Remote Communication Using KQML

If an information source server is loaded into the running SIMS environment, a straight function call to the appropriate information source wrapper function is sufficient to process a query. For communication with servers running on remote hosts, SIMS uses the Knowledge Query and Manipulation Language (KQML)

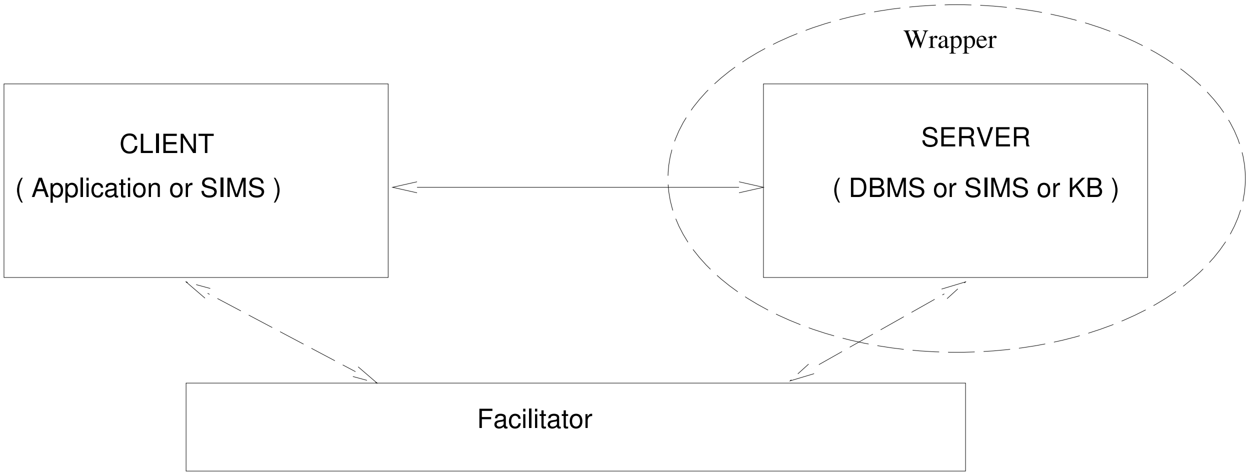

Figure 16: Communication via KQML

protocol [2]. KQML is a language for communication and knowledge sharing between autonomous programs. A simplified view of KQML-based communication is presented in Figure 16.

For our purpose, KQML provides two main types of functionality that ease the communication between clients (application programs using SIMS or SIMS itself) and servers. KQML provides a flexible standard language for client-server communication that is available for many platforms as well as implementation in different languages. It also provides a registry of all clients and servers so that a client only need to refer to the name registered on the registry by the server (which is usually the name of the service provided and hence more meaningful than just a host address) to communicate with the servers.

The central registry of services in KQML is called the *facilitator*, and it records all KQML clients and their addresses. One can query the facilitator for services available as well as other information, we are mostly interested in the facilitator for providing the addresses of information source servers SIMS need to communicate with. (This address resolution process happens transparently and does not require user intervention.) The client and server must both be registered with the facilitator. The global variable kqml::*local-facilitator-name* specifies where the facilitator is located and both the client and server should agree on a facilitator accessible to both. A server registers itself by executing the command:

```
(KQML:START-KQML <server name>)
```

The <server name> must be unique. Clients are started by invoking (start-kqml-client). This will register the client using a unique name of the following form, <user><host>-<timestamp>, where timestamp is gotten from (get-universal-time). Information servers must, of course, also be declared in SIMS, as described in Section 4.2.

The way to check what is available on a facilitator, or to verify that a service that was registered is up, is to issue the command (display-kqml-clients) in Lisp. Or using telnet:

```
> telnet <facilitator host> 5500
...<telnet msgs>...
(ask-all :content "" :reply-with t)
...<list of services registered>...
```

Note that the KQML clients/servers only contact the facilitator once to verify the existence of a server and to get its address. The user need not know where a particular server is located but only its name (e.g., UNISYS-GEO-SERVER). KQML resolves the location (through the facilitator) transparently and caches it. The communication protocol used by KQML is TCP/IP. It creates a process that listens on a remote TCP/IP stream to detect messages from remote hosts.

The facilitator used by KQML must be accessible to both SIMS and to any users of the SIMS system but need not be run on those systems itself. To run a facilitator at a site, execute the following command in a Unix shell:

```
kqml/C/bin/facilitator &
```

The client must know the messages supported by the server as only those can be processed. In KQML terms, SIMS acts as a mediator between the SIMS client and the information sources. A KQML mediator receives a request and either delegates it to one or more other servers, or processes it internally/locally (e.g,. in a local database). Hence the information source server needs to define a handler for the messages it will support and the client needs to know these messages and their form.

SIMS currently use only the `:ASK-ONE` KQML performative to communicate between with remote information source servers. This minimizes the number of handlers the information source KQML servers need to define. The handler will receive the following arguments:

```
(<transaction fun> <query> <kb>)
```

`<query>` is of the form `(<type> <output  args> <query body>)`. The handler should check that `<type>` is one of the supported operations (i.e., retrieve, insert, update, and delete). The `<kb>` argument specifies the knowledge base in which to execute the query and is optional.

Here is a simple `:ASK-ONE` handler that simply evaluates the expression received from the client:

```
(kqml::define-handler (ask-one)
   ;; content is expected to be of the form (<exec fn> <query>)
   (let* ((content (kqml::msg-content message))
          (exec-fn (car content))
          (query (cdr content))))

     (when (member (car query) '(retrieve update delete insert))
      (kqml::make-msg 'reply (apply exec-fn query))))
```

6 Running SIMS

SIMS can be run either by issuing commands to the Lisp listener or using the graphical interface. The commands that are available in both modes are described in this section.

6.1 Top-Level Commands

The top level commands of SIMS can be classified in six groups: query execution, transactions, active notification, query set management, information source management, and tracing.

6.1.1 Query Commands

The main command to execute a query is:

(sims-retrieve `<parameter-list>` `<query-exp>`)

A complete description of the query syntax is provided in Section 2. A pre-stored query (see query set management subsection) can be executed with:

(run-query `<num>`)

6.1.2 Transaction Commands

SIMS supports insert, delete, and update transactions as explained in Section 2.

(sims-insert `<parameter-list>` `<instance-exp>`) Creates new instances in the information sources with roles and values as given by `<instance-exp>`, which may be expressed in domain terms.

(sims-delete `<parameter-list>` `<instance-exp>`) Deletes the instances from the information sources specified by the (domain or source) expresion `<instance-exp>`,

(sims-update `<parameter-list>` `<instance-exp>`) Changes values of some roles for the source instances corresponding to the (domain or source) `<instance-exp>`.

6.1.3 Active Notifications

Clients can ask SIMS to monitor certain data items with specified conditions and actions. SIMS will execute these actions and send a notification via TCP/IP to the client whenever the data items experience a change to a state that satisfies the specified conditions.

(sims-begin-notify `:concept <concept-name> :when <sims-query> :do <function>`
`:host <address> :port <port>`)
When the `<sims-query>` is satisfied over the concept
`<concept-name>`, `<function>` is executed and the results sent to the `<port>` of the machine identified by `<address>`. This functions returns an identifier for each notification currently in the system.

(sims-end-notify `<notification-id>`) Informs SIMS to stop monitoring the activities specified in the notification with `<notification-id>`.

6.1.4 Query Set Management

Sometimes it is convenient to have frequently used queries stored in the system. A query set can be predefined by setting the global variable `*queries*` to the list of queries. This query set can also used from the graphical interface described in the next section.

(list-queries) Provides a list of the numbers of predefined queries.

(load-comment `<num>`) Retrieves the comment for query `<num>`.

(load-query `<num>`**)** Retrieves query `<num>`.

(plan-query `<num>`**)** Generates the plan for performing query `<num>`), but does not execute it.

(run-query `<num>`**)** Executes query `<num>`.

(run-queries `&optional <dont-run>`**)** Sequentially executes all queries in `*queries*` except the numbers in the `<dont-run>` list.

The syntax for the query set is:

```
(setq *queries* '(
  (<num> <comment> <query>)
  (<num> <comment> <query>)
  .
  .
  .))
```

Here is an example of a query set:

```
(setq *queries* '(
  (1 "List all patients"

     (sims-retrieve (?id ?fname ?lname ?dob ?doctor)
                    (:and (patient ?patient)
                          (patient-id ?patient ?id)
                          (first-name ?patient ?fname)
                          (last-name ?patient ?lname)
                          (date-of-birth ?patient ?dob)
                          (doctor-name ?patient ?doctor)))
     )

  (3  "Which patient is in room 101"

      (sims-retrieve (?fname ?lname)
                     (:and (patient-room ?room)
                           (room-nm ?room 101)
                           (used-by-patient ?room ?patient)
                           (first-name ?patient ?fname)
                           (last-name ?patient ?lname)))
      )
  ))
```

6.1.5 Information Source Management

The commands for manipulating the information sources are:

(list-sources) Lists all of the declared information sources.

(available-sources) Lists all of the currently available information sources that the system can access.

(initialize-source `<unique-id>`**)** Initializes the given information source.

(initialize-all-sources) Initializes all defined information sources.

(close-source `<unique-id>`**)** Closes the given information source.

(close-all-sources) Closes all of the defined information sources.

6.1.6 Tracing

In order to facilitate debugging and show the behavior of the system in a greater detail, the following commands instruct SIMS to print additional information about its processing.

(sims-trace-on) Turns on tracing. SIMS prints additional information on the query planning and execution, such as plan steps, partial reformulations, information sources accessed, intermediate results, etc.

(sims-trace-off) Turns off tracing.

(sims-trap-on) SIMS traps all errors (returning nil at the end of execution if the errors prevented the successful execution).

(sims-trap-off) When an error occurs in the processing, SIMS allows the original error handler to interrupt the execution. This command is useful when debugging an application.

6.2 The Graphical Interface

This section describes how to interact with SIMS through its graphical user interface.

The graphical interface to SIMS is invoked by calling the function `sims`, which has the optional keyword `:host`, used when the display is different from that of the machine in which SIMS is executed. Examples of invocation are: `(sims)`, or `(sims :host "sunstruck.isi.edu")` to display the interface on the machine `sunstruck.isi.edu`.

The SIMS interface (see Figure 17) is divided into three main panes: the Interaction/Trace pane (lower right quadrant), the Query pane (lower left quadrant), and the Graph pane (upper half).

The user issues commands either by selecting a command in the Command menu or typing the command in the Interaction/Trace pane. Command completion is supported. This is achieved by typing the first few keystrokes of a command and if unique, a space will complete it. Typically, an interaction sequence will proceed as follows (note that in the example below the commands can be the menu commands or typed in ones):

1. Select `Load Query` and choose a query to solve. The chosen query will be displayed in the Query pane.
2. To produce a plan to solve the query, select `Plan Query`. A graph of the generated plan will be displayed in the graph pane.
3. To perform the actual retrieval, once a plan has been generated, select `Execute Plan`. The appropriate plan graph will be shown in the Graph pane and the state of the execution is indicated by highlighting the currently executing node in the graph. The final answer will be displayed in the Interaction/Trace pane.

6.2.1 Graphical Interface Commands

Load Query Brings up a menu of the set of queries currently loaded in the system (in the variable `*queries*`). This is the query that will be used by `Plan Query` and `Solve`. The selected query will be displayed in the bottom left pane.

Edit Query Once a query has been input to SIMS, we may want to issue a similar one. It is often faster to modify a loaded query than to retype one from scratch. This command allows the user to edit the currently selected query. Several checks are made to verify the consistency of the query. For example, concepts and roles must be defined in the SIMS knowledge base.

Set Current Query Allows the user to set the query to be processed by the interface by allowing the user to type it in the interaction/trace pane.

Text Edit Query Starts a text editor (emacs) to freely edit/input a query.

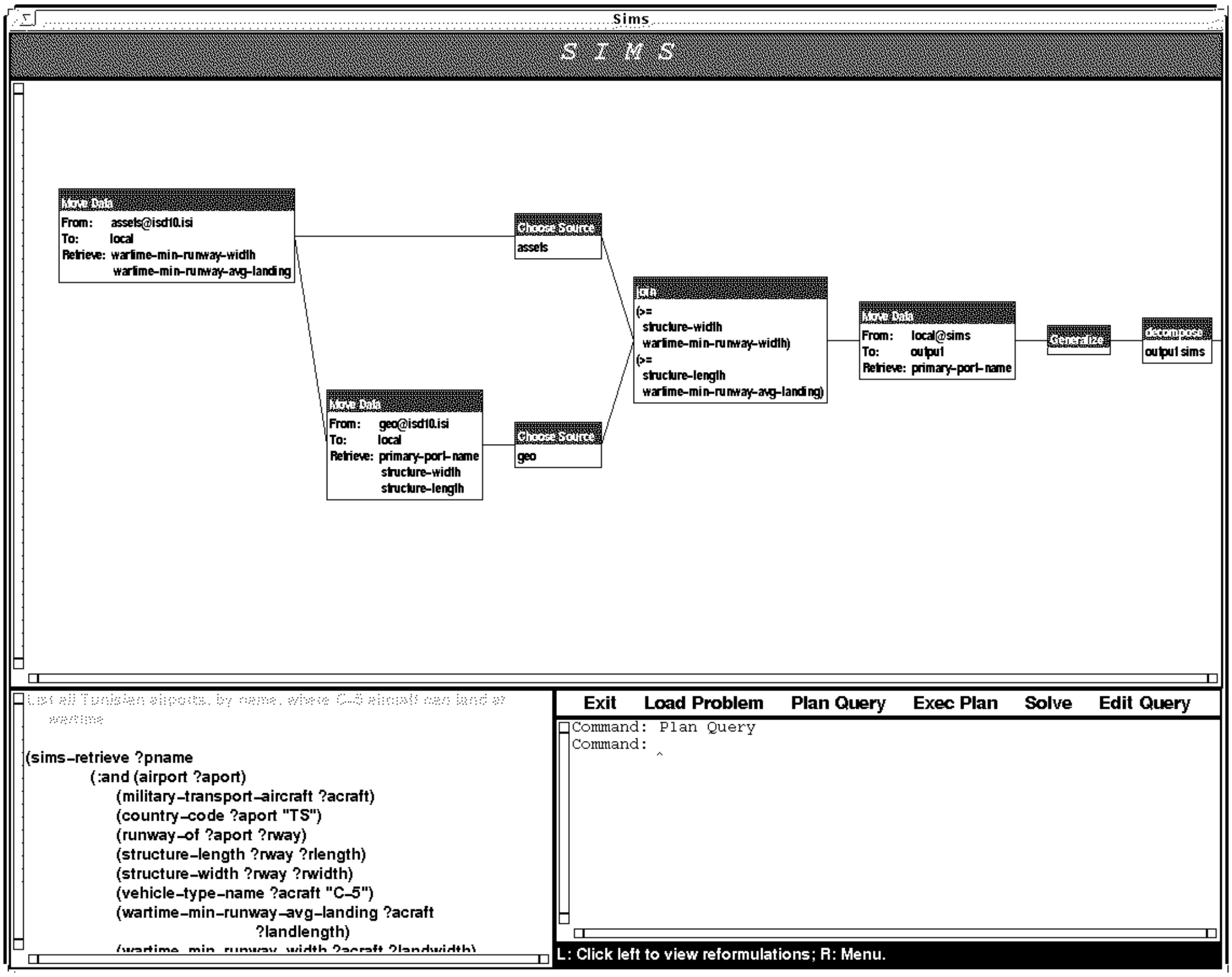

Figure 17: SIMS graphical interface

Plan Query Generates a plan for the current query. The query could have been loaded using `Load Problem`, or directly. If a plan is found, its graph will be drawn in the graph pane.

Execute Plan Executes the current plan. The node currently being executed will be highlighted (reverse video) while executed and un-highlighted when done. The final answer will be displayed in the Interaction/Trace pane.

Solve Problem Combines the previous two steps, that is, generates the plan for the currently selected query and executes it.

Exit Quits SIMS.

Display Answers Displays the results of the last executed query. The user is prompted for the number of retrieved data items to display - the default is to display all.

! <expr> Allows the evaluation of <`expr`>, that is, it will behave like a Lisp listener.

6.3 Plan Cost Evaluation

The SIMS architecture allows the user to change the policy of the generation of query access plans to account for different cost models. The function `set-evaluation-function` establishes the function that will guide this generation.

Currently, SIMS provides two functions. The first one, `ucpop::evaluate-plan-cost`, generates plans with the minimum number of steps. The second one, `ucpop::evaluate-plan-cost-by-size`, produces query plans in which the size of intermediate data transmitted from the information sources and processed in local joins is minimized. It uses a series of traditional database techniques to estimate the size of the queries. It considers both the expected number of tuples that a query will produce and the projection attributes. In order to calculate this estimate, it uses some statistics computed from the current contents of the information sources, such as, number of instances of a concept, number of distinct values (present in the source) of an attribute, and maximum and minimum values for numeric attributes.

Generally, `ucpop::evaluate-plan-cost-by-size` both improves the efficiency of the planning process (2 to 5-fold speed-up) and the quality of the generated plans. For complex queries this should be the function of choice. For simple queries the performance of both functions is similar. Nevertheless, size estimation needs statistics that may or may not be available for some sources. The function `gather-stats-by-info-source` will generate a set of files (one per information source) with the computed statistics for the current domain. These files can then be loaded as desired into SIMS, or incorporated into the defsystem definition of the current model to be loaded automatically.

In summary,

- to use `ucpop::evaluate-plan-cost` (the default), evaluate:

  `> (set-evaluation-function #'ucpop::evaluate-plan-cost)`

- to use `ucpop::evaluate-plan-cost-by-size`, evaluate:

  `> (set-evaluation-function #'ucpop::evaluate-plan-cost-by-size)`

- to create the statistics files, evaluate:

  `> (gather-stats-by-info-source)`

7 Trouble Shooting

What do you do once you have built the wrappers for your information sources, defined the domain and information source models, and submitted the first query to SIMS only to find that it does not work? We recommend that you first test your system incrementally from the bottom up by testing the wrappers to the information sources, then testing the source-level queries, and finally testing your domain-level queries. This section describes each of these in turn:

7.1 Testing the Information-Source Wrappers

Before invoking SIMS, individual wrappers for all of the information sources that are to be used should be thoroughly tested. Each wrapper should accept a source-level query as input and return a set of tuples that are the answer to that query. To test the individual wrappers, invoke the access-function for each wrapper that is defined in Section 4.2. For example the access function for a LIM Oracle database is:

```
#'(lambda (query)
       (info-source-op 'lim::execute-lim-query
                       (second query)
                       (third query)))
```

This access function can be tested directly by defining it as a function:

```
(defun lim-wrapper (query)
       (info-source-op 'lim::execute-lim-query
                       (second query)
                       (third query)))
```

Then call this function on a source-level query:

```
(lim-wrapper '(retrieve (?id ?lname ?dob)
                        (:and (patients ?patient)
                              (patients.patientid ?patient ?id)
                              (patients.lastname ?patient ?lname)
                              (patients.dob ?patient ?dob))))
(("P1001" "Okumura" 20561) ("P1002" "DeSpain" 120442)
 ("P1003" "Kumar" 63065) ("P1004" "Cooper" 101030)
 ("P1005" "Brown" 30152) ("P1006" "Smith " 71570)
 ("P1007" "Smith" 91851) ("P1008" "Chame " 12745)
 ("P1009" "Kayano" 111740) ("P1010" "Hamilton" 30359)
 ("P1011" "Hammer" 63077) ("P1012" "Dosek" 41563)
 ("P1013" "Wills" 51772) ("P1014" "Richardson" 12268)
 ("P1015" "Mizushima" 40761))
```

If this does not return the expected data, one must determine the cause of the problem and fix it before continuing to the next step.

7.2 Testing the Source-level Queries

Once all of the wrappers are working correctly, it is time to begin testing the source-level queries in SIMS. The first thing to test are exactly the same source-level queries that were used to test the individual wrappers. This will ensure that the SIMS model of the information source and the actual information source are in sync.

```
(sims-retrieve (?id ?lname ?dob)
                        (:and (patients ?patient)
                              (patients.patientid ?patient ?id)
                              (patients.lastname ?patient ?lname)
                              (patients.dob ?patient ?dob)))

UCPOP Stats: Initial terms = 3 ;   Goals = 4 ;  Success (1 steps)
      Created 11 plans, but explored only 9
      CPU time:    0.0200 sec
      Branching factor:  1.111
      Working Unifies: 23
```

```
      Bindings Added: 22
(("P1001" "Okumura" 20561) ("P1002" "DeSpain" 120442) ("P1003" "Kumar" 63065) ("P1004" "Cooper" 101030)
("P1005" "Brown" 30152) ("P1006" "Smith " 71570) ("P1007" "Smith" 91851) ("P1008" "Chame " 12745) ("P1009"
"Kayano" 111740) ("P1010" "Hamilton" 30359) ("P1011" "Hammer" 63077) ("P1012" "Dosek" 41563) ("P1013"
"Wills" 51772) ("P1014" "Richardson" 12268) ("P1015" "Mizushima" 40761))
```

If you get the message:

```
>>Error: No information sources are currently available!
```

that means that the information sources were not initialized in SIMS. Do so by using the **initialize-information-source** commands:

```
(initialize-information-source 'room-kb)
local-assets-kb initialized
NIL
49
> (initialize-information-source 'patient-kb)
local-geo-kb initialized
NIL
50
```

One should also test source-level queries in SIMS that span several information sources. After testing all of the individual source-level concepts, proceed to testing the domain-level queries.

7.3 Testing the Domain-level Queries

If all the models were set up correctly, domain-level queries should execute correctly without any problems. However, people often make mistakes in constructing the models, resulting in the system failing to produce the expected results.

Upon discovering a problem, the first thing to do is to examine the relevant portion of the domain model, source model, and IS-links to see if there are any obvious errors (e.g., missing links, misspellings, etc). Correct any obvious mistakes and try again. Note that Loom often gets confused if the same concept is defined twice, so it may be best to restart things after making changes to the model.

If a complex query does not execute correctly, break it up into smaller units and test the individual parts. That will help to pinpoint the source of the problem more quickly. For example, if the following query fails:

```
(SIMS-RETRIEVE (?FNAME ?LNAME)
               (:AND (HOSPITAL-ROOM ?ROOM)
                     (ROOM-NM ?ROOM 101)
                     (PID ?ROOM ?ID)
                     (PATIENT ?PATIENT)
                     (PATIENT-ID ?PATIENT ?ID)
                     (FIRST-NAME ?PATIENT ?FNAME)
                     (LAST-NAME ?PATIENT ?LNAME)))
```

the next thing to do is to break it into smaller queries:

```
(SIMS-RETRIEVE (?FNAME ?LNAME)
               (:AND (PATIENT ?PATIENT)
                     (FIRST-NAME ?PATIENT ?FNAME)
                     (LAST-NAME ?PATIENT ?LNAME)))
```

After identifying the simplest query that fails, go back and examine the relevant portions of the model to see if it is correct. If so, the next step is to see if the system can generate a plan for the query. This is done using the **plan-query** command:

```
(plan-query '(SIMS-RETRIEVE (?FNAME ?LNAME)
               (:AND (PATIENT ?PATIENT)
                     (FIRST-NAME ?PATIENT ?FNAME)
                     (LAST-NAME ?PATIENT ?LNAME))))

Sources: ((IS-AVAILABLE ASSETS KB2) (IS-AVAILABLE GEO KB1))

UCPOP Stats: Initial terms = 2 ;   Goals = 4 ;  Success (2 steps)
      Created 29 plans, but explored only 18
      CPU time:    0.0600 sec
```

```
      Branching factor:  1.389
      Working Unifies: 68
      Bindings Added: 65
Step 1  :
(UCPOP::MOVE GEO
             KB1
             UCPOP::OUTPUT
             (RETRIEVE (?FNAME ?LNAME)
                       (:AND (PATIENTS ?PATIENT)
                             (PATIENTS.FIRSTNAME ?PATIENT ?FNAME)
                             (PATIENTS.LASTNAME ?PATIENT ?LNAME))))
Step 2  :
(UCPOP::SELECT-SOURCE UCPOP::OUTPUT
                      UCPOP::SIMS
                      (RETRIEVE (?FNAME ?LNAME)
                                (:AND (PATIENT ?PATIENT)
                                      (FIRST-NAME ?PATIENT ?FNAME)
                                      (LAST-NAME ?PATIENT ?LNAME)))
                      (RETRIEVE (?FNAME ?LNAME)
                                (:AND (PATIENTS ?PATIENT)
                                      (PATIENTS.FIRSTNAME ?PATIENT ?FNAME)
                                      (PATIENTS.LASTNAME ?PATIENT ?LNAME)))
                      GEO)
#plan<S=3; O=0; U=0>
```

If this fails to generate a plan, then either the required sources are not available or there is still a problem with the model. If a plan is generated, but the correct data is not returned, then tracing of the execution needs to be turned on to help pinpoint the error.

```
(sims-trace-on)
```

Now rerun the problem query and the execution trace will print out each action as it is executed and the results of the individual steps. From this trace it should be possible to figure out which step or steps are failing to return the expected data.

If a steps fails during execution, by default SIMS will simply print the error message and then exit. It traps these errors so that it can attempt to replan failed actions. However, you can turn the error trapping off to investigate further using the command:

```
(sims-trap-off)
```

Now instead of trapping the error, the system will drop into the error handler and one can proceed to debug the problem.

If all else fails, create the simplest version of the domain model, source models, and query that reproduce the problem and send them to `sims-bug-report@isi.edu`.

8 Installation and System Requirements

The SIMS system currently runs in Common Lisp with MCL 2.0 on the Mac and LUCID 4.0 on Unix. We expect to have the system running in Allegro on both the PC and Unix environments shortly. SIMS requires the following software components:

LOOM provides the underlying knowledge representation and programming support. Currently using version 2.0

KQML provides remote communication support between remote DB servers and SIMS. It can also be used to communicate between multiple SIMS servers.

CLIM provides the graphical user interface. Currently using version 1.1. This component is optional since SIMS can be run without the graphical interface.

LIM provides a wrapper for accessing relational databases. Currently using version 1.1 or 1.3. If you have your own wrapper for your databases, then this is optional.

8.1 Component Structure

Here is our current component and directory structure, which we recommend that users adopt:

```
defsys - for definitions of various components and systems

planner - for the SIMS planner based on UCPOP

operators - for reformulation operators

qsize-eval-fun - evaluation function for the planner

sims-interface - for the user interface files.

domains - for domain and information source models, and queries.

sockets - TCP/IP interface to SIMS (optional)
```

8.2 Define, Load and Compile Components

We use the CMU defsystem (this comes with LOOM) to define each subsystem. Each of the above component has its own system declaration file, e.g., lim.system, planner.system. If you want to define your own subsystem, please see the files in the `defsys` directory for examples.

One can define a component that includes many other components. For example, there is a subsystem called "BASIC-SIMS" that includes: lim, kqml, planner, operators, and interface.

Before you can use the defsystems, you will need to set two global variables. The first variable, `*sims-sys-dir*`, sets the directory for the location of all of the subsystems:

```
(setq *sims-sys-dir* "/home/johndoe/sims/sys/")
```

The second variable, `make::*central-registry*`, sets the location of the defsystem definitions for each of the components:

```
(setq make::*central-registry* "/home/sims/sys/defsys/")
```

One can load a system using the command `make:operate-on-system`. For example, to load "basic-sims", you do:

```
(make:operate-on-system :basic-sims :load)
```

You can substitute the keyword `:load` by `:compile` to compile the system.

```
(make:operate-on-system :basic-sims :compile)
```

You can also force the system to recompile all of the files in a component by appending the `:force` keyword:

```
(make:operate-on-system :basic-sims :compile :force t)
```

The typical sequence of loading SIMS is to load the `basic-sims` first, then load the optional components that you need, and followed by the domain model and information source models that are specific for your application.

For example, after loading in the basic-sims system, you would load the example from the manual as follows:

```
(make:operate-on-system :manual-kbs :load)
```

9 Coded Example

This section gives the code that implements the example discussed throughout the manual.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; domain-model.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Domain model for the example in the SIMS user manual
;;;
(in-package :sims)
(in-kb 'sims-kb)

;;; define concepts

(defconcept PERSON
  :is-primitive
  (:and
        (:all LAST-NAME string)
        (:all FIRST-NAME string)
        (:all DATE-OF-BIRTH number))
  :annotations ((key (last-name))))

(defconcept PATIENT
  :is-primitive
  (:and PERSON
        (:all PATIENT-ID string)
        (:all PATIENT-OF DOCTOR)
        (:all DOCTOR-NAME string))
  :annotations ((key (patient-id))))

(defconcept ELDERLY-PATIENT
  :is
  (:satisfies (?p)
      (:for-some (?dob)
        (:and (PATIENT ?p)
              (DATE-OF-BIRTH ?p ?dob)
              (older-than-65 ?dob)))))

(defconstant *YEAR* 95)

(defconcept older-than-65 :is
  (:and Number
        (:predicate (dob)
            (<= 65 (- *YEAR* (- dob (* 100 (truncate (/ dob 100)))))))))

(defconcept DOCTOR
  :is-primitive
  (:and PERSON
        (:all DOCTOR-ID string))
  :annotations ((key (doctor-id))))

(defconcept ROOM
  :is-primitive
  (:all ROOM-NM number)
  :annotations ((key (room-nm))))
```

```
(defconcept HOSPITAL-ROOM
  :is-primitive
  (:and ROOM
        (:all USED-BY-PATIENT PATIENT)
        (:all PID string))
  :annotations ((key (room-nm))))

(defconcept OFFICE
  :is-primitive
  (:and ROOM
        (:all OFFICE-USER DOCTOR))
  :annotations ((key (room-nm))))

;;; define relations

(defrelation LAST-NAME
  :domain PERSON
  :range string)

(defrelation FIRST-NAME
  :domain PERSON
  :range string)

(defrelation DATE-OF-BIRTH
  :domain PERSON
  :range number)

(defrelation PATIENT-ID
  :domain PATIENT
  :range string)

(defrelation PATIENT-OF
  :domain PATIENT
  :range DOCTOR)

(defrelation DOCTOR-NAME
  :domain PATIENT
  :range string)

(defrelation DOCTOR-ID
  :domain DOCTOR
  :range string)

(defrelation ROOM-NM
  :domain ROOM
  :range number)

(defrelation PID
  :domain HOSPITAL-ROOM
  :range string)

(defrelation USED-BY-PATIENT :is
  (:satisfies (?room ?patient)
             (:FOR-SOME (?pid)
                        (:AND (hospital-room ?room)
                              (patient ?patient)
                              (pid ?room ?pid)
                              (patient-id ?patient ?pid)))))

(defrelation OFFICE-USER
  :domain OFFICE
  :range PERSON)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; queries.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Example queries
;;;
(in-package "SIMS")

(setq *queries* '(

 (1 "List all patients"

 (SIMS-RETRIEVE (?id ?FNAME ?LNAME ?dob ?doctor)
                 (:AND (PATIENT ?patient)
                     (patient-id ?patient ?id)
                     (FIRST-NAME ?PATIENT ?FNAME)
                     (LAST-NAME ?PATIENT ?LNAME)
                     (date-of-birth ?patient ?dob)
                     (doctor-name ?patient ?doctor)))
 )

 (2  "Which patient is in room 101"

 (SIMS-RETRIEVE (?FNAME ?LNAME)
               (:AND (HOSPITAL-ROOM ?ROOM)
                     (ROOM-NM ?ROOM 101)
                     (PID ?ROOM ?ID)
                     (PATIENT ?PATIENT)
                     (PATIENT-ID ?PATIENT ?ID)
                     (FIRST-NAME ?PATIENT ?FNAME)
                     (LAST-NAME ?PATIENT ?LNAME)))
 )

 (3  "Which patient is in room 101"

 (SIMS-RETRIEVE (?FNAME ?LNAME)
               (:AND (HOSPITAL-ROOM ?ROOM)
                     (ROOM-NM ?ROOM 101)
                     (USED-BY-PATIENT ?ROOM ?PATIENT)
                     (FIRST-NAME ?PATIENT ?FNAME)
                     (LAST-NAME ?PATIENT ?LNAME)))
 )

(4 "List elderly patients"

(SIMS-RETRIEVE (?FNAME ?LNAME ?dob)
                 (:AND (ELDERLY-PATIENT ?patient)
                     (FIRST-NAME ?PATIENT ?FNAME)
                     (LAST-NAME ?PATIENT ?LNAME)
                     (DATE-OF-BIRTH ?patient ?dob)))
)
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; source-model.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Example source model for SIMS manual
;;;
(in-package :sims)
(in-kb 'sims-kb)

;; define concepts

(defconcept PATIENTS
  :is-primitive
  (:and
        (:the PATIENTS.patientid String)
        (:the PATIENTS.lastname String)
        (:the PATIENTS.firstname String)
        (:the PATIENTS.dob Number)
        (:the PATIENTS.doctor String))
  :mixin-classes (info-source-class)
  :annotations ((Info-Source geo)))

(defconcept ROOM_PATIENT
  :is-primitive
  (:and
        (:the ROOM_PATIENT.room Number)
        (:the ROOM_PATIENT.patient String))
  :mixin-classes (info-source-class)
  :annotations ((Info-Source assets)))

;; define relations

(defrelation PATIENTS.patientid
  :domain PATIENTS)

(defrelation PATIENTS.lastname
  :domain PATIENTS)

(defrelation PATIENTS.firstname
  :domain PATIENTS)

(defrelation PATIENTS.dob
  :domain PATIENTS)

(defrelation PATIENTS.doctor
  :domain PATIENTS)

(defrelation ROOM_PATIENT.room
  :domain ROOM_PATIENT)

(defrelation ROOM_PATIENT.patient
  :domain ROOM_PATIENT)

;; define IS-links

(def-IS-links patients patient
    ((PATIENTS.patientid PATIENT-ID)
     (PATIENTS.lastname LAST-NAME)
     (PATIENTS.firstname FIRST-NAME)
     (PATIENTS.dob DATE-OF-BIRTH)
     (PATIENTS.doctor DOCTOR-NAME)))

(def-IS-links ROOM_PATIENT HOSPITAL-ROOM
    ((ROOM_PATIENT.room ROOM-NM)
     (ROOM_PATIENT.patient PID)))
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; kb-source-defs.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Definition of the Loom Knowledge Base
;;;
(in-package :sims)
(in-kb 'sims-kb)

(define-information-source patient-kb
      :name 'geo
      :host 'kb1
      :term-fn #'(lambda ()(format t "local-geo-kb terminated"))
      :init-fn #'(lambda ()(format t "local-geo-kb initialized"))
      :transaction-fn #'(lambda (trans) (info-source-op #'execute-loom-trans trans)))

(define-information-source room-kb
      :name 'assets
      :host 'kb2
      :term-fn #'(lambda ()(format t "local-assets-kb terminated"))
      :init-fn #'(lambda ()(format t "local-assets-kb initialized"))
      :transaction-fn #'(lambda (trans) (info-source-op #'execute-loom-trans trans)))


(initialize-information-source 'patient-kb)
(initialize-information-source 'room-kb)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Knowledge-Base Data
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(tell (:about patients145439 patients
      (patients.doctor "Hotz")
      (patients.dob 30359)
      (patients.firstname "Sheila")
      (patients.lastname "Hamilton")
      (patients.patientid "P1010")))
(tell (:about patients145431 patients
      (patients.doctor "Berson")
      (patients.dob 63077)
      (patients.firstname "Janice")
      (patients.lastname "Hammer")
      (patients.patientid "P1011")))
(tell (:about patients145438 patients
      (patients.doctor "Gutierrez")
      (patients.dob 40761)
      (patients.firstname "Dana")
      (patients.lastname "Mizushima")
      (patients.patientid "P1015")))

      .
      .
      .


(tell (:about room_patient145541 room_patient
      (room_patient.patient "P1015")
      (room_patient.room 101)))
(tell (:about room_patient145543 room_patient
      (room_patient.room 107)))
(tell (:about room_patient145536 room_patient
      (room_patient.patient "P1010")
      (room_patient.room 116)))

      .
      .
      .


(tellm)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; db-source-defs.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Definition of the Oracle Databases
;;;
(in-package :sims)

(define-information-source room-db
      :name 'assets
      :host 'isd10.isi.edu
      :term-fn #'(lambda ()(lim-close-db 'assets))
      :init-fn #'(lambda ()(lim-open-db :db-name 'assets))
      :transaction-fn #'(lambda (trans)(info-source-op #'execute-lim-trans trans)))


(define-information-source patient-db
      :name 'geo
      :host 'isd10.isi.edu
      :term-fn #'(lambda ()(lim-close-db 'geo))
      :init-fn #'(lambda ()(lim-open-db :db-name 'geo))
      :transaction-fn #'(lambda (trans)(info-source-op #'execute-lim-trans trans)))

(initialize-information-source 'ROOM-DB)
(initialize-information-source 'PATIENT-DB)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lim-source-model.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Definition of the databases for the LIM Wrapper
;;;
(in-package :sims)
(in-kb 'sdo-kb)

;; define concepts

(defconcept PATIENTS
  :is-primitive
  (:and Db-Concept
        (:the PATIENTS.patientid String)
        (:the PATIENTS.lastname String)
        (:the PATIENTS.firstname String)
        (:the PATIENTS.dob Number)
        (:the PATIENTS.doctor String))
  :attributes :clos-class
  :annotations ((Nulls-Ok-Cols
                 (PATIENTS.lastname
                  PATIENTS.firstname
                  PATIENTS.dob
                  PATIENTS.doctor))
                (Source-Db geo)))

(defconcept ROOM_PATIENT
  :is-primitive
  (:and Db-Concept
        (:the ROOM_PATIENT.room Number)
        (:the ROOM_PATIENT.patient String))
  :attributes :clos-class
  :annotations ((Nulls-Ok-Cols
                 (ROOM_PATIENT.room
                  ROOM_PATIENT.patient))
                (Source-Db assets)))

;; define relations

(defrelation PATIENTS.patientid
  :is-primitive DB-Relation
  :domain PATIENTS
  :annotations ((Source-Db-Table PATIENTS)
                (Source-Db-Column "patientid")
                (Source-Db-Datatype String)))

(defrelation PATIENTS.lastname
  :is-primitive DB-Relation
  :domain PATIENTS
  :annotations ((Source-Db-Table PATIENTS)
                (Source-Db-Column "lastname")
                (Source-Db-Datatype String)))

(defrelation PATIENTS.firstname
  :is-primitive DB-Relation
  :domain PATIENTS
  :annotations ((Source-Db-Table PATIENTS)
                (Source-Db-Column "firstname")
                (Source-Db-Datatype String)))

(defrelation PATIENTS.dob
  :is-primitive DB-Relation
  :domain PATIENTS
  :annotations ((Source-Db-Table PATIENTS)
                (Source-Db-Column "dob")
```

```
                (Source-Db-Datatype Number)))

(defrelation PATIENTS.doctor
  :is-primitive DB-Relation
  :domain PATIENTS
  :annotations ((Source-Db-Table PATIENTS)
                (Source-Db-Column "doctor")
                (Source-Db-Datatype String)))

(defrelation ROOM_PATIENT.room
  :is-primitive DB-Relation
  :domain ROOM_PATIENT
  :annotations ((Source-Db-Table ROOM_PATIENT)
                (Source-Db-Column "room")
                (Source-Db-Datatype Number)))

(defrelation ROOM_PATIENT.patient
  :is-primitive DB-Relation
  :domain ROOM_PATIENT
  :annotations ((Source-Db-Table ROOM_PATIENT)
                (Source-Db-Column "patient")
                (Source-Db-Datatype String)))


(def-key-roles PATIENTS PATIENTS.patientid)
(def-key-roles ROOM_PATIENT ROOM_PATIENT.room)
```

The relational tables present in the databases have the following definitions:

```
*** In the "geo" database:

create table patients
(patientid  char(7) not null,
 lastname   char(15),
 firstname  char(15),
 dob        number,
 doctor     char(15)
);

SQL> describe patients
 Name                            Null?    Type
 ------------------------------- -------- ----
 PATIENTID                       NOT NULL CHAR(7)
 LASTNAME                                 CHAR(15)
 FIRSTNAME                                CHAR(15)
 DOB                                      NUMBER
 DOCTOR                                   CHAR(15)

SQL> select patientid,lastname,firstname, dob, doctor from patients;

PATIENT LASTNAME        FIRSTNAME              DOB DOCTOR
------- --------------- --------------- ---------- ---------------
P1001   Okumura         Benjamin             20561 Fucich
P1002   DeSpain         Ann                 120442 Goldman
P1003   Kumar           Barbara              63065 Tzartzanis
P1004   Cooper          Albert              101030 Jain
P1005   Brown           Anant                30152 Gonzalez
P1006   Smith           Jacqueline           71570 Casner
P1007   Smith           Akitoshi             91851 Tzartzanis
P1008   Chame           Amanda               12745 Gonzalez
P1009   Kayano          Lee                 111740 Goldman
P1010   Hamilton        Sheila               30359 Hotz
P1011   Hammer          Janice               63077 Berson
P1012   Dosek           Thomas               41563 Woolf
P1013   Wills           Daniel               51772 Datta
P1014   Richardson      Vance                12268 Vernier
P1015   Mizushima       Dana                 40761 Gutierrez
```

*** In the "assets" database:

```
create table room_patient;
(room       number,
 patient    char(7)
);

SQL> describe room_patient;
 Name                            Null?    Type
 ------------------------------- -------- ----
 ROOM                            NOT NULL NUMBER
 PATIENT                                  CHAR(7)

SQL> select * from ROOM_PATIENT;

      ROOM PATIENT
---------- -------
       101 P1015
       102 P1002
       103 P1001
       104
       105 P1008
       106 P1011
       107
       108 P1014
       109 P1005
       110 P1007
       111 P1009
       112 P1013
       113 P1012
       114 P1004
       115
       116 P1010
       117
       118
       119 P1006
       120 P1003
```

10 Additional Reading

Using this manual and following the instructions in it require familiarity with SIMS, as well as with the Loom knowledge representation language, the LIM/IDI system for accessing remote information sources, and the KQML transport protocol.

The following papers may be consulted for further information about these programs.

10.1 SIMS

1. Arens, Y., Chee, C.Y., Hsu, C-N., and Knoblock, C.A. 1993. Retrieving and Integrating Data from Multiple Information Sources. In *International Journal of Intelligent and Cooperative Information Systems.* Vol. 2, No. 2. Pp. 127-158.

2. Arens, Y., Knoblock, C.A., and Shen W-M. Query Reformulation for Dynamic Information Integration, Submitted to *Journal of Intelligent Information Systems.*

3. Arens, Y. and Knoblock, C.A. 1994. Intelligent Caching: Selecting, Representing, and Reusing Data in an Information Server. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM-94)*, Gaithersburg, MD.

4. Arens, Y. and Knoblock, C.A. 1992. Planning and Reformulating Queries for Semantically-Modeled Multidatabase Systems, *Proceedings of the First International Conference on Information and Knowledge Management (CIKM-92)*, Baltimore, MD.

5. Hsu, C-N., and Knoblock, C.A. 1995. Estimating the Robustness of Discovered Knowledge, in *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95)*, Montreal, Quebec, Canada.

6. Hsu, C-N., and Knoblock, C.A. 1995. Using inductive learning to gen- erate rules for semantic query optimization. In Gregory Piatetsky-Shapiro and Usama Fayyad, editors, *Advances in Knowledge Discovery and Data Mining*, chapter 17. MIT Press.

7. Hsu, C-N., and Knoblock, C.A. 1994. Rule Induction for Semantic Query Optimization, in *Proceedings of the Eleventh International Conference on Machine Learning (ML-95)*, New Brunswick, NJ.

8. Hsu, C-N., and Knoblock, C.A. 1993. Reformulating Query Plans For Multidatabase Systems. In *Proceedings of the Second International Conference of Information and Knowledge Management (CIKM-93)*, Washington, D.C.

9. Knoblock, C.A., Arens, Y. and Hsu, C-N. 1994. An Architecture for Information Retrieval Agents. In *Proceedings of the Second International Conference on Cooperative Information Systems*, University of Toronto Publications, Toronto, Ontario, Canada.

10. Knoblock, C.A. 1995. Planning, Executing, Sensing, and Replanning for Information Gathering. In *IJCAI-95*, Montreal, Quebec, Canada.

11. Knoblock, C.A. 1994. Generating Parallel Execution Plans with a Partial-Order Planner. *Artificial Intelligence Planning Systems: Proceedings of the Second International Conference (AIPS94)*, Chicago, IL.

These publications, as well as additional information about SIMS, can be accessed through the WWW at `http://www.isi.edu/sims/`.

10.2 Loom

1. MacGregor, R. A Deductive Pattern Matcher. In *Proceedings of AAAI-88, The National Conference on Artificial Intelligence.* St. Paul, MN, August 1988.

2. MacGregor, R. The Evolving Technology of Classification-Based Knowledge Representation Systems. In John Sowa (ed.), *Principles of Semantic Networks: Explorations in the Representation of Knowledge.* Morgan Kaufmann. 1990.

Additional papers and information about Loom can be accessed trough the WWW at the Loom Project homepage: `http://www.isi.edu/isd/LOOM/LOOM-HOME.html` .

10.3 LIM/IDI

1. McKay, D.P., Finin T., and O'Hare, A. The Intelligent Database Interface: Integrating AI and Database Systems. In *AAAI-90: Proceedings of The Eighth National Conference on Artificial Intelligence.* 1990.

2. Pastor, J. A., McKay, D.P., and Finin T. View-Concepts: KnowledgeBased Access to Databases. In *Proceedings of the First International Conference on Information and Knowledge Management*, Baltimore, MD. 1992.

3. LIM User's Manual. Available from Paramax Systems Corporation by anonymous FTP at `ftp://louise.vfl.paramax.com/`.

10.4 KQML

1. Finin, T., Fritzson, R. and McKay, D. A Language and Protocol to Support Intelligent Agent Interoperability. In *Proceedings of the CE and CALS Washington '92 Conference*, June, 1992.

Additional papers and information about KQML can be accessed through the WWW at the KQML homepage: `http://www.cs.umbc.edu/kqml/` .

Acknowledgements

We would like to thank the developers of the software systems that we have used extensively in the construction of SIMS. In particular, thanks to Bob Macgregor and Tom Russ for the Loom knowledge representation system. Thanks to Don Mckay, Jon Pastor, and Robin McEntire at Paramax/Unisys/Loral for both the LIM relational database wrapper and their implementation of the KQML language. And thanks to Dan Weld and Tony Barrett at the University of Washington for the UCPOP planner, which we used to build the SIMS planner. In addition, thanks to Ping Luo for his testing of and feedback on an earlier version of this manual.

References

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

[2] Tim Finin, Rich Fritzson, and Don McKay. A language and protocol to support intelligent agent interoperability. In *Proceedings of the CE and CALS*, Washington, D.C., June 1992.

[3] Robert MacGregor. A deductive pattern matcher. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Saint Paul, Minnesota, 1988.

[4] Robert MacGregor. The evolving technology of classification-based knowledge representation systems. In John Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, 1990.

[5] Donald P. McKay, Timothy W. Finin, and Anthony O'Hare. The intelligent database interface: Integrating AI and database systems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990.