PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

MANDALA: A LOGIC BASED KNOWLEDGE PROGRAMMING SYSTEM

Koichi Furukawa, Akikazu Takeuchi, Susumu Kunifuji, Hideki Yasukawa, Masaru Ohki, Kazunori Ueda†

ICOT Research Center
Institute for New Generation Computer Technology
Tokyo, Japan

† C&C Systems Research Laboratories
NEC Corporation
Kawasaki, Japan

ABSTRACT

Mandala is a programming system aimed for developing knowledge information processing systems in logic programming framework. It is not only a knowledge programming system but also a basis for a knowledge base management system. The nature of this duality comes directly from two-facedness of logic programming, that is, both procedural and declarative interpretation. Instead of conventional sequential execution environment, Mandala provides users with parallel execution environment, that is, Mandala allows users to describe multiple-processes. On this parallel execution environment, cooperative problem solving systems can be constructed, in which more than one problem solvers can be active in parallel. Moreover, Mandala realizes object oriented programming using a process mechanism in KL1, thus achieving to manipulate dynamically changing states. Objects in Mandala can be seen as problem solvers which have inference engines and knowledge base. It is possible for users to define a specific inference engine of some objects. Some experimental studies have been done to exemplify its expressive power and flexibility.

1 INTRODUCTION

Knowledge representation is a major challenge for research in artificial intelligence. Human beings possess a wide variety of knowledge that, so far, no language or system that has been developed is capable of adequately representing its entire range. As is the case of information processing research in general, there is a constructive aspect also in artificial intelligence research: it repeats the process of developing and testing a language or system. In that sense, knowledge programming languages and systems have been rapidly gaining attention as tools for developing knowledge representation systems. Examples include LOOPS (Bobrow and Stefik 1983), KEE (Kehler and Clemenson 1983), STROBE (Smith 1984) and GLISP (Novak 1983).

Since complex relations between objects play a basic role in knowledge, it is mandatory that a knowledge representation language have the capability to express and manipulate these relations. This basically requires list processing, a function for which LISP has long been used. All languages listed above use LISP as their base language, thereby providing environments which allow object-, data-, and rule-oriented programming, beside other sophisticated techniques. However, these systems need distinct mechanisms to realize such advanced programming functions as mentioned above and therefore the language structures as well as system structures are quite complex. It might be worth to name them as PL/1 in knowledge programming languages, which means that they are not quite well refined languages/systems.

In addition to those programming functions, the functions required for knowledge representation are knowledge base management capabilities, including consistency checking, and inference using incomplete knowledge. Among the systems aimed at realizing these functions are MRS (Genesereth *et al.* 1980) and KRYPTON (Brachman *et al.* 1983). These are both based on the use of predicate logic. This may be said to indicate that predicate logic is specifically suited to such problems.

We have been designing a knowledge programming language/system Mandala based on logic programming. Its major characteristic is that it realizes both advanced programming feature and knowledge base management feature in a single framework, which is a direct reflection of the capability of double readings of Horn clauses: procedural readings and declarative readings. That is, we may say that Mandala is more like a knowledge programming system in terms of procedural readings, while it is more like a knowledge base management system in terms of declarative readings.

Furthermore, main functions in these two features correspond to principal concepts in logic programming; namely, object oriented programming corresponds to stream programming in Concurrent Prolog (hereafter we call it CP) (Shapiro 1983(a)), rule oriented programming to clause-wise programming in Pure Prolog (hereafter we call it PP) and data oriented programming to data flow synchronization mechanism in CP. Concerning on knowledge base management feature, the entire mechanism itself corresponds to meta programming feature in logic programming languages by regarding knowledge described by a set of Horn clauses to be object level programs. Also, the inconsistency check can be done by using PP interpreter which performs Horn clauses deduction (as a theorem prover). There are further possibilities to realize inductive inference as well as non monotonic reasoning in a logic programming framework

(Miyachi *et al.* 1984(a)), (Kitakami *et al.* 1984).

Another aim of Mandala is to incorporate parallelism both in problem description and in execution. The ultimate goal of the Fifth Generation Computer Systems Project is to develop a highly parallel computer for knowledge information processing. To achieve the goal, we need a proper tool for extracting a large amount of parallelism from application problems related to knowledge information processing. Since Mandala supports object oriented programming and also is written in KL1 (Furukawa *et al.* 1984) (which is a logic programming language with stream-AND-parallelism like CP and will run on a Parallel Inference Machine (PIM), it should be one of the best suited languages for that purpose.

In this paper, Chapter 2 explains the basic components of Mandala. Chapter 3 discusses knowledge programming aspects in Mandala. Chapter 4 describes knowledge management issues. And the final chapter provides a summary of this paper and future work.

2 BASIC COMPONENTS AND LINKS OF MANDALA

There are two basic components of Mandala: unit worlds and instances. Each instance is associated with one unit world. Logically a unit world represents a set of axioms and an instance represents a query handler for the associated unit world. An instance receives theorems from other instances and tries to prove them from a set of axioms stored in the associated unit world. In Mandala, it is generally possible for a user to define a proof procedure for an instance associated with some unit world. Query handlers can be regarded as receiving queries and trying to answer them from the knowledge in the associated unit world. In a knowledge representation system, unit worlds represent static entities such as declarative knowledge and they may be used as components for creating a more complex world. As contrasted with unit worlds, instances represent dynamic entities such as actor-like objects, which can hold local states and can receive, send and process messages. In Mandala, instances can take actions in parallel. Furthermore, they are used to implement knowledge base managers in knowledge representation systems.

In the current implementation, unit worlds are composed of sets of PP and/or CP clauses, and instances are realized by CP processes, where the term *process* means a chain of goal reductions initiated by a given goal (Takeuchi and Furukawa 1983). Unit worlds and instances are illustrated by cylinders and circles respectively (Fig. 1).

Fig.1 Graphic representation of basic constructs

The four basic links used in Mandala, *instance_of*, *is_a*, *part_of* and *manager_of*, are defined and their logical meanings are described as follows:

2.1 Instance_of link

An *instance_of* link relates a unit world to an instance. If an instance J is connected to a unit world U by an *instance_of* link, J is called an instance of U, which is indicated by an undulating line, as illustrated in Fig. 2. As decribed above, a unit world and an instance connected by this basic link are logically seen as a set of axioms and a prover (query handler) for the theory formed by the set of axioms. If an instance receives a query, it tries to solve it based on a set of axioms stored in the associated unit world. Based on the procedural interpretation of logic program, this can be seen that a set of clauses which specifies the behavior and the property of an instance is defined in the unit world as a program and the instance executes the program for some input. However, the point here is that several instances can be connected to the same unit world and they can execute the program for their own input queries in parallel. This gives a basis for object oriented programming in Mandala, which is further described in the chapter 3.

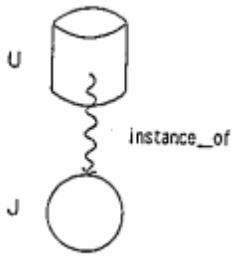

Fig.2 *instance_of* link

2.2 Is_a link

An *is_a* link, established between unit worlds, represents a conceptual hierarchical relation between them. For example, if unit world U2 is a specialization of unit world U1, U2 *is_a* U1, and U2 is connected to U1 by solid line, as shown in Fig. 3. Logically a set of unit worlds connected linearly by *is_a* links forms a theory. In this sense, if a unit world is connected to the other unit worlds by *is_a* links, the unit world represents only a fragment of the theories.

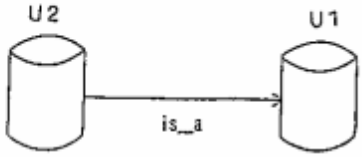

Fig.3 *is_a* link

So-called property inheritance between conceptual entities is automated in this framework (Goldberg and Robson 1983). Unlike other object-oriented systems, Mandala implements property inheritance by creating a

world for message processing. That is, if an instance fails to process a message (query) by any axioms stored in the unit world connected to the instance by *instance_of* link, Mandala gets a higher-level unit world connected to the unit world by an *is_a* link, combines the two into a new world and tries to process the message there, and repeats this process until it will succeed. If an instance fails to process a message in all inherited unit worlds, processing the message is failed. Also, a basic mechanism for handling multiple inheritance is provided.

2.3 part_of link

A *part_of* link is used to define a composite instance having lower level instances as its parts. As shown in Fig. 4, this link extends between unit worlds in a whole-to-part direction (Considering its name, the link should have a part-to-whole direction, but in the figure it is shown as having the opposite direction, reflecting the access path of information). The *part_of* link also connects corresponding instances. In this respect, it differs from an *is_a* link since no *is_a* link exists between the instances. A *part_of* link has meta-logical meaning that some theory refers to other theory as data objects. Note that these two kinds of *part_of* links have different roles. While a *part_of* link between unit worlds (indicated by a broken line) represents general facts, such that the eyes are part of the face, a *part_of* link between instances (indicated by a zig-zag line) represents a specialized situation, such that a particular face has its own eyes as its parts. This difference is reflected in implementation methods; that is, a *part_of* link between unit worlds is represented by such an assertion stored in the unit world corresponding to the *whole* that says "(LocalName,PartWorld) *part_of* WholeWorld" where LocalName, PartWorld and WholeWorld are a local identifier of a part, a name of a part unit world and a name of a whole unit world respectively. However, a *part_of* link between instances is established by a communication channel between them.

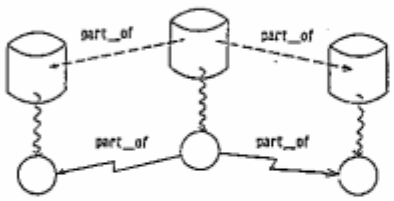

Fig.4 *part_of* link

A *part_of* link between unit worlds is traced when creating a composite instance in order to find what are its components, and instances to be used as the parts are created at the same time. Since local names for parts represent their roles in a composite, they are local in its composite rather than global in the entire system.

2.4 manager_of link

A *manager_of* link connects an instance to a unit world as does an *instance_of* link, but instances connected by a *manager_of* links play an entirely different role. One *manager_of* link is attached to each unit world, (indicated by a double line), as shown in Fig. 5. However one instance can be connected to more than one unit worlds by *manager_of* links. An instance connected to a unit world by a *manager_of* link manages the unit world. Such an instance is called a manager. The functions of a manager include modifying axioms stored in a unit world (i.e., a set of PP or CP clauses or ...) and generating or eliminating instances belonging to the unit world. Note that a manager itself is connected to some unit world by an *instance_of* link and its functions are described in the unit world (Fig. 6). These functions are activated when managers receive messages from other instances.

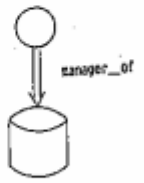

Fig.5 *manager_of* link

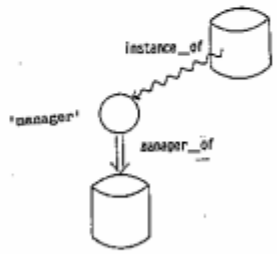

Fig.6 *manager* instance

The relation between a unit world *O* and the unit world *M* whose instance manages the unit world *O* is analogous to the relation between object and meta theory, since the unit world *M* describes the knowledge necessary to manage the unit world *O*. Based on this object-meta hierarchy, assimilation/acquisition of object knowledge can be performed by a manager. This logically natural implementation of knowledge assimilation/acquisition will be described in the chapter 4.

A method of amalgamating object-level and meta-level processing has been proposed by Bowen & Kowalski (Bowen and Kowalski 1981) and implemented in Prolog by Miyachi et al. (Miyachi *et al.* 1984(b)). This method employs a procedure called *demo* which checks the provability of a Prolog program considered as a theorem. *demo* is a predicate having four arguments, i.e., a set of axioms, a goal, a control and a proof tree. It tries to demonstrate that a certain goal is derived from a given set of axioms under a given control and gets a proof tree as a result.

We have expanded *demo* into a predicate named *simulate* to check the provability of CP programs (Furukawa 1984). *Simulate*, which itself is written in CP, is also the core of the Mandala processing system.

3 KNOWLEDGE PROGRAMMING ISSUE

In the previous chapter, we have introduced basic components of Mandala and described the logical interpretation of them. On knowledge programming, the most important feature is its expressive power. Mandala basically inherits its expressive power from its base language KL1. Since provers (query handlers) can be active in parallel and can send messages each other, this gives the basic framework for distributed problem solving, where many problem solvers cooperate to solve one big prob-

lem. In fact, each prover can be seen as an actor which solves a part of a big problem using its knowledge and exchanges information with other provers. The communicating distributed problem solvers can provide powerful basis for problem solving. Inference procedures of provers associated with different unit worlds may not be the same, because a user can define his own inference procedures for some provers. An inference procedure of a prover depends on its associated unit world. Examples of inference procedures and unit worlds are:

Inference procedure	A set of axioms
PP interpreter	PP program
CP interpreter	CP program
First order prover	First order predicates
Term rewriting system	Equation system

As already mentioned, generally a prover is a proof system for a unit world based on some inference procedure. However, like procedural interpretation of Prolog, several pragmatic interpretation of provers are possible. The most general interpretation is the view of regarding provers as actors. Mandala realizes this general actor as a prover which is a CP interpreter. The more special pragmatic interpretation is possible to other kinds of provers. For example, Prolog interpreter can be seen as an rule inference engine by regarding Horn clauses as rules. It is quite important for a problem solving system to provide several kinds of inference mechanisms. Combining the parallel inference mechanism achieved by distributed provers and the variety of user-defined inference systems, Mandala provides powerful framework for constructing a large problem solving system.

In the following sections, the basic implementation scheme of Mandala constructs are presented. In the first section, the implementation of instances, the basis of the parallel inference, will be given, and the general view to consider instances to be actors will be explained. In the second section, as an example of inference systems, the rule inference engine, which is a PP interpreter that manipulates certainty factors and returns proof tree, will be presented. In the last section, a programming environment for manipulating instances interactively will be presented.

3.1 Instances

In this section we show the representation of unit worlds and instances in CP. It is assumed that readers are familiar with CP (Shapiro 1983(a)).

A unit world is represented as a named set of axioms. A name of a unit world is global identifier which can be used to refer the unit world from other unit worlds and also from all instances. A syntactic form of a unit world is as follows:

```
<unit world name>(<axiom_1>).
                  .
                  .
                  .
<unit world name>(<axiom_n>).
```

Information concerning *is_a* and *part_of* illustrated in chapter 2 is placed in a unit world as axioms. An instance is an active object, which can hold local states and can send and receive messages. An instance is implemented by a perpetual process which takes local states as arguments. In other words, an instance is realized as a chain of goal reductions. A goal always takes the form:

```
instance(<name>, <input stream>, <world>)
```

where `name` is an identifier of the instance and `input stream` is a stream of messages received by the instance. The third argument, `world`, conceptually represents a set of axioms contained in the unit world which is associated with the instance. The second and third arguments are always used as read only. As mentioned above, axioms contained in `world` are CP program, PP program, First order predicates, equation system and so on. The following is a CP program that solves the above goal.

```
instance(Name, [Message | Input], World):-
        simulate(Name, Message, World, NewWorld),
        instance(Name, Input?, NewWorld?).
instance(Name, [], World).
```

The first clause describes the case in which there is at least one message in the second argument, `input stream`. In this case, an instance solves the message using the set of axioms in the associated unit world, which specified in the third argument, by `simulate` predicate. `simulate` predicate returns a new set of axioms to the fourth argument after solving the message. The second goal, recursive call to `instance`, is activated when the `NewWorld` will be fixed and tries to solve subsequent messages. The second clause describes the case in which the `input stream` becomes empty. In this case, an instance terminates. In the above program, the `simulate` predicate can be seen as a prover. User can define his own proof procedure for an instance. Different proof procedures are realized by different programs for the `simulate` predicate. Below, the program of the `simulate` predicate which solves CP programs (Shapiro 1984) are shown (We show simplified `simulate` program in which the first and fourth argument are omitted because they are irrelevant here).

```
simulate(true, World).
simulate((A, B), World) :-
    simulate(A, World), simulate(B, World).
simulate(A, World) :- system(A) | call(A).
simulate(A, World) :- clauses(A, Cs, World) |
    simulate_resolve(A, Cs?, B, World),
    simulate(B?, World).
simulate_resolve(A, [C|Cs], B, World) :-
    simulate_unify(A, C, G, B),
    simulate(G?, World) | true.
simulate_resolve(A, [C|Cs], B, World):-
    simulate_resolve(A, Cs?, B, World) | true.
simulate_unify(A, (A:-(G|B)), G, B).
simulate_unify(A, (A:-B), true, B).
simulate_unify(A, A, true, true).
```

In the program, given a goal and a world, the `clauses` predicate returns the axioms which can solve the given

goal, and the `call` predicate solves the goal given as an argument. Another example of the program of the `simulate` predicate is shown in section 3.2. Users can define his own `simulate` predicate as a unit world. Which definitions of the `simulate` predicate should be used for an instance must be specified at the time of instance creation.

Each instance is associated with a unit world by *instance_of* link. Such unit worlds are used as a template for creation of instances. Information concerning *instance_of*, *part_of* and *manager_of* links illustrated in chapter 2 is placed in `<world>` at the time of instance creation. In general, more than one instance can be created from a single unit world, and they share the knowledge in the unit world. Individual instances, however, are not identical with each other even if they are created from the same unit world. This difference among instances comes from their histories of messages received and processed. The third argument, `<world>`, of instance predicate is used to keep states of individual instances reflecting the histories. Specifically, `<world>` contains not only axioms in the unit world, but also axioms which have been added to or deleted from the unit world in the course of message processing.

The actual `simulate` predicate is more complicated than shown above, because it solves the so-called property inheritance. Since the unit world taken as an argument of `simulate` predicate may only represent a fragment of a theory, `simulate` predicate must solve a goal by expanding an available world through *is_a* links when a goal can not be solved under the current world. The following is more specific description of actions taken by `simulate` predicate: When the `simulate` predicate receives a goal (a message), it first tries to solve the goal using the axioms stored in the unit world associated with the instance (the unit world which is connected to the instance by *instance_of* link). If it fails, it extracts an axiom in the form of `Name` *is_a* `W`. from the current world and tries to solve the goal using *the template unit world + W*. If `simulate` predicate fails again, it tries to solve the goal by further expanding the world by tracing *is_a* relation. Generally, `simulate` searches a tree, the root of which is the template unit world of the instance and consists of unit worlds connected via *is_a* relations, depth first, from left to right, for a combination of unit worlds which can solve the goal.

If `Goal` is `add(C)` or `delete(C)` (which means addition or deletion of an axiom C, respectively), `simulate` predicate updates a world and returns a new world at the fourth argument.

Note that, since added axioms are stored in the `World` which is kept as an argument of `simulate` predicate, an instance can keep logical variables, which are shared with other instances, without loss of the property *share*. Therefore channel variables to other instances can be kept in a world so that an instance can send messages to other instances by instantiating these channel variables to messages.

The creation of an instance from a unit world is performed by the manager of the unit world. Generally, a manager itself is also created from the unit world `Manager` or a unit world which has `Manager` above the *is_a* hierarchy. In the following, the part of the definition of `Manager` which is relevant to the creation of a new instance is shown (it is written in CP).

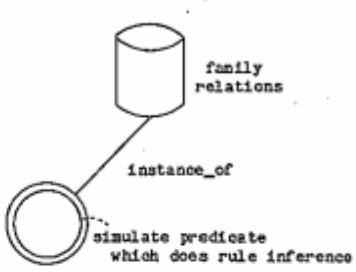

Fig.7 Rule inference engine

```
Manager(create(Name,Goals,InfProc):-
    Me  manager_of UnitWorld &
    instantiate(UnitWorld,Name,DW) |
    instance(Name, [init | Goals],DW):InfProc).
```

This axiom is invoked when the received message is `create(Name, Goals,InfProc)`, which indicates creation of a new instance with inference procedure `InfProc`. When a manager, an instance of the unit world `Manager`, receives this message, the manager creates a new world by the `instantiate` predicate and invokes a new goal which represents the new instance. `InfProc` attached to the goal specifies the inference procedure (the program of `simulate` predicate) used by this `instance` goal. At the time of instance activation, a message `init` is given for the purpose of initialization. If the instance is a composite instance and has parts, the `init` message is used for automatic creation of the parts.

3.2 Rule inference

As is already shown by the recent researches (Clark and McCabe 1982) (Shapiro 1983(b)), by regarding clauses and top-down proof procedure as inference rules and inference procedure respectively, logic programming can be also seen as rule-oriented programming. In this sense, Mandala also provides rule-oriented programming without introducing additional computational mechanism. However, in practice, we need mechanism to handle a certainty factor and to extract a proof tree. In this section, we show the CP program of rule inference engine, which is more elaborated program of `simulate` predicate presented in the previous section (`simulate` program that solves PP program).

In Fig.7, an instance which performs as a rule inference engine is shown. In the figure, the unit world stores individual family relations as data and inference rules, which are represented as Prolog clauses. The instance of this unit world has rule-inference `simulate` predicate, the program of which is shown below, and perform as a rule inference engine.

The examples of the contents of the unit world is shown below.

```
fclause(father(domdom,damdam),true,50).
fclause(mother(domdom,tomtom),true,30).
fclause(father(damdam,dimdim),true,80).
fclause(mother(damdam,tamtam),true,40).
fclause(father(dekeden,damdam),true,20).
fclause(mother(dekeden,tokoton),true,70).
brother(X,Y) :- father(X,F),father(Y,F),X\=Y.
brother(X,Y) :- mother(X,M),mother(Y,M),X\=Y.
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

where `fclause(P,Q,N)` represents a Prolog clause `P :- Q` with certainty factor `N` (certainty factor is a number between 0 and 100). All other clauses such as brother, parent and ancestor predicates are regarded as having certainty factor 100.

When `simulate` predicate receives a goal (a message), it solves the goal by invoking the CP program `prove(Goal,[],Tr,Tr,Chan)`. The answer, which is a list of all solutions, is returned to the fifth argument as a stream. During the solution process, `simulate` predicate creates many child processes, each of which corresponds to a process searching alternative solutions. These processes invoke goals of the form, clauses(Goal,List,World), in order to get rules and data that can be unified with the goal `Goal` they need to solve.

In the following, the CP definition of the rule inference `simulate` predicate is shown (again irrelevant arguments are omitted).

```
simulate(Goal,World) :-
    prove(Goal,[],Tree,Tree,Proofs,World) ,
    show_stream(Proofs?).
prove(A,[],((A<=)/100)<<[100],
          Tree,[Tree],World) :-
    system(A) & call(A) | true.
prove(A,[(C,TC<<X)|D],((A<=)/100)<<[100],
          Tree,Chan,World) :-
    system(A) & call(A) |
    prove(C,D,TC<<X,Tree,Chan,World).
prove((A,B,C),D,(TA & TB)<<[X|Y],
          Tree,Chan,World) :-
    prove(A,[((B,C),TB<<Y)|D],
          TA<<[X],Tree,Chan,World).
prove((A,B),D,(TA & TB)<<[X,Y],
          Tree,Chan,World) :-
    prove(A,[(B,TB<<[Y])|D],TA<<[X],
          Tree,Chan,World).
prove(A,C,STree,Tree,Chan,World) :-
    clauses(A,Clauses,World) |
    try_each(Clauses,A,C,STree,Tree,Chan,World).
try_each([],_,_,_,_,[],World).
try_each([(A0:-B)<<F|R],A,C,ST,T,Chan,World) :-
    copy(A+C+ST+T,A1+C1+ST1+T1) &
    A0=A1 & ST1=((A0 <= TB)/CF<<[CF]) |
    prove(B,[(cf(F,CF1,CF),Tcf<<Xcf)|C1],
          TB<<CF1,T1,Chan1,World),
    try_each(R,A,C,ST,T,Chan2,World) ,
    merge(Chan1?,Chan2?,Chan).
try_each([_|Clauses],A,C,S,T,Chan,World) :-
    otherwise |
    try_each(Clauses,A,C,S,T,Chan,World).
```

The program is based on the two papers (Clark and McCabe 1982) and (Shapiro 1983(b)). The main components of the program are two CP programs, `prove` and `try_each`. `prove` is a 6-ary predicate.

```
prove(Goal,Stack,SubTree,Tree,Channel,World)
```

The predicate `prove` solves the given goal `Goal` in `World` and returns all the solutions to the fifth argument `Channel` in the form of stream of proof trees. `Stack` is used as a control stack. `SubTree` and `Tree` are used to keep a partially obtained proof tree. A general form of a proof tree is as follows.

```
Proof Tree :== (P <= ST1 & ST2 & ...  & STn)/CF
                  or
               (true<=)/100
```

where `ST1, ST2, ... , STn` are all proof trees and `CF` is a certainty factor of the inference deriving `P`. Below an example of a proof tree is shown when the given goal is `ancestor(domdom,X)`.

```
(ancestor(domdom,dimdim)<=
   (parent(domdom,damdam)<=
      (father(domdom,damdam)<=(true<=)/100)
      /50)
   /50 &
   (ancestor(damdam,dimdim)<=
      (parent(damdam,dimdim)<=
         (father(damdam,dimdim)<=(true<=)/100)
         /80)
      /80)
   /80)
/40
```

This proof tree can be read:

The goal `ancestor(domdom,dimdim)` is derived with the certainty factor 40 from two subgoals, `parent(domdom,damdam)` and `ancestor(damdam,dimdim)`, using the second clause of the `ancestor` program. Each of two subgoals are derived with the certainty factor 50 and 80 respectively, and so on.

3.3 Programming Environment

Considering Mandala as a knowledge programming system, it is desirable to develop a powerful programming environment to facilitate interactive program development and debugging. Besides the functions to manipulate unit worlds, the programming environment should

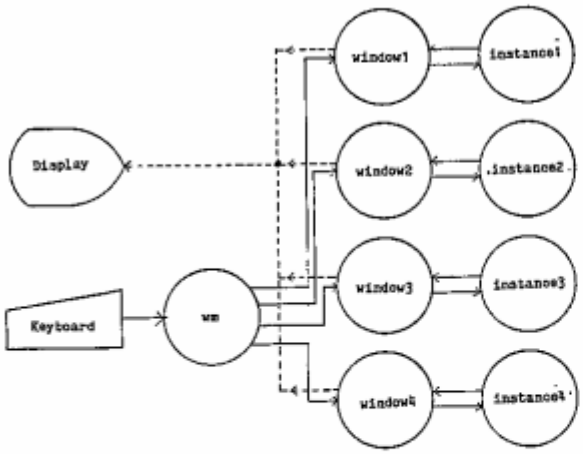

Fig.8 The programming environment

provide users functions to manipulate instances such as monitoring the history of the behavior of each instance, updating the local state of each instance and changing the communication channel network among instances dynamically. It is also important for users to use such functions interactively. In the following, the experimental implementation of the programming environment is described.

The basic configuration of the programming environment is based on the multiple window manager (Shapiro and Takeuchi 1983), which manages the outputs of concurrent processes by using the multiple windows associated with each of them. The functions described below are added to it for the purpose of managing instances :

(a) creating an instance and the window associated with it

(b) modifying the local state of an instance

(c) changing the communication network of instances

(d) distributing messages to instances

The multiple window manager plays a role of the user interface in programming environment, and keeps pairs of the names and the input channels of all the instances created by it. The multiple window manager accepts the messages to perform all the functions listed above in addition to distributing the messages to instances. All the inputs and outputs of an instance are displayed onto the screen of the window associated with it, and recorded in the window as the history of it. It is able to monitor the history of the behavior of an instance by monitoring the window associated with it. It is also able to keep tracks of the history of the behavior of an instance by using session manager mode of the multiple window manager.

In Mandala, the multiple window manager is defined by the axioms stored in the unit world **Wm** like other knowledge, and the configuration of the programming environment realized by the **Wm** is shown in Fig. 8.

The unit world **Wm** holds the CP program **wm** as shown below. The first argument of the predicate **wm** is the input message stream to the **Wm**, and the second argument is a list of pairs of the name of an instance and the output channel to the window associated with the instance. The messages to the instance are also sent to the instance through this channel.

```
Wm(wm([(Class,create(Name,(XO,YO,W,H)))|Input],
                              ListOfChan):-
    send(Class,create(Name,InCh,OutCh),
              ListOfChan,NewListOfChan) |
    window(([show(Name)|In],InCh,OutCh?),
           ((XO,YO,W,H),YO,(C,C,C,C)),Name),
    wm(Input?,[(Name,In)|NewListOfChan])).
Wm(wm([(Name,edit)|Input],ListOfChan):-
    find_process(Name,ListOfChan,
          [show(Name),edit|In],ListOfChan1) |
    wm(Input1?,[(Name,In)|ListOfChan1])).
Wm(wm([(Name,get_chan(Chan))|Input],ListOfChan):-
    find_process(Name,ListOfChan,
                      In,ListOfChan1) |
    wm(Input?,[(Name,Chan1)|ListOfChan1]),
    merge(Chan1?,Chan?,In)).
Wm(wm([(Name,Msg)|Input],ListOfChan):-
    send(Name,Msg,ListOfChan,NewListOfChan) |
    wm(Input?,NewListOfChan)).
```

On receiving a `(Class,create(Name,(XO,YO,W,H)))` message, the **Wm** creates an instance named **Name** of the unit world **Class** by sending a `create(Name,InCh,OutCh)` message to the manager of the unit world **Class**. In this case, the definition of **Manager** listed in section 3.1 is modified in order to handle the output channel of an instance. The output channel is held in a local state of the instance **Name**, and its value is initially a difference list `Out(OutCh,OutCh)`, i.e. an uninstantiated stream.

Then, the window process associated with the instance is created and the window is displayed to the user terminal according to the parameter `(XO,YO,W,H)`, where the **XO** and **YO** specifies the location of the window and the **W** and **H** specifies the size of the window. This window process has the same name as the instance, i.e. **Name**, and holds the input and output channel of the instance. All the inputs to the instance and the outputs of the instance are displayed to the screen of the window through these two channels, and are kept in the window as text. The inputs to the instance are sent from the keyboard through the **InCh**, and the outputs of it are sent to the window process associated with it through the **OutCh** by using the predicates **show** and **show_stream** of the unit world **Instance**, which is the highest-level unit world of all unit worlds along *is_a* hierarchy. The predicate `show(X)` sends the output **X** and the predicate `show_stream(L)` sends the elements of a stream through the output channel by instantiating the output channel variable of an instance.

If the **Wm** receives a `(Name,edit)` message, it tries to find the input channel **In** of the window process associated with the instance **Name** by the predicate **find_process** defined in the **Wm**. Then the window process enters the session manager mode, sm mode for short, by the message

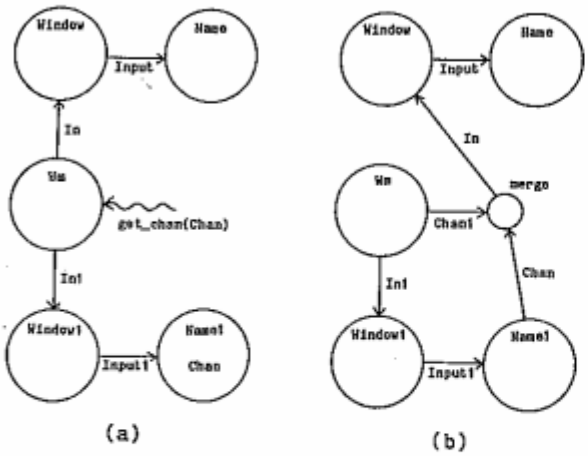

Fig.9 Creation of channel connection

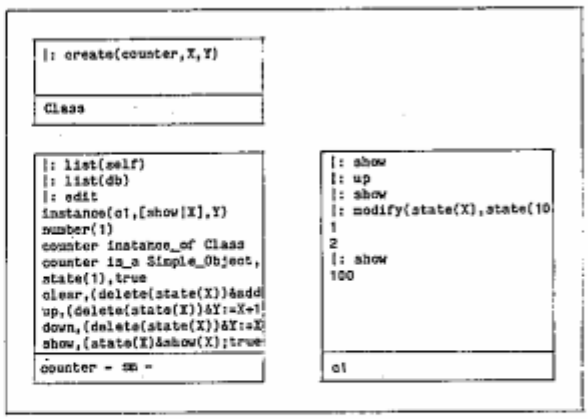

Fig.10 The example of screen output

edit. In sm mode, the window process becomes an editor of the history of the instance, i.e. the window scrolls up and down the screen to see all the text displayed to the window. The message **exit** causes the window process to exit from the sm mode and return back to the normal window manager mode.

If the **Wm** receives a **(Name,get_chan(Chan))** messages, it tries to find the input channel **In** of the window process associated with the instance **Name** as shown in Fig. 9 (a). Then the channel **Chan** given by a user and the newly created channel **Chan1** are merged into **In** as shown in Fig. 9 (b).

The channel **Chan1** is used to send messages from **Wm** to the window process, that is, the messages are sent through the channel **Chan1** and **In** (see Fig. 9 (b)). And the messages are sent to the instance **Name** unless a message is the query to the window process itself.

The channel connection between the instance **Name** and the other instance **Name1** is created by the given channel **Chan** as shown in Fig.9(b). The channel **Chan** is used to send messages from the instance **Name1** to the window process associated with the instance **Name** and the instance **Name** itself. As a result, a new communication channel between two instances **Name** and **Name1** is created dynamically.

If a message, say **(Name,Message)**, is not the query for **Wm**, it is regarded as the message to the instance **Name**. At first, the message is sent to the window process associated with the instance **Name**. The window process displays the message and sends it to the instance transparently.

The modification of the current local world of an instance is done by the predicate **add**, **delete** and **modify** of the unit world **Instance**. These predicates are used to add, delete, or modify the axioms in the current local world of an instance.

The screen output of the example execution of **Wm** is shown in Fig. 10. The window labeled **Class** is associated with the manager **Class** and the window labeled **counter** is associated with the manager of the unit world **counter** which is created by **Class**. The message **create(counter,X,Y)** appeared in the window **Class** indicates the creation of the manager of **counter**. Note that messages preceded by **|:** are input messages and other messages are outputs of instances.

The window **counter** is now in the sm mode by **edit** message. The prompt **-- sm --** within the label of the **counter** window indicates that the window is in the sm mode.

The window labeled **c1** is associated with the instance of **counter** which is created by the manager of **counter**. The modification of the unit clause **state(X)** is done by the message **modify(state(X),state(100))** as shown in the **c1** window.

4 KNOWLEDGE BASE MANAGEMENT ISSUES

The basic functions of a knowledge base management system include knowledge representation, knowledge utilization and knowledge acquisition. Since the basis of knowledge representation is included in the basic functions of Mandala, here, we show how the functions of knowledge utilization and knowledge acquisition are achieved.

4.1 Knowledge base search

Knowledge utilization depends on an effective search strategy for the knowledge base. Therefore, we focus our discussion on the knowledge base search function.

In Mandala, a unit of knowledge is represented by a unit world. Such a unit world may be regarded as a relation table in a relational database. To retrieve information existing in a unit world, it is necessary to have a *librarian* that searches for a specific unit world. Such librarian can be embodied as an instance of the unit world. That is, if we assume a unit world to be an item of knowledge, we can regard an instance connected by an *instance_of* link as

a librarian that searches the knowledge base, rather than as an instance represented by the unit world. The concept of a librarian explained here is an another interpretation of an instance which is explained as a prover in chapter 2.

A librarian dedicated to the knowledge base can periodically update the knowledge base. The updated knowledge is held by the librarian itself, while the original knowledge base remains unchanged. This function can be used to implement hypothetical reasoning and can be further expanded to place different hypotheses in several librarians, thereby making it possible to deal with such problems as assuming many unit worlds at the same time and determining which hypothesis is most likely. In MYCIN (Shortliffe 1976), for example, the likelihood of assuming an infections disease is computed for all infections diseases and several hypotheses are selected as conclusions in descending order of the degree of their likelihood. This may be said to be a similar problem. Local updating of the knowledge base can be implemented by a change in the state of the librarian itself. Note that this differs from global updating, which is performed by the knowledge base manager. Global updating actually rewrites the unit world; it has a much greater influence on the system as a whole, and requires strict checking. This problem will be addressed in the following section.

4.2 Knowledge base management and knowledge assimilation

There are different aspects of knowledge acquisition for knowledge base management, but from the standpoint of conformity with a logic programming language, we will focus on assimilation problem (Bowen and Kowalski 1981), (Miyachi *et al.* 1984(b)). The purpose of assimilation is to acquire knowledge while, at the same time, ensuring that it is free from logical contradictions and redundancies.

We implemented a knowledge assimilation program which manages consistency and eliminates redundancies in ordinary sequential Prolog (Miyachi *et al.* 1984(b)). The program employs the approach of dividing knowledge into positive knowledge, which provides specific facts concerning individual instances, and negative knowledge, which provides conditions to be satisfied by such specific facts. Negative knowledge is checked when new positive knowledge is being acquired. This program incorporates an extension of the **demo** predicate mentioned in chapter 2. However, it was found that the knowledge assimilation program using the extended **demo** predicate was not quite practical in terms of execution efficiency. If it is necessary to demonstrate that any new knowledge acquired is consistent with all negative knowledge, the execution time required is proportional to the amount of negative knowledge. Besides, the execution time for redundancy elimination is proportional to the amount of positive knowledge, since the algorithm for redundancy elimination checks each unit of positive knowledge for redundancy.

We found that it is possible to implement a more efficient algorithm in Mandala. As for consistency checking, segmentation of positive knowledge and negative knowledge permits reducing the amount of negative knowledge to be checked when acquiring a unit of positive knowledge. Furthermore, the consistency checking can be done in parallel in two levels: checking on different negative clauses, and also checking on each negative clause (Hirakawa *et al.* 1983).

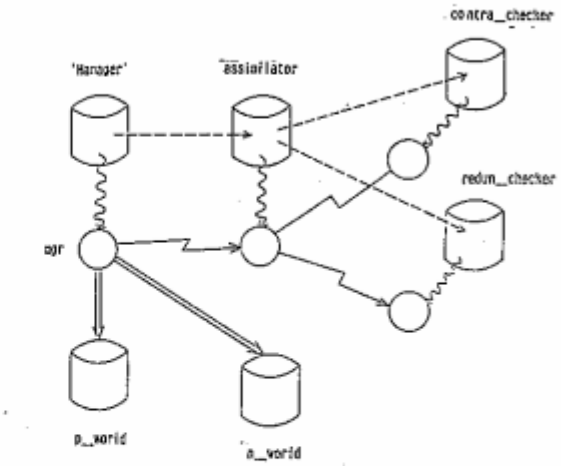

Fig.11 The knowledge assimilation system

For redundancy elimination, it is possible to independently check the redundancy of each unit of knowledge within a limited scope and later eliminate all units of knowledge found to be redundant. Suppose that a graph formed by implication relations (⇒) has no closed loop and that P and Q are independently shown to be redundant from knowledge base T. The redundancy of P, for example, is demonstrated by the derivation of P from T-P. However, as we assumed the absence of a loop of implicit relations, it is impossible that Q is used for proving P and, at the same time, that P is used for proving Q. In such a case, therefore, it is possible to eliminate redundancy by the parallel algorithm referred to above. (As an example in which the above conditions are not satisfied, consider the set of formulas A, B, A⇒B, B⇒A.) This technique can also be applied to Prolog or CP programs which generally do not fall into an endless loop, because they have no loop of implication relations.

Fig.11 roughly illustrates how a knowledge assimilation program is implemented in Mandala. As can be seen from the figure, the knowledge base management program, **Manager**, is expanded to have an assimilator as a part. The assimilator itself has two modules, contradiction-checker and redundancy-checker, which check for contradiction and redundancy, respectively. **Manager** instance, **mgr**, manages two unit worlds called p-world and n-world which contain positive knowledge and negative knowledge respectively.

A prerequisite for improvement of the execution speed is to develop dedicated computers capable of executing KL1 in parallel. This will have to await future research. In the meantime, we can say that Mandala at

least demonstrates the usefulness of parallelism.

5 SUMMARY AND PROSPECTS

In this paper we have discussed the basic framework and implementation of Mandala, a knowledge programming system based on KL1. We have also stated that Mandala provides a basis for a knowledge base management system. However, the development of Mandala is still in a very early stage. We have much more research to be done before the system will be completed.

Of particular interest for Mandala as a programming system, is the introduction of a partial execution mechanism. This is closely related to the concept of parameterization or compilation. If the values of parameters are specified early and the program partially executed, compilation will be performed. In this case, parameters cannot be dynamically changed at the time of program execution. For a programming system, it is desirable to have the capability of freely specifying levels of partial execution. This would allow users to control the trade-off between flexibility and execution efficiency. In a sense, whether a system places emphasis on flexibility or execution efficiency determines whether it is oriented toward knowledge programming or system programming. To provide a practical system with these two facets, it is required to realize smooth transition from one facet to the other by means of a partial execution mechanism.

Another possible extension of Mandala as a knowledge base system is enhancement of its expressive power. We defined unit world knowledge as Pure Prolog and/or CP programs, but this limitation is too strict. The knowledge representation language KRYPTON is capable of describing statements in first-order predicate logic, which is more powerful than Horn logic in dealing with incomplete knowledge. A similar capability is a possible direction in which Mandala will be extended. This requires a powerful theorem prover in first-order predicate logic.

ACKNOWLEDGEMENTS

We wish to express our thanks to Kazuhiro Fuchi, Director of ICOT Research Center, who provided us with the opportunity to pursue this research in the Fifth Generation Computer Systems Project at ICOT. We would also like to thank Hiroyasu Kondou and other ICOT research staffs, members of ICOT Working Groups 2, 3 and 4, and the Fujitsu, NEC and Oki Electric researchers who participated in discussions with the knowledge representation task force. Our thanks go especially, to Dr. Fumio Mizoguchi of Science University of Tokyo, who, chairman of Working Group 4, not only provided us with insights on the development of Mandala but consistently gave us valuable advice in subsequent discussions. We also gratefully acknowledge the informative discussions we had with Dr. Ehud Shapiro from Weizmann Institute of Science and Dr. Keith Clark from Imperial College during their stay at the ICOT Research Center.

REFERENCES

Bobrow, D. G., Stefik, M. *The LOOPS Manual (Preliminary Version).*, XEROX PARC Knowledge-based VLSI Design Group Memo KB-VLST-81-13, 1983.

Bowen,K. A., Kowalski, R. A., *Amalgamating Language and Meta Language in Logic Programming.*, School of Computer and Inform- ation Sciences, University of Syracuse, 1981.

Brachman, R.J. et al., *KRYPTON: A Functional Approach to Knowledge Representation.*, Fairchild Laboratory for Artificial Intelligence Research, Fairchild TR No.639, 1983.

Clark, K., McCabe, F., *PROLOG : A Language for Implementing Expert Systems.*, In D. Michie and Y.H.Pao (ed.), Machine Intelligence 10, 1982.

Furukawa, K. et al., *The Conceptual Specification of the Kernel Language version 1.*, ICOT TR-054, 1984.

Genesereth, M. R., et al., *MRS Manual.*, Stanford University, Stanford Heuristic Programming Project Memo HPP-80-24, 1980.

Goldberg, A., Robson, D., *Smalltalk-80 : The language and its implementation.*, Addison-Wesley, 1983.

Hirakawa, H. et al., *Implementing an OR-Parallel Optimising Prolog System (POPS) in Concurrent Prolog.*, ICOT TR-020, 1983.

Kehler, T. P., Clemenson, G. D., *KEE: The Knowledge Engineering environment for Industry.*, IntelliGenetics, 1983.

Kitakami, H. et al., *A Methodology for Implementation of a Knowledge Acquisition System.*, Proceedings of 1984 International Symposium on Logic Programming, 1984.

Miyachi, T. et al., *A Knowledge Assimilation Method for Logic Databases.*, to appear in New Generation Computing, 1984(a).

Miyachi, T. et al., *A Knowledge Assimilation Method for Logic Databases.*, Proceedings of 1984 International Symposium on Logic Programming, 1984(b).

Novak, G. S. Jr., *GLISP : A Lisp=based Programming System with Data Abstraction.*, AI MAGAZINE, FALL 1983.

Shapiro, E., *A Subset of Concurrent Prolog and Its Interpreter.*, Institute for New Generation Computer Technology, ICOT TR-003, 1983(a).

Shapiro, E., *Logic Programs with Uncertainties: A Tools for Implementing Rule-Based Systems.*, Proc. of IJCAI 83., 1983(b).

Shapiro, E., Takeuchi, A., *Object Oriented Programming in Concurrent Prolog.*, New Generation Computing, Vol.1, No.1, 1983.

Shapiro, E., *Systems Programming in Concurrent Prolog.*, Proc. of Principles of Programming Languages, 1984.

Shortliffe, E. H., *Computer-Based Medical Consultations : MYCIN.*, American Elsevier, 1976.

Smith, R. G., *Structured Object Programming in Strobe.*, Schlumberger Technology Corporation, 1984.

Takeuchi, A., Furukawa, K., *Interprocess Communication in Concurrent Prolog.*, Logic Programming Workshop'83 in Portugal, 1983.