The KIWIS Knowledge Base Management System

M. Ahlsen [1] *A. D'Atri* [4] *P. Johannesson* [1] *E. Laenens* [5]
N. Leone [3] *P. Rullo* [3] *P. Rossi* [3] *F. Staes* [2]
L. Tarantino [4] *L. Van Beirendonck* [5] *F. van Cadsand* [2] *W. Van Santvliet* [5]
J. Vanslembrouck [6] *B. Verdonk* [6] *D. Vermeir (ed.)* [5]

[1]*SISU,* [2]*ORIGIN,* [3]*CRAI,* [4]*University of LAquila,* [5]*University of Antwerp UIA,* [6]*Alacatel-Bell*

Abstract

This report describes the functionality and architecture of the KIWI system, an advanced knowledge-base environment for large database systems, which is currently being developed within the framework of the ESPRIT programme (P2424) by a consortium of industrial and research organizations consisting of Alacatel-Bell Tel. (BE), CRAI (I), Enidata (I), FORTH (GR), ORIGIN (NL), SISU (SE), the universities of Antwerp UIA (BE, Coordinating Contractor), Calabria (I), and LAquila (I).

1. INTRODUCTION

This report presents an overview of the KIWI system's functionality and architecture.

Functionality

KIWI is a knowledge base management system which can be used both as a sophisticated stand-alone "personal knowledge machine" that supports knowledge-based applications, as well as a "window on the world" that provides a seamless integration of information coming from a wide variety of other sources with the local knowledge bases. The knowledge base management system's features include

1. An advanced graphical user interface design system that provides a smooth transition between default (generic) and special purpose (application) knowledge base usage.
2. A new knowledge representation and manipulation language, called LOCO[Lae89a] , that is based on a tight integration between the object-oriented and the logic programming paradigm. In addition, the language supports features such as defeasible and default reasoning, making it suitable also for AI-flavored applications, such as expert systems.

3. Efficient query evaluation algorithms that extend the state-of the art in deductive database technology.
4. An efficient underlying main-memory resident database management system that is tuned to the requirements of the efficient manipulation of large numbers of complex objects and deductive queries.

Integration with external information sources is possible in two ways:

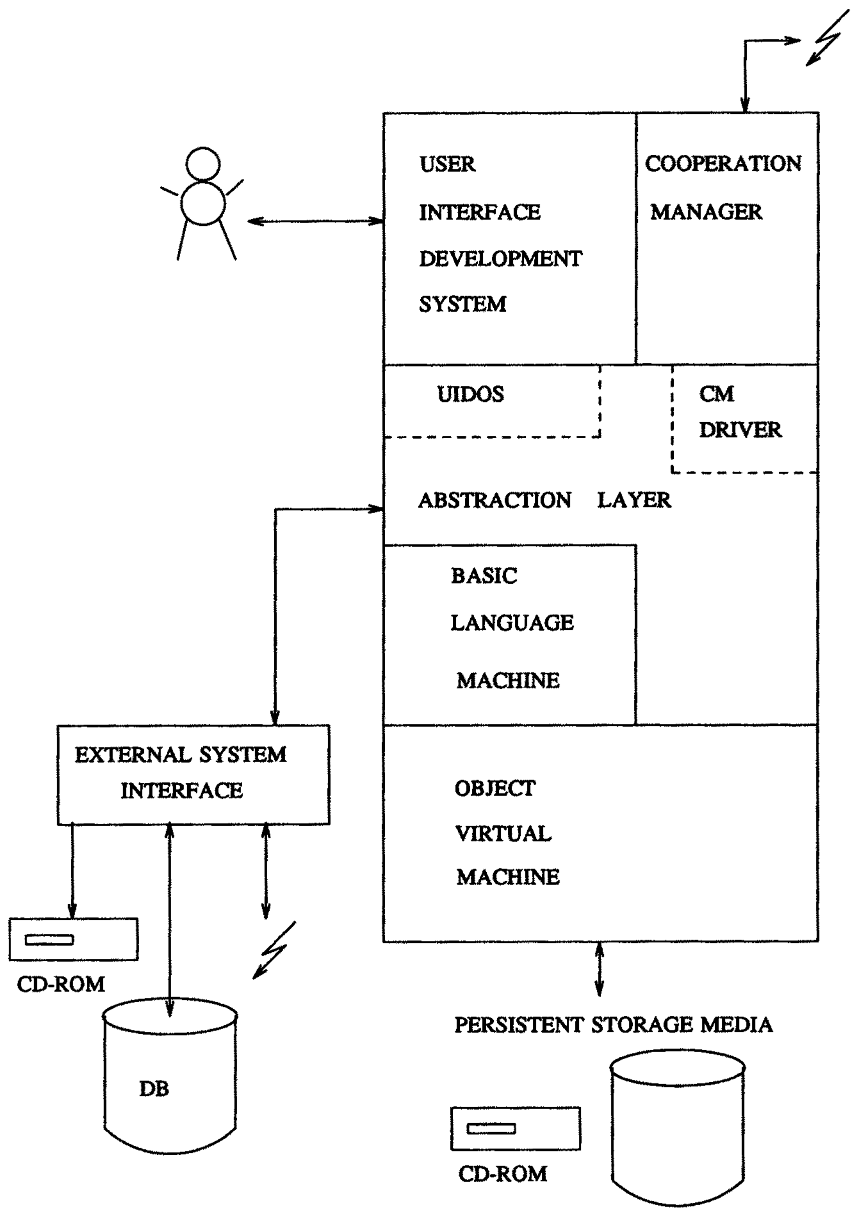
1. The first possibility provides a tight and transparent read-only interface to a variety of external information sources. These include traditional databases (relational, network or hierarchical), text or picture databases and applications. These databases may reside on the same or remote systems, using traditional or "new" devices (such as CD-ROM). The interface module of the KIWI system is extendible so that the above list is not exhaustive.
2. The second possibility provides for a federation of autonomous KIWI nodes, in which KIWI systems unite to form a network for information sharing and cooperation without a commitment to a centrally maintained global schema.

It should be stressed that in both cases, the imported information is, in the user's or LOCO programmer's view, completely integrated with the local knowledge base. This gives the opportunity for the user to enrich "dumb" data from outside sources with local higher level knowledge.

Architecture

The KIWI system, as illustrated in Figure 1, contains the following modules:

- The Abstraction Layer (AL) implements a logic formalism supporting complex objects, inheritance and true negation. It is responsible for the intelligent manipulation of (sets of) objects, interpreting rules, performing updates etc., using the OVM basic methods.
- The Basic Query Machine (BQM) provides optimization for queries over large sets of objects. In contrast to the AL, which uses a top-down evaluation method, BQM supports more efficient bottom-up evaluation.
- The Object Virtual Machine (OVM) manages objects in both main and secondary memory, including efficient access methods.
- The External System Interface (ESI) module provides read-only access facilities to a variety of external information sources.
- The User Interface Development System (UIDS) supports a toolbox of user interface construction tools, which are used by the underlying layers in order to implement an environment that is consistent over applications.
- The Cooperation Manager (CM) module gives the KIWI system the possibility to be part of a larger federated system.



2. KNOWLEDGE REPRESENTATION

In this section, we present a brief and informal overview of LOCO, the native language of the KIWI system.

Both object-oriented programming and logic programming have received increased attention over the last decade, due in part to their applicability in the area of database management systems. Although the term "object-oriented" is loosely defined, a number of concepts have been identified as the most salient features of that approach. According to[Atk89a] some of these concepts are: complex objects, object identity, inheritance, defaults. LOCO is an object-oriented database programming language which models all of these concepts and integrates them with the logic programming paradigm. This greatly enhances the capabilities of the language. Indeed, on one hand, including the object-oriented concepts yields superior modeling capabilities while on the other hand, the deductive approach allows for a clear formal semantics as well as declarative query capabilities, as enjoyed by relational systems. A number of other proposals which combine the object-oriented and logic programming paradigms, can be found in the literature. We refer among others to[Kif89a, Che89a, Abi89a] and references therein. The approach taken in LOCO extends these proposals however, in that we introduce nonmonotonic and default reasoning[McD80a, Nut88a] in LOCO rather than monotonic reasoning. In many cases the latter has indeed proven to be too restrictive for AI-flavored applications. It is essential to note that the nonmonotonic inheritance is built into the semantics of the language, and is not algorithmically defined, as is the case in [Dal89a].

A LOCO program describes a knowledge base (schema and initial population) as a partially ordered set of interrelated objects. The properties of an object (i.e. its relationships to other objects) are described using an extended logic program [Lae90a, Lae90b], i.e. a logic program where negation may also occur in the rule heads. Hence, to LOCO, an object is just a logical theory. However, the sentences in an object's definition do not constitute the entire knowledge about that object. A specificity relation defined on the objects allows for the introduction of some rules for knowledge flow between them. This specificity relation (also called instance-of relation) is sufficiently general and powerful to be useful to model e.g. delegation [Lie86a, Ste87a], classification and/or generalization hierarchies, etc. Therefore, the language does not enforce a particular modeling paradigm.

2.1. Objects as theories

Our starting point is the notion of object. In object-oriented languages, an object has, besides an identity that remains fixed throughout its lifetime, properties that may be either passive (such properties are often called "instance variables") or active ("methods"). Methods may rely, using parameters, on other properties of the same or different objects, in order to compute a result.

In LOCO, we use logic to describe the properties of an object, i.e. we identify an object with a logical theory. In particular, an object is represented by a set of rules of the form

$$H{:-}B_1, B_2, \ldots, B_n \qquad (n \geq 0)$$

where the head H of the rule is a literal, i.e. a positive or negative atom, and each B_i, $0 \leq i \leq n$ in the body of the rule is either a literal or an extended literal, i.e. an expression of the form $X.p$ where X is the name of an object (or a variable) and p is a literal. Intuitively, an extended literal $X.p$ should be read as "p is true at object X" or "X has property p", while a literal p refers to the truth of p at the "current" object (see below), in other words p and $Self.p$ are equivalent.

The only differences between a rule and a clause as used in a "traditonal" logic program are that the head H of a rule can be a negative atom and the object reference in an extended literal:

As an example, consider the following LOCO fragment describing two objects: fred and sally.

```
/* Example 1 */
fred = { name("Fred").
         mother(sally).
         parent(X) :- mother(X).
         parent(X) :- father(X).
         ancestor(X) :- parent(X).
         ancestor(X) :- parent(Y),Y.ancestor(X).
         child(X) :- X.parent(Self). };
sally = { name("Sally"). };
```

The first rule *name("Fred")* in *fred* is a fact, i.e. a rule with an empty body. The second rule is a fact whose argument refers to the other object *sally*. Informally, facts in objects may be regarded as the equivalent of "instance variables" in traditional object-oriented languages.

The next four rules define the derived properties *parent* and *ancestor* of *fred*. The rules for *parent* are as in ordinary logic programs. The second (recursive) rule for *ancestor* illustrates the possibility for a literal in the rule body to refer to a property of another object: fred has an ancestor *a* if he has a parent *Y* where *ancestor(a)* holds. Rules such as those defining the *parent* and *ancestor* properties play a role similar to that of "methods" in imperative object-oriented languages. This analogy will be confirmed by the procedural semantics described later.

A complete LOCO program then consists of a partially ordered set of object descriptions.

2.2. Negation

LOCO supports true negation rather than negation by failure, as is illustrated by the following example:

```
/* Example 2 */
situation = { safe :- ¬dangerous.
              ¬dangerous :- p.
              dangerous :- q.
              ... };
```

Informally, *situation* will be deemed safe if it is not dangerous. The latter will be true only if there are explicit reasons (e.g. *p*) to accept this. In other words, failure to prove *dangerous* will not be sufficient reason to conclude *¬dangerous* and hence *safe*.

Allowing negative literals in the head of a rule raises the possibility for contradictory evidence to appear. E.g. in the previous example, if both *p* and *q* are true, we have rules suggesting *dangerous* and ¬*dangerous* at the same time. In this case, LOCO will take a sceptical[Tou84a] approach and neither *dangerous* nor its negation will be true. Note that this implies that our logic is really three-valued since literals may be either true, false or unknown, where the latter value represents both the case where there is no information (undefined) and the case where there is contradictory information.

2.3. Multiple inheritance

An important feature of the object-oriented programming paradigm is the ability to structure objects in hierarchies such that properties of lower level objects ("instances" or "subclasses") may be derived using rules at the higher level objects (usually called "classes"). In LOCO, such inheritance will be obtained by allowing the objects that constitute a program to be structured in a "specificity" partial order‡, denoted " ≤ ". For example, we could rewrite Example 1 as:

```
/* Example 3*/
person = { parent(X) :- father(X).
           parent(X) :- mother(X).
           ancestor(X) :- parent(X).
           ancestor(X) :- parent(Y), Y.ancestor(X).
           child(X) :- X.parent(Self).  };
(person) fred = { name("Fred").
                  mother(sally).  };
(person) sally = { name("Sally").  };
```

The construction $(o_1 \ldots o_n)\ o$ indicates that the object o is more specific than each of the objects o_i, $1 \le i \le n$, i.e. $o \le o_i$. We say that an object a is an **instance** of an object b if $a \le b$.

Informally, the rules defined at an object do not constitute the entire knowledge about that object; objects can also use the rules defined at more general (less specific) objects. E.g., *sally*, being an instance of *person*, will have the property *child(fred)*, by using the rule

```
child(X) :- X.parent(Self).
```

and by substituting *fred* for *X* and *sally* for *Self*. (In analogy with many other object-oriented languages, the keyword *Self* in LOCO always refers to the object where the rule is "used", in this case *sally*).

‡ Note that LOCO does not distinguish between the instance-of (a class) and subclass-of relationships. This means that "own" (i.e. non-inheritable) properties of classes are not part of the core language.

It should be stressed that objects "inherit" rules, not conclusions, from higher objects. This corresponds to dynamic binding in traditional object-oriented languages.

The specificity order can be used to define default properties, as is illustrated in the following example:

```
/* Example 4 */
bird = { fly. };
(bird) tweety;
(bird) penguin = { ¬fly. };
(penguin) joe;
```

Since *tweety* is an instance of *bird*, it inherits the *fly* property of the latter.

If several rules conflict, the rule which is defined at the more specific object "wins". Thus, *penguin.¬fly* will hold as will *joe.¬fly*. We say that, for *penguin* and *joe*, the "*fly*" rule at *bird* is **overruled** by the "¬*fly*" rule at the more specific object *penguin*. If conflicting rules are defined at incomparable objects, the sceptical approach is taken and both rules are said to be **defeated.** This is illustrated below using a familiar example from nonmonotonic logic.

```
/* Example 5 */
quaker = { pacifist. };
republican = { ¬pacifist. };
(quaker republican) nixon;
```

Here neither *nixon.pacifist* nor *nixon.¬pacifist* will be true. It is interesting to note the connection between default properties and negation by failure. Indeed, in order to set up a particular predicate, e.g. p, such that it will be false whenever we fail to prove its truth, it suffices to have a "default" rule $\neg p$ at a "top" object (of which all other objects are instances). This will ensure that, whenever no rules for establishing p are applicable at an object o, $o.\neg p$ holds because o inherits the default rule $\neg p$ at $top \geq o$.

The inheritance mechanism of LOCO can be put to good use in order to support database updates and version control. Indeed, asserting a new rule or fact r at an object o is interpreted as the creation of a new instance (**version**) o' of o rather than as a modification of the set of rules describing o. This instance has only the rule r in its object definition. The semantics of LOCO's inheritance mechanism ensure that each update is correctly interpreted: the asserted information is taken to overrule any previously available information. This is true for the addition of rules as well as facts.

As an example, consider again the program of Example 7. Executing

```
!fred.sick
```

where '!' denotes the assert operator, will result in a new program equivalent to:

```
(bird) fred_000;  /* The old version of fred */
(fred_000) fred = /* the new version of fred */
          {
          sick.
          };
```

Note that the present version of LOCO has no explicit retract. In order to "remove" the fact that *fred* is *sick*, it suffices to execute

```
!fred.¬sick
```

resulting in

```
(bird)fred_000;
(fred_000) fred_001 = { sick. };
(fred_001) fred = { ¬sick. };
```

Note that the "removal" of *sick* implies the fact *¬sick* at *fred*. If we want *fred* to be neither *sick* nor *¬sick* it suffices to execute

```
!(fred.¬sick, fred.sick)
```

When updating a "class" (an object that has instances), care should be taken to "detach" all its "direct" instances and attach them to the new version instead. In a future version of LOCO, we intend to add facilities which allow the exploitation of the other advantages of this approach to update: retrieval with respect to some previous version, explicit version control in general, parallel versions etc.

2.4. Types and constraints

LOCO accepts type constraints in objects, as in

```
person = { name<string>.
        parent<person>.
        like<food|person>.  };
```

The above fragment requires that *p.name* (s) implies $s \leq string$ (*string* is a built-in "class") for all $p \leq person$. Similarly, a *person*'s *parent* should be another *person* and a person is only allowed to like foodstuffs or other persons. Type checking is static, using a simple type checking algorithm.

2.5. Query evaluation

Whenever possible, LOCO queries are processed by the BQM. The latter is based on the declarative semantics which have been described e.g. in [Lae90b, Lae90c, Lae91a, Lae90a, Rul90a]. In particular, the KIWI system supports the well-founded semantics, which are based on a suitable extension of the notion of unfounded set given for classical logic programs. It is not surprising that, just as for classical logic programming, the computation of the well-founded model is quite demanding. To this regard, some work has been done in[Rul90a, Leo90a] considering a suitable adaptation of the iterated fixpoint algorithm[Prz89a] given for classical logic programs. It is possible, however, to specialize the general method and make it very efficient for a meaningful class of programs, i.e. for stratified programs [Rul90b].

Typically, a stratified program shows a two level structure: the definition of a property belongs to a certain object, while the exceptions (i.e. conflicting rules) belong only to its subobjects. In a sense, the notion of stratification can be intended as a way to detect when a program has a simple structure w.r.t. the treatment of contradiction so that its well-founded model can be computed in a monotonic (non-alternating) fashion. We remark, finally, that the class of stratified programs encompasses a very large class of applications. As a result, the BQM wil provide very good performance in most "real life" cases.

3. USER INTERACTION AND INTERFACE MODELS

3.1. Introduction

Different classes of users may have different perceptions of the reality the KB represents, and of the way in which such reality is represented by means of the knowledge representation model. Furthermore, in the course of using the system, the user's perception may change as a result of different influences (e.g., training, exposure), and hence both needs and requirements change over time. It is clear from these considerations that a fixed user interface (UI) is not a good solution, since it would be unable to address the needs and the requirements of all users and deal with user transitions[Lae89b]

Besides the different classes of users, the system also has to support different applications. This implies that it has to be possible to create custom interfaces for the more advanced applications, or to adjust the default interface to them.

3.2. Architecture

To satisfy these flexibility requirements, we propose an architecture which allows to take into account both the requirements from the users and the necessity of customizing the UI towards the contents of the information base. Examples of this kind of customization can vary from simply hiding non-interesting properties for the naive-users to completely changing the appearance of the UI according to the semantics of the application.

The User Interface consists of three layers: the User Interface Description Objctes, the Display Model and the Interaction Paradigms. The User Interface Description Objects (UIDO's) are used to describe customizations of existing User Interfaces (or even complete user interfaces) in the Loco language. The Display Model serves to associate with each object in the knowledge base an appropriate representation: a description of how the object has to be represented on the screen. The Interaction Paradigms are a collection of different ways in which one can interact with the knowledge base. The interaction is performed by direct manipulation of the graphical entities as defined by the Display Model.

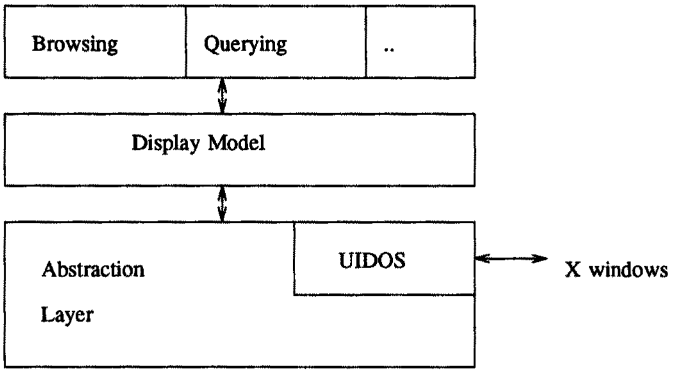


Figure 2: The User Interface Architecture

3.3. The User Interface Description Objects

In order to facilitate the definition of customized User Interfaces, the LOCO language supports a set of builtin (graphical) objects. In other words, these User Interface Definition Objects (UIDO's in short) are fully integrated in the native KR formalism of the system, so that the knowledge engineer can define and maintain an application environment - from KR over connecting external sources[The89a] to UI definition - using a single uniform language.

The User Interface Description Object are modelled using a hierarchy of object classes as shown in figure 3.

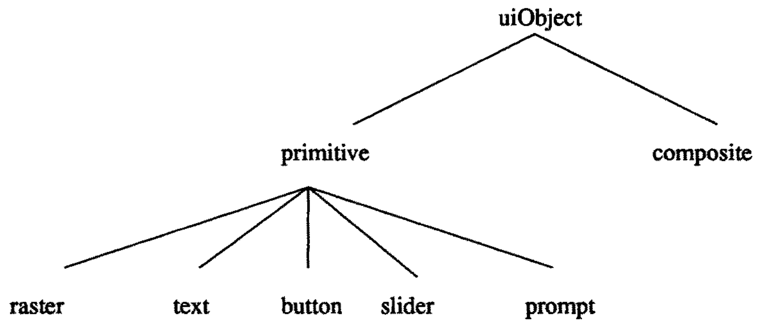


Figure 3: The hierarchy of User Interface Description Objects

The nodes in this tree are implemented using LOCO objects while the links are realized by the LOCO specificity relation. As LOCO features inheritance and defaults, we put properties common to all instances of a certain object class in the class itself in order to avoid redundant definitions. Some of these properties may be redefined at the instance object, thus overruling the default values. We distinguish between primitive and composite UIDO's. The class of primitive UIDO's is further divided in subclasses that represent a primitive display or interaction paradigm such as buttons, sliders, texts, etc. The composite UIDO's serve as a kind of containers. They are used to group a collection of primitive and/or composite UIDO's while enforcing layout constraints on them. A more precise description of these object classes can be found in [Sta90a].

3.4. The display model

The display model associates a standard visualization with each LOCO object, to provide a natural and easy-to-understand graphical representation (suitable for a direct manipulation-based interaction), which takes into account the "structural" aspects of the objects. Of course, a standard model cannot take care of the specific semantics of objects and of the needs of a particular user. However, the default model can be easily enriched, e.g. by customizing the presentation of those objects that are more relevant to the user and/or to the application. The model can be customized in two ways[Sta90a]

- By tailoring the default display model. This can be done both in an interactive and a programming way and is intended to help both the naive user and the knowledge engineer in making adjustments to the default representation. Examples of these adjustments are the ordering of the properties, hiding some of the properties etc.
- By defining a customized representation. This is achieved by describing a representation using a collection of built-in LOCO classes.

3.5. Interaction Paradigms

Supporting different classes of users does not only mean giving the possibility of tailoring the graphical appearance of object representations, but it also requires the interaction environment to provide several interaction paradigms with different usage complexity. In order to achieve a powerful and friendly interface which assists the users in all the phases of their interaction, the different paradigms must be uniformly included in a graphical dialogue environment based on the direct manipulation of the visible objects. Furthermore, working in such an integrated environment, users learn to describe their goals more and more precisely.

To satisfy these user's demands, KIWI offers a number of interaction paradigms from browsing to querying thus covering in a continuum the gap from very naive users to sophisticated ones. The interaction is always carried on in a uniform way, based on the direct manipulation of the graphical objects previously described. We briefly discuss here some of these paradigms while a more detailed description can be found in [Sta90b].

The simplest interaction with a knowledge base is an elementary navigation based on the browsing paradigm (oriented to users with a poor knowledge of the KB and not able to use querying formalisms): at any time the user examines a current object and its neighborhood, while the browsing procedure proceeds by iteratively selecting one of the objects in such a neighborhood to become the current one.

Expert users may want to express their goals in a more direct way, even if they are not able to formulate standard queries in LOCO. To support these users, some querying paradigms are also provided, with different complexity and retrieval power, e.g., based on the synchronized browsing (to speed up repetitive searches) and on a by-example approach (which extends the well known QBE defined for the relational data model to an object-oriented data model); the query formulation is based on the manipulation of the graphical objects, and, under the system assistance, it is possible to graphically adjust previously formulated queries when their answers are not satisfactory.

4. COOPERATION

In addition to the local knowledge base, there are two other sources of information available to users of a KIWIS system. One alternative is to define interfaces to existing applications, such as conventional database systems. Another possibility is to connect several KIWIS knowledge bases in a loosely coupled network to form a Federation.

4.1. The Federated Architecture

The KIWIS system supports advanced techniques for knowledge representation in combination with database management. To provide for information sharing, several KIWI systems (or nodes) are connected to form a loosely coupled network of knowledge/database management systems. *The*

Cooperation Manager (CM) is the component in the prototype system that provides the basic cooperation and communication facilities between otherwise independent KIWI systems. The KIWIS system is in this sense a Federated Information System based on the concept of federated databases[Hei85a, Hei87a] and the principles for open systems[Hew84a].

The federated architecture has the following characteristics which form the underlying design principles for the CM:

- there is no central authority for global control
- an absence of a strict global schema
- nodes have possibly only partial knowledge of the surrounding system
- sharing is at the discretion of the individual nodes
- the topology of the federation may change

Autonomy is the central to this architecture and is the fundamental property of the member nodes. We distringuish three forms of this autonomy[Ahl90a] for the nodes in a KIWIS federation. The *semantical autonomy* means that the schema of a node does not have to conform to the schema of any other node, hence there is no (strict) global schema in the federation. *Behavioral autonomy* implies that any node exercises full control over its own resources, and is thus not obliged to process requests from other nodes. Finally, there is a *network autonomy* which says that there is no central authority to organize the interactions between nodes; hence a node is not necessarily dependent on intermediary nodes for communication with other members of the federation.

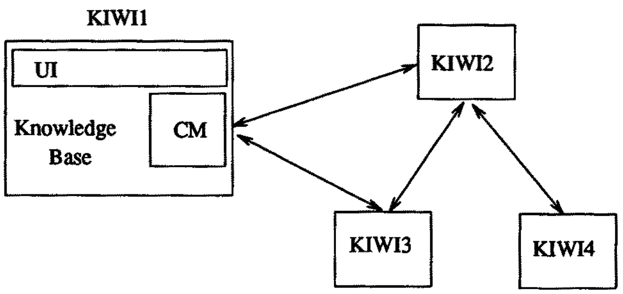


Figure 1: The KIWIS Federation

The KIWIS federation is homogeneous wrt data model type, i.e., each node uses LOCO for describing the local schema. Thus, the CM does not perform any model type translations, a KIWI node could ofcourse use other languages with local mappings to LOCO. The CM will in part run as a LOCO application in a KIWIS system (similar to parts of the UI). The nodes in the federation form a logical network, and they can thus reside on the same or different hosts.

4.2. Contracts

Cooperation between nodes in a KIWIS Federation is based on the importation and exportation of knowledge base objects. The behavioural autonomy means that the behaviour of nodes may be unpredictable wrt other nodes. Therefore, sharing of information can only take place if there exists an explicit bilateral aggreement between two nodes. Such an agreement is represented by a *contract*, which controls when and how nodes may exchange knowledge base objects.Contracts are established thru bilateral dialogues between nodes[Joh88a] where the contract terms may be subject to negotiation[Smi80a, Alo89a] between the nodes. There is a Contracts Establishment Protocol which coordinates the dialogue between the potential importer, and the potential exporter. The terms of a contract represent the rules for import/export of objects. In general, these terms are specified by events and time-intervals. One obvious term is the duration of the contract, defining the time during which the objects concearned, the domain objects, are available to the importer. Another term defines the type of object access used between two nodes. An importing node may either operate directly on the objects in the exporter's knowledge base (corresponding to remote querying), or, copies of exportable objects may be transferred to the importing node(corresponding to snapshots or quasi copies). Remote updates are not supported, for autonomy reasons.

4.3. The Cooperation Schema

The information needed by the CM is described by a conceptual schema, called *the Cooperation Schema*, identical for each node. This schema is stored and maintained in the local knowledge base of each node and may be operated upon to provide information on existing and prior contracts establishments. The cooperation schema is represented in LOCO. Based on this schema, each node has an export interface, *the Export Schema*, which serves as view on the local knowledge base made available to other members of the federation and forms the basis for establishment of new contracts. Further, the semantical autonomy and the network autonomy imply a responsibility for each node to aquire and maintain knowledge about the federation. To this end, each node has a local catalogue, *the Federal Map*, which stores information on known member nodes of the federation, such as their adressess and their Export Schemata. In addition, the Cooperation Schema is used to store contracts and imported objects. Figure 2 shows some of the object types used to represent this knowledge.

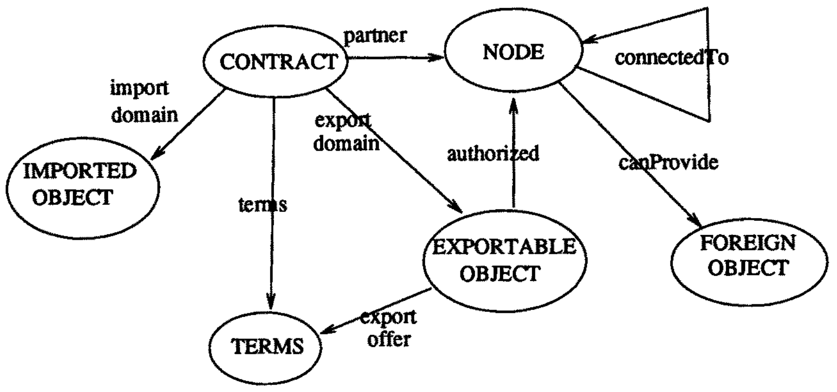


Figure 2: Part of The Cooperation Schema

In order to aquire knowledge about the federation, a node may request to import the Export Schemata of other nodes, as well as their Federal Maps. Based on the object descriptions given in the Export Schemata thus imported, a node may then request the establishment of contracts with the corresponding nodes. In a similar way, the importation of Federal Maps may provide new knowledge about unaquainted nodes in the federation. It should be noted that a node selectively decides how much of this information it is willing to export.

Once a contract has been established, the domain objects may be integrated with the local KB. This will allow for possibly transparent querying of imported information.

5. CASE STUDIES

The viability of the KIWI system will be evaluated in a number of case studies, the most important of which is in the area of telecommunication, in particular the modelling and management of intelligent network services. Intelligent networks require an extension of the capabilities of traditional networks to allow for specialized, customer-tailored service provision and processing. The distributed nature of service implementation, along with the complex interactions among services, requires advanced, highly efficient knowledge-based techniques. We therefore believe that the case study will allow to thoroughly evaluate the different layers of the KIWIS system, and at the same time illustrate the applicability of KIWIS for novel real-life applications in telecommunication.

LOCO is already extensively being used to describe basic and supplementary services such as a basic (two-party) call, call transfer, call forwarding, call waiting, etc. New services are built as much as possible upon existing ones by inheriting the functionality of existing services, using the (multiple) inheritance mechanism of the language. For each service request, a number of rules is provided. This leads to a more modular approach and increased reusability than is possible with present day specification languages such as SDL or Estelle. The state of each device is itself described by a number of objects which are dynamically created and removed. The declarative update mechanism of the LOCO language is extremely important in this context.

In the future, the other layers of the KIWIS system will gradually be integrated in the case study. The user interface will be very useful for browsing and querying the available services. Because of the distributed nature of the subject, several KIWIS systems communicating via the Cooperation Manager can be used for simulating parts of the real world. Charging and routing information can be stored on non-KIWIS systems and be accessed via the Exteral System Interface.

Apart from simulation purposes, the possibility to integrate the KIWIS system (or a system based on KIWIS) in a future telecommunication system to replace the usual database part of these systems is considered.

6. CONCLUSION

A first prototype of the KIWI system, integrating (part of) the UIDS, the AL and the OVM, is currently available. According to the project schedule, this prototype will be gradually extended and enlarged, integrating more and (more powerful versions of) layers, cumulating in a first fully integrated system by 1992. A final version of the KIWI system should be available by June 1992.

Prototype versions of the KIWI systems are (and will be) available for the price of manuals and media, provided they are used for non-commercial purposes only. Further information can be obtained from any of the participating organizations.

References

Abi89a.

S. Abiteboul, "Towards a deductive object-oriented database language," in *Proceedings DOOD89, Kyoto*, pp. 419-438, 1989.

Ahl90a.

M. Ahlsen and P. Johannesson, *Contracts in Database Federations*, SISU - Swedish Institute for Systems Development, Stockholm, Oct 1990. Presented at Conference on Cooperating Knowledge Based Systems, Keele, Oct 1990.

Alo89a.

R. Alonso and D. Barbara, "Negotiating Data Access in Federated Database Systems," *Proc. Fifth International Conference on Data Engineering*, pp. 56-65, Los Angeles, Feb. 1989.

Atk89a.

M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik, "The object-oriented database system manifesto," in *Proceedings DOOD89, Kyoto*, pp. 40-57, 1989.

Che89a.

W. Chen and D. S. Warren, "C-Logic of Complex Objects," in *Proc. of the Eight Symposium on Principles of Database Systems*, pp. 369-378, 1989.

Dal89a.

M. Dalal and D. Gangopadhyay, "OOLP: A translation approach to object-oriented logic programming," in *Proceedings DOOD89, Kyoto*, pp. 555-568, 1989.

Hei85a.

D. Heimbigner and D. McLeod, "A Federated Architecture for Information Management," *ACM Transactions on Office Information Systems*, vol. 3, no. 3, pp. 253-278, 1985.

Hei87a.

D. Heimbigner, "A federated system for software management," *IEEE Database Engineering Bulletin*, vol. 10, no. 3, 1987.

Hew84a.

C. Hewitt and P. deJong, "Open Systems," in *On Conceptual Modelling*, ed. J. Schmidt, pp. 147-164, Springer Verlag, 1984.

Joh88a.

P. Johannesson and B. Wangler, *The Negotiation Mechanism in a Decentralized Autonomous Cooperating Information Systems Architecture*, 1988. Tech. Report Nr. 62, SYSLAB, University of Stockholm, S-10691 Stocholm, Sweden

Kif89a.

M. Kifer and G. Lausen, "F-Logic: A Higher-Order Language for Reasoning About Objects, Inheritance and Scheme," in *Proc. of SIGMOD*, pp. 134-146, 1989.

Lae89a.

E. Laenens, D. Vermeir, and B. Verdonk, "LOCO, a logic-based language for complex objects," in *Proceedings of the Esprit conference*, pp. 604-616, 1989.

Lae89b.

E. Laenens, F. Staes, and D. Vermeir, "A Customizable Window-Interface to Object-Oriented Databases," in *Proc. of the ECOOP conference*, pp. 367-382, 1989.

Lae90b.

E. Laenens and D. Vermeir, "A Fixpoint Semantics of Ordered Logic," *Journal of Logic and Computation*, vol. 1, no. 2, pp. 159-185, 1990.

Lae90c.

E. Laenens, B. Verdonk, D. Vermeir, and D. Sacca, "The LOCO language: towards an integration of logic and object oriented progra mming," *Proceedings of the Workshop on Logic Programming and Non-Monotonic Reasoning*, Austin, Texas, 1990.

Lae90a.

E. Laenens, D. Sacca, and D. Vermeir, "Extending logic programming," in *Proceedings of the SIGMOD conference*, pp. 184-193, 1990.

Lae91a.

E. Laenens and D. Vermeir, *On the Relationship between Well-Founded and Stable Partial Models*, 1991. Proc. of the Mathematical Fundamentals of Database and Knowledge Base Systems,to appear

Leo90a.

N. Leone, A. Mecchia, M. Romeo, G. Rossi, and P. Rullo, "From DATALOG to Ordered Logic Programming," in *Proc. of the 13th Int. Seminar on DBMS, Romania*, 1990.

Lie86a.

H. Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," *OOPSLA '86*, pp. 214-223, 1986.

McD80a.

D. McDermott and J. Doyle, "Non-monotonic logic I," *Artificial Intelligence*, vol. 13, pp. 41-72, 1980.

Nut88a.

D. Nute, "Defeasible reasoning and decision support systems," *Decision support systems*, vol. 4, pp. 97-110, 1988.

Prz89a.

T. C. Przymusinski, "Every logic program has a natural stratification and an iterated fixed point model," in *Proc. of the Symposium on Principles Of Database Systems*, pp. 11-21, 1989.

Rul90a.

P. Rullo, P. Rossi, and D. Sacca, *Revised Specification of BQM*, 1990. The KIWIS project, Report D2

Rul90b.

P. Rullo, P. Rossi, and D. Sacca, *BQM: strategies and basic architecture*, 1990. The KIWIS project, Report BLM4

Smi80a.

R.G. Smith, "The Contract Net Protocol: High-Level Communication and Control in a Distrib," *Readings in Distributed Artificial Intelligence*, pp. 357-366, 1980.

Sta90a.

F. Staes, E. Laenens, L. Tarantino, and D. Vermeir, "A seamless integration of graphics and dialogues within a logic based object-oriented language," *Journal of Visual Languages and Computing*, p. to appear, 1990.

Sta90b.

F. Staes and L. Tarantino, *Report G2: Revised specifications of the User Interface*, 1990. Esprit P2424 (KIWIS) Report G2

Ste87a.

L. Stein, "Delegation is Inheritance," *OOPSLA '87*, pp. 138-146, 1987.

The89a.

The KIWIs Team, "The KIWI(s) projects: past and future," in *Proc. of 6th Esprit Conference*, pp. 594-603, 1989.

Tou84a.

D.S. Touretzky, "Implicit ordering of defaults in inheritance systems," in *Proceedings 5th National Conference on Artificial Intelligence (AAAI)*, pp. 322-325, Austin, TX, 1984. Also in 'Readings in nonmonotonic reasoning', M.L. Ginsberg