Interoperation for Development and Improvement of Expert Systems

Noriaki IZUMI Akira MARUYAMA
Atsuyuki SUZUKI Takahira YAMAGUCHI
Dept. Computer Science, Shizuoka University
3–5–1 Johoku Hamamatsu Shizuoka 432–8011 JAPAN
{izumi,s4038,suzuki,yamaguti}@cs.inf.shizuoka.ac.jp

Abstract

This paper proposes the interoperation environment which enables an expert system to get information available to improve its performance from others. First, we have given a method library of reusable templates in order to provide a correspondence between specification and implementation of inference structures. Next, a cooperation method has been presented, using the difference arising in the context of the correspondence between inference primitives of an originator and those of recipients. The wrapper with conversion facilities has been also provided, using a common domain ontology developed manually. After designing and implementing such an interoperation environment, experiments have been done among four heterogeneous expert systems. Furthermore, it has been shown that an expert system finds a way to perform a given task better by the interoperation with other three expert systems.

Proceedings of the IJCAI-99 workshop on Ontologies and Problem-Solving Methods (KRR5) Stockholm, Sweden, August 2, 1999

(V.R. Benjamins, B. Chandrasekaran, A. Gomez-Perez, N. Guarino, M. Uschold, eds.)

http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/

1 Introduction

As expert systems have been built up in many real fields over the past decade, the research on Cooperative Distributed Expert Systems (CDES) has emerged, integrating two kinds of technology from knowledge acquisition and software agents. The work in the field of CDES focuses on the cooperation among distributed expert systems but has not yet been getting into cooperation in real complex domains at a semantic level.

As seen in the fields of software agents and CDES, at present, multiagent and knowledge engineering technologies are becoming integrated. However, in order to develop robust cooperative knowledge systems in real and large scale industrial applications, the approaches from knowledge level analysis and knowledge modeling are few and so we are still in shallow interoperation just at a syntactic level among distributed heterogeneous expert systems. Even if useful information is acquired, there is a significant issue how the information is reflected in the implementation-structure of knowledge systems. Thus, in this paper, we propose an environment for deep interoperation among four heterogeneous expert systems at a semantic level, modeling them at a proper level of granularity of knowledge, defining the relationship between models and implementations, using the difference arising in the context of the correspondence between the inference structure of an originator and the one of recipients, and presenting a wrapper with conversion facilities using a common domain ontology.

In the remainder of this paper, we first describe methods of modeling, operationalizing, cooperating and communicating (wrapping) heterogeneous expert systems. Next, we put the methods together into an interoperative environment, INDIES(an Interoperative eNvironment for Development and Improvement

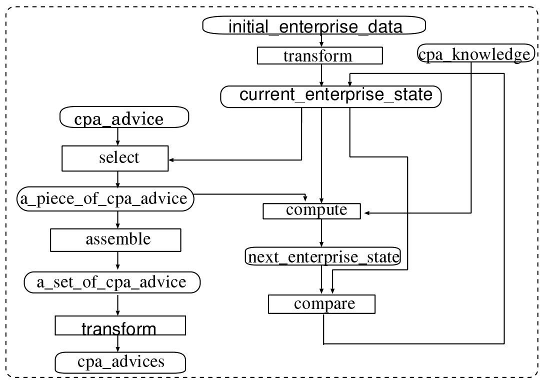


Figure 1: Common KADS specification of FIMCOES

for Expert Systems) for them. The empirical results have shown us that a financial management expert system is supported by other three expert systems in finding a better solution.

2 Modeling and Implementing Distributed Expert Systems

In order for distributed expert systems to exchange useful information applicable to their better performance from others, there are two important considerations: how to specify expert systems at a conceptually acceptable level of details, and how to reflect a received information of alteration on their implementation. The following discusses a method to identify the proper kinds and granularity of information and a method for operationalizing the conceptual models.

2.1 Modeling Expert Systems by Common KADS

Developed expert systems are much different in their implementation details, such as knowledge representation languages and how to run inference engines. If they would exchange information at such implementation details, each expert system would have no way to find information for the improvement in their performance. So, the information about inference engines and knowledge bases must be lifted from the implementation details to some proper conceptual details acceptable to be exchanged. In the field of knowledge engineering, the methodology has recently been developed to specify the semantics of expert systems free from implementation details. Common KADS[Bre94] is one of the well-organized knowledge libraries that provides inference primitives called canonical functions, such as `Select`, `Compare`, `Merge` and so on. In Figure 1, a specification description of the expert system FIMCOES which consults a financial management of an enterprise[Gra94] is given as an example case.

2.2 Correspondence between Specification and Implementation of Expert Systems

Through our previous work[YMG98b], in which we have employed Common KADS as an abstract description method, a significant issue comes up in correspondence between specification and implementation of expert systems. To put the issue concretely, we have to spend more than two weeks in order to change an implementation structure of an expert system according to an alteration obtained by an exchange of specification information. Main causes which costs such a long time can be listed up as follows.

1. Difference of data structures

   An alteration in input, output and reference knowledge of an inference primitive effects not only internal structure of the primitive but also other primitives because of a variety of data structures of knowledge, including their implementation and data call.

2. A variety of refinement methods

   Developers of expert systems often make inference primitives share the program codes and cut the redundancy of data exchange. Because of these varieties, it is difficult to decide which primitive should be changed according to an alteration on an abstract description.

3. Side effects of adding control structure

   An abstract description can be interpreted in different ways of implementation with control structures. For example, we have a lot of options for a behavior of a loop in data flow diagrams such as iteration with conditional, repetition with a counter, a fail loop with backtracking, and so on.

In the further analysis to get the above features over, we've found out that the employment of different languages makes a gap between specifications and implementations in the development process of expert systems. So, design specifications and implementation codes of expert systems should be described in a unified language in order to reflect specification alteration immediately to the implementation structure.

2.3 Building Reusable Templates for Specification and Implementation

From the importance of a unified language for the reflection of the change on a specification description, we rebuild and extends canonical functions into "REPOSIT (REusable Pieces Of Specification-Implementation Templates)" which combines declarative semantics employed in Common KADS and procedural semantics like Prolog. A unit of a description

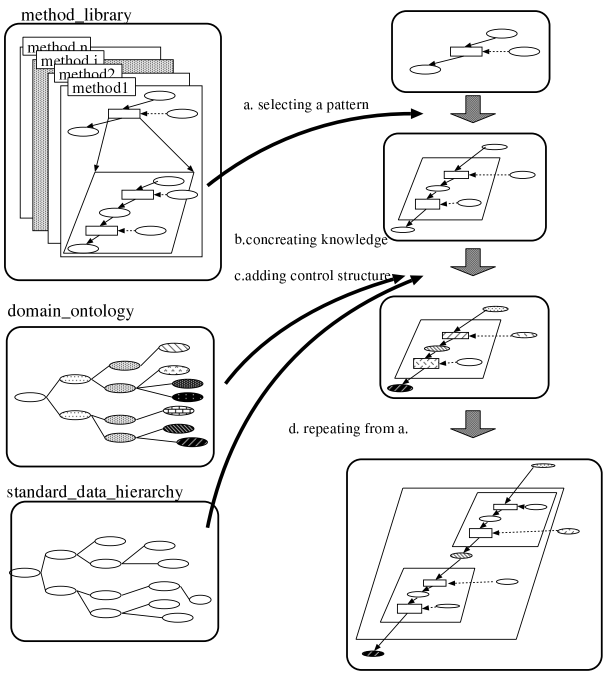


Figure 2: Overview of a Development Process

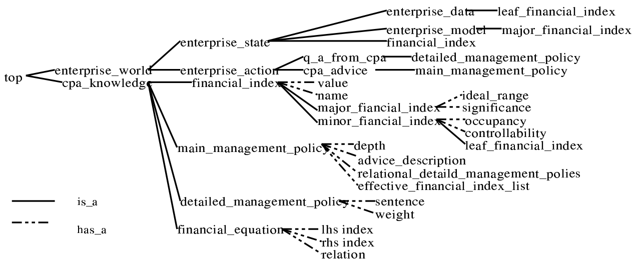


Figure 3: Domain ontology for FIMCOES

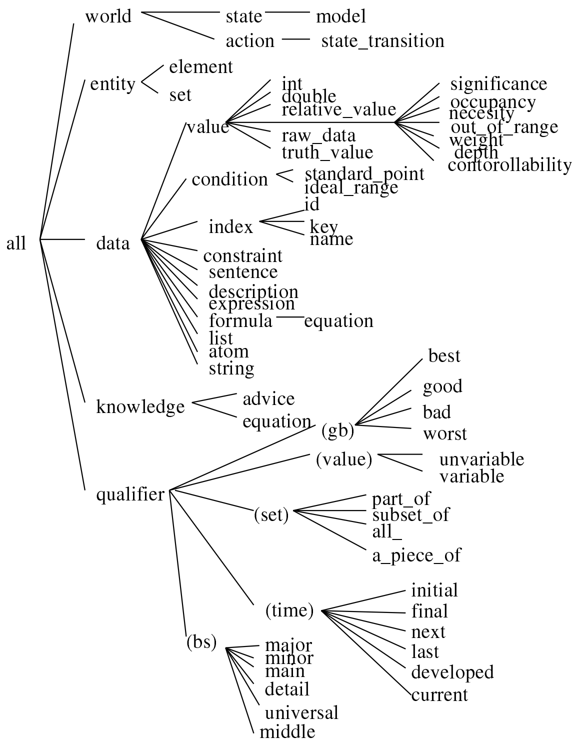


Figure 4: Standard data hierarchy

in REPOSIT, defined as a relationship among input, output and reference knowledge, is called a "unit function". A set of unit functions is rebuilt into a method ontology by abstracting knowledge types of input, output and reference.

Furthermore, patterns of a combination of unit functions, which appear frequently in the development process, are gathered, sorted out and constructed as a method library based on the following standpoint:

(1) providing refinement policies,

(2) standardizing a way of the knowledge (data) management,

(3) classifying the adding patterns of control structures given to specifications.

In order to keep a correspondence between descriptions of specifications and implementations, REPOSIT supports step by step operationalization of an abstract description into a detailed implementation description, as the following way (Figure 2):

a. selecting a pattern of the method library according to a task type of an expert system,

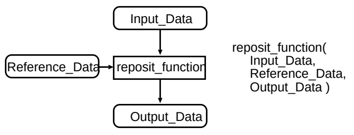
b. concreting knowledge type of input, output and reference by comparing a domain ontology (Figure 3), a standard data hierarchy (Figure 4) and requirement specification,

c. adding a control structure to the description with the obtained information of knowledge type,

d. selecting a pattern for each unit function of the description and continuing the above process.

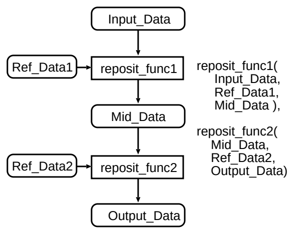
The standard data hierarchy is manually constructed as a domain independent ontology which gives words for expanding domain ontologies such as building a set, picking up an atom of a set, indicating a calculation stage, data structures for implementation details and so on.

Each method of REPOSIT has two type of expressions: one is the relationship among input, output and reference knowledge, which consists of specification library, and the other is a prolog-based representation which consists of implementation library. Figure 5 shows a basic correspondence between expressions in a specification of a knowledge(data)-flow-diagram and



```
reposit_function(
   Input_Data,
   Reference_Data,
   Output_Data )
```

(a) primitive component



```
reposit_func1(
   Input_Data,
   Ref_Data1,
   Mid_Data ),
reposit_func2(
   Mid_Data,
   Ref_Data2,
   Output_Data)
```

(b) sequential connection

Figure 5: Basic relationships between REPOSIT expressions (sequential connection)

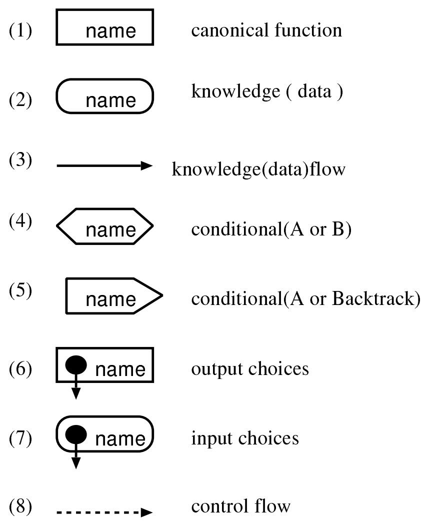


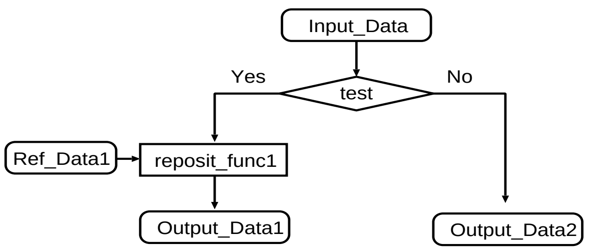
Figure 6: Syntax of a REPOSIT description

those in first-order-predicates. In the figure, rectangles express methods corresponding to unit functions and quarter-circles express knowledge as data used in the connected methods.

To combine description languages for abstract models and implementation structures, we reinforce Common KADS with operational information, given in Figure 6, which enables us to put control structures directly to knowledge-flow-diagrams in the way of Figure 7.

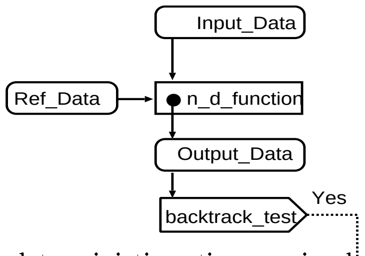
2.4 Operationalization of Refined Models

We augment a knowledge-flow-diagram selected as a pattern with control structures, such as conditional branches, a distinction of deterministic and non-

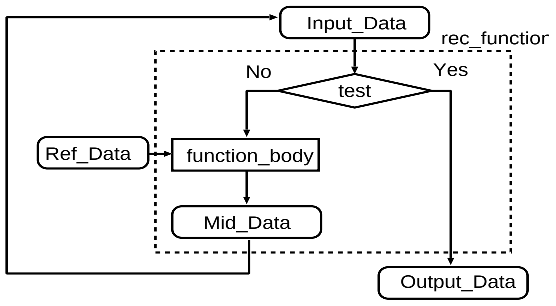


```
{ test( Input_Data), !,
  reposit_func1( Input_Data, Ref_Data1, Output_Data )}
;
  replace( Input_Data, Output_Data )
```

(a) a conditional branch

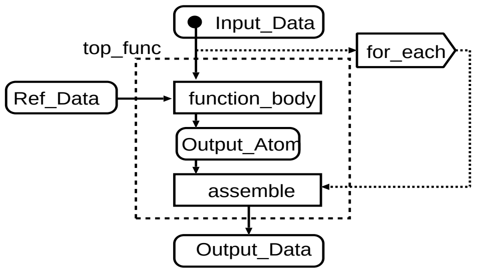


(b) non-deterministic action causing backtrack



```
rec_function(Input_Data, Ref_Data, Output_Data) :-
  { test( Input_Data), !,
    replace( Input_Data, Output_Data )}
;
  { function_body( Input_Data, Ref_Data, Mid_Data ),
    replace( Mid_Data, Input_Data),
    rec_function( Input_Data, Ref_Data, Output_Data)}
```

(c) a loop structure with recursive



```
top_func(Input_Data, Ref_Data, Output_Data) :-
  for_each( Atom, Input_Data, [
    function_body( Atom, Ref_Data, Output_Atom
    assemble( Output_Atom, Output_Data)]).
```

(d) a loop structure with iteration

Figure 7: Control structures of REPOSIT

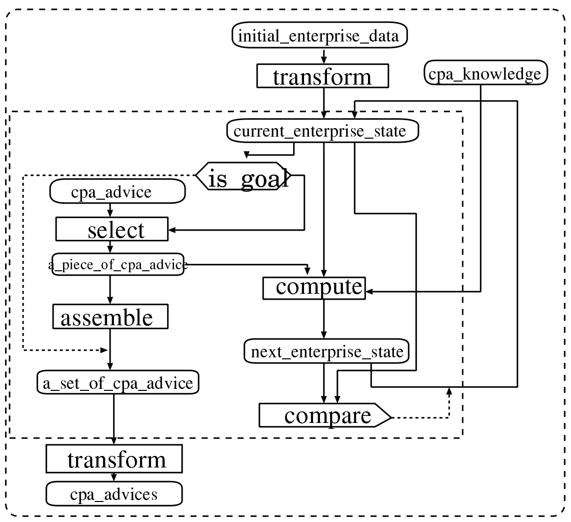


Figure 8: An operationalized specification of FIMCOES

deterministic actions, and repetitions as shown in Figure 7. To put it concrete, the following procedure is given:

1. Adding conditional

   In a knowledge-flow-diagram, a method corresponding a conditional action is changed its shape into a diamond (Figure 7(a):`test`). Conditional action which causes backtrack is expressed in a shape of the home-base style(Figure 7(b):`backtrack_test`). Function call invoked by conditional is connected by a broken line with a diamond,

2. Clarify of non-deterministic actions

   A non-deterministic action which gives an alternative output with backtrack is represented by the a rectangle with a solid circle(Figure 7(b)`n_d_function`),

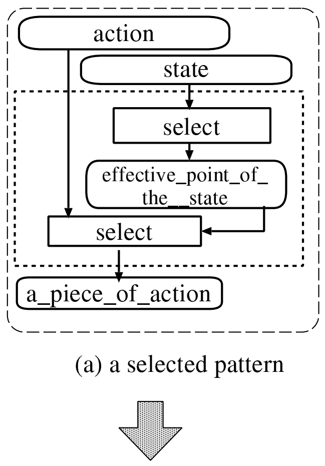
3. Clarify of loop structures

   A loop structure called recursively is enclosed with a broken line(Figure 7(c)(d)).

After applying the above augmentation, we can get a description given in Figure 8.

2.5 Refinement of Abstract Specifications by REPOSIT

In order to refine REPOSIT specifications expressed as augmented knowledge-flow-diagram, we replace a unit function with combinations of unit functions given as a pattern in the method library. Figure 9(a) gives a pattern chosen for the unit function `select` in Figure 8. Each knowledge, appearing as an abstract type



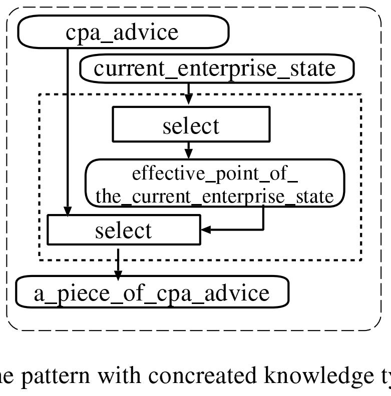

cpa_advice
current_enterprise_state
select
effective_point_of_the_current_enterprise_state
select
a_piece_of_cpa_advice
(b) the pattern with concreated knowledge type

Figure 9: an example pattern selected for `select`

in the chosen pattern, is refined by using a domain ontology and a standard data hierarchy. In order to use knowledge name both descriptions of specifications and implementations, we augment a domain ontology according to the standard data hierarchy in a process of the knowledge refinement (Figure 10). Figure 9(b) shows the knowledge refined pattern from Figure 9(a) as an example. In the process of the example, the domain ontology of Figure 3 is augmented with `current enterprise state` which is constructed by `current` in the standard data hierarchy and `enterprise state` in the domain ontology. Because of this augmentation, we can distinguish the same type of knowledge by its name with the one in a different stage of calculation.

2.6 Implementation into Prolog-Codes

In this paper, we employ atoms and lists as data primitives because of our prolog-based development. To use names of knowledge in the specification directly, we define the primitive expressions of knowledge, which support a generic method of data call by name as follows:

$$\mathtt{atom}(\mathtt{Atom\_id}, [\mathtt{Cat}_1 : \mathtt{Val}_1, ..., \mathtt{Cat}_n : \mathtt{Val}_n]), \quad (2)$$

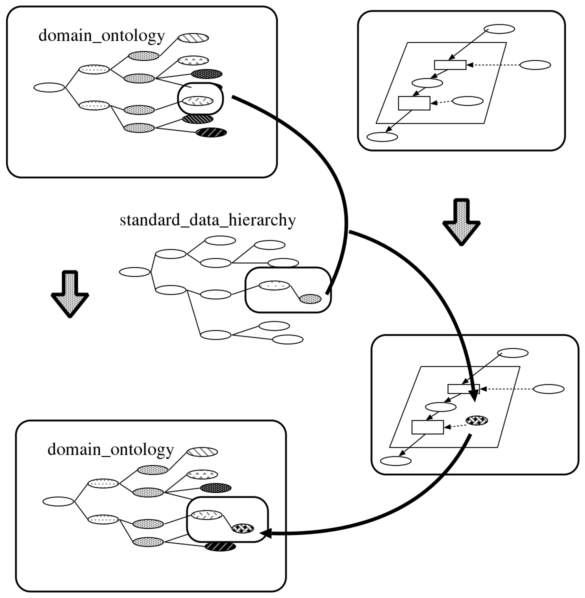


Figure 10: Refinement of knowledge type by using domain ontologies

$$\texttt{list(List\_id, [Atom}_1\texttt{, Atom}_2\texttt{, ..., Atom}_n\texttt{]),} \quad (3)$$
$$\texttt{alias(Alias\_name, Atom\_id\_or\_List\_id)).} \quad (4)$$

In the above formulas, `Atom_id` and `List_id` represent entities consisting knowledge, and `Alias_name` corresponds the name of the entities

A prolog-based description of specification can be refined into the prolog-implementation codes by using REPOSIT library for implementation, as in the same way of the refinement of specification. Figure 11 shows examples of implementation templates for control structures: `for_each` and detailed function: `select_outof_range`. `for_each` , which is re-defined like a built-in predicate `forall` typically appearing in Prolog language, receives a name of a list:`ListID` and executes the programs: `ProgBody` for each `Atom` of `AtomList` corresponding to the name: `ListID`. `select_within_range` , which instanciates a specification template of REPOSIT: `select` (which also appears in canonical functions of Common KADS). By using a method: `pickup` of implementation library, which extracts a value: `I_Val` of the data-id of the corresponding category: `Id@I_Cat`, we can refine `select` into `select_within_range` with call by name. A function body: `is_not_contained( I_Val, R_Intvl)` can also be refined into combinations of method in an implementation library.

As repeating the above procedures, we can get a fine-grained knowledge-flow-diagrams corresponding to a prolog-based description. The method "select" which appears in Figure 9(b) is refined into the diagram in Figure 12.

```
/* REPOSIT: Control structure implementation :
   for_each */
for_each( Atom, ListID, ProgBody) :-
        list( ListID, AtomList),
        member( Atom, AtomList),
        exec_prog(ProgBody),
        fail.
for_each( _Id, _I_DataBody, _ProgBody).
/* REPOSIT: detailed function description:
   select_outof_range */
select( I\_DataName@I\_Cat,
                         @R\_Cat, OutputName) :-
     name2data( I\_DataName, I\_DataBody),
     for\_each( Id, I\_DataBody,
       [
           pickup( Id@I\_Cat, I\_Val),
           pickup( Id@R\_Cat, R\_Intvl),
           is\_not\_contained( I\_Val, R\_Intvl),
           assemble\_data\_z( OutID, Id)
       ]
     ),
     setname2id(OutputName, OutID).
```

Figure 11: Methods in the implementation library

3 Interoperating Distributed Expert Systems

Once expert systems have been modeled and implemented by REPOSIT, the next issue is how to interoperate an expert system with other expert systems to solve problems that cannot be solved alone. In order to modularize such expert system deep interoperation, two issues remains - cooperation(coordination) and communication among expert systems.

3.1 Cooperation for Distributed Expert Systems

3.1.1 Construction of a Common Domain Ontology

In order to exchange information on descriptions of heterogeneous expert systems, a common domain ontology is manually constructed by using the standard data hierarchy and domain ontologies for expert systems. Although it is an important issue how to construct a common domain ontology, we pay much more attention to consider a cooperative method among distributed expert systems, based on a REPOSIT library for specification serving a common method ontology.

3.1.2 Shared Specification

Work made on cooperation to date has been divided into two different approaches, direct communication and assisted coordination in [Gen94]. The former includes the contract-net approach and specification sharing(SS), which are proper in small scale cooperation. The latter includes a facilitators and a mediator, which are better in large scale cooperation. Because

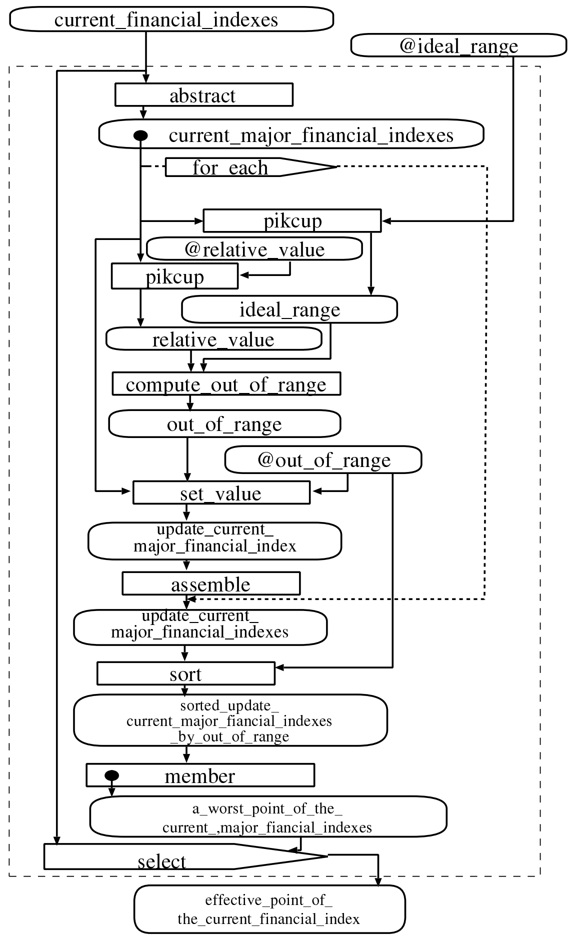


Figure 12: Refined specification of FIMCOES (a part)

we have a common ontology,it is easy to enact an SS-based approach to interoperate expert systems. The shared specification comes from a common method ontology and a common domain ontology. Although one expert system (originator) can get the information about capabilities of the other expert system (recipient) through the shared specification, it is important to identify the information available to (be able to) improve the originator. Because it costs too much to find out differences in extensive range to the whole inference structure, we adopt only the difference arising in the small context of correspondence between inference primitive of an originator and those of a recipient. A method to find out the difference arising is presented (Figure 14) as follows:

1. Make a set of correspondence in which inference primitives are the same between an originator and a recipient.

2. By taking a look at the context (pre-inference-primitive, post-inference primitive, input, output and reference knowledge) of the inference primitive in the correspondence, the correspondence value is computed. The value is computed by comparing the following reference knowledge components: entity,parameter of those, value, dependency relation and criteria for evaluation. The more similar the context, the larger the value.

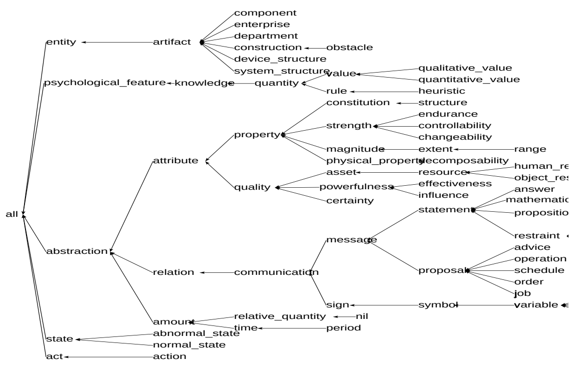


Figure 13: A Common Domain Ontology

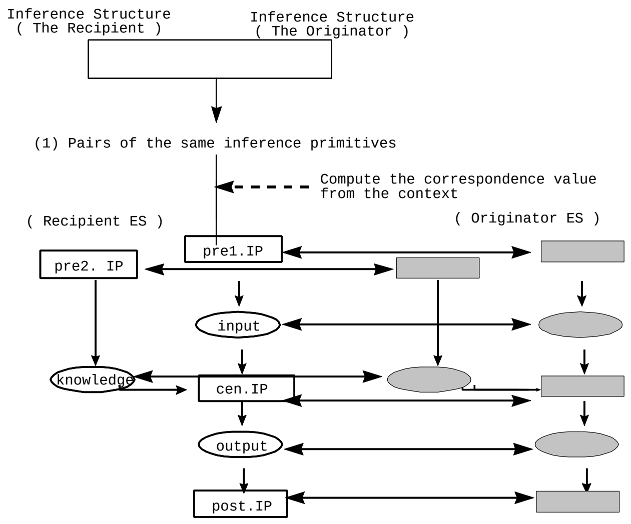


Figure 14: A message generation facility

3. The correspondence value is propagated to pre- and post- inference primitives.

4. After completing propagation over all inference primitives in the correspondence, the difference arising in the context of the correspondence with large values can be used as a reply message to modify the originator's inference engine.

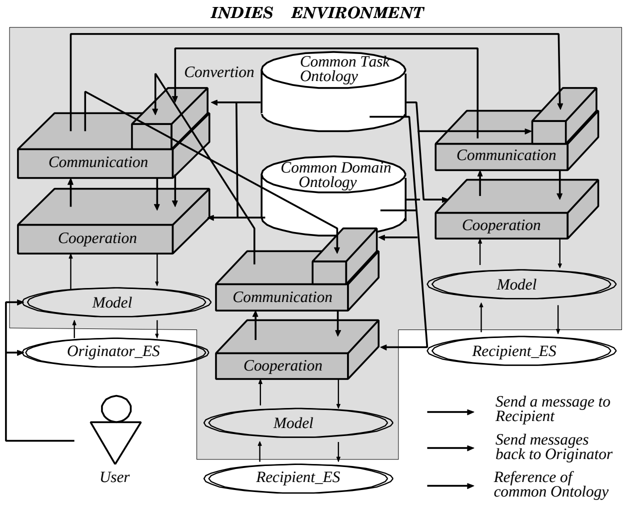
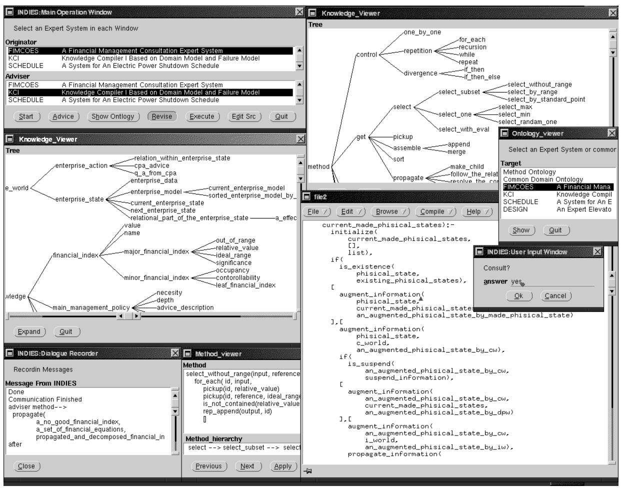


Figure 15: An Overview of INDIES



Select an Expert System in each Window

Originator

FIMCOES	A Financial Management Consultation Expert System
KCI	Knowledge Compiler I Based on Domain Model and Failure Model
SCHEDULE	A System for An Electric Power Shutdown Schedule

Adviser

FIMCOES	A Financial Management Consultation Expert System
KCI	Knowledge Compiler I Based on Domain Model and Failure Model
SCHEDULE	A System for An Electric Power Shutdown Schedule

```
e_world
  enterprise_action
    relation_within_enterprise_state
    cpa_advice
    q_a_from_cpa
  enterprise_state
    enterprise_data
    enterprise_model
      current_enterprise_model
      sorted_enterprise_model_by_
    current_enterprise_state
    next_enterprise_state
    relational_part_of_the_enterprise_state ——a_effec
wledge
  financial_index
    value
    name
    major_financial_index
      out_of_range
      relative_value
      ideal_range
      significance
    minor_financial_index
      occupancy
      contorollability
      leaf_financial_index
  main_management_policy
    necesity
    depth
    advice_description
```

```
method
  control
    one_by_one
    repetition
      for_each
      recursion
      while
      repeat
    divergence
      if_then
      if_then_else
  get
    select
      select_subset
        select_without_range
        select_by_range
        select_by_standard_point
      select_one
        select_max
        select_min
        select_randam_one
      select_with_eval
    pickup
    assemble
      append
      merge
    sort
    propagate
      make_child
      follow_the_relat
      resolve_the_co
```

Select an Expert System or commor

Target

Method Ontology	
Common Domain Ontology	
FIMCOES	A Financial Mana
KCI	Knowledge Compil
SCHEDULE	A System for An E
DESIGN	An Expert Elevato

Consult?

answer yes

Recordin Messages

Message From INDIES

```
Done
Communication Finished
adviser method-->
  propagate(
        a_no_good_financial_index,
        a_set_of_financial_equations,
        propagated_and_decomposed_financial_in
after
```

Method

```
select_without_range(input, reference
   for_each( id, input,
      pickup(id, relative_value)
      pickup(id, reference, ideal_range
      is_not_contained(relative_value
      rep_append(output, id)
      []
```

Method_hierarchy

```
select --> select_subset --> select
```

```
    current_made_phisical_states):-
      initialize(
          current_made_phisical_states,
          [],
          list),
      if(
        is_existence(
            phisical_state,
            existing_phisical_states),
      [
        augment_information(
            phisical_state,
            current_made_phisical_states
            an_augmented_phisical_state_by_made_phisical_state)
      ],[
        augment_information(
            phisical_state,
            c_world,
            an_augmented_phisical_state_by_cw),
        if(
          is_suspend(
              an_augmented_phisical_state_by_cw,
              suspend_information),
        [
          augment_information(
              an_augmented_phisical_state_by_cw,
              current_made_phisical_states,
              an_augmented_phisical_state_by_dpw)
        ],[
          augment_information(
              an_augmented_phisical_state_by_cw,
              i_world,
              an_augmented_phisical_state_by_iw),
          propagate_information(
```

Figure 16: Execution of INDIES

3.1.3 Message Selection

When an originator sends a message to recipients, similarity value is attached with the message. When the recipients can generate the message with values greater than the similarity value, the recipients can send the messages back to the originator. On the other hand, when an originator gets more than one reply message from recipients, the reply message with largest correspondence value is selected.

3.2 Communication between Expert Systems

Because each expert system is modeled by its own vocabulary, it needs a conversion facility so that it can understand the replay messages from other expert systems. This paper calls a wrapper the module to convert one message from one expert system into another message that can be processed in the other expert system. When it is communicated between an originator and a recipient, originator's wrapper uses a common domain ontology to convert the reply messages from recipients.

4 INDIES Design

As the methods of modeling, operationalizing, cooperating and communicating (wrapping) distributed expert systems come up, we put them together into an interoperation environment for distributed expert systems, INDIES as shown in Figure 15. In order for INDIES to be enacted at each development sites, each expert systems should be manually modeled there, using REPOSIT as a common method ontology.

When one expert system finds a fault in itself(for example, when one output was wrong or rejected by a user), it asks INDIES to support it in changing for a better performance. First, the expert system (originator) broadcasts its own model description message to other expert systems (recipients) through communication facilities. After getting the message, the recipient tries to make correspondence between two models from the originator and the recipient, based on SS-based cooperation facilities. The recipient finds differences in the context of good correspondence and also selects the message with higher value than correspondence value requested by the originator, based on message selection facilities. The recipient makes them up into a reply message. Then it sends the message to the originator through communication facilities. The reply message is converted into another reply message using a common domain ontology so that the originator can use it to change itself. The originator selects the message with highest value among received messages, using its own message selection facilities. Finally, the originator tries to reflect the selected message on its own implementation. When the reflection failing, the developers manually change the implementations of the originator based on the selected message. The modified originator's performance is tested. If the same fault still exists, or another fault comes up, the above-mentioned interoperation process is repeated and another reply message is given to the originator until the originator's performance improves or the recipients send no reply message.

INDIES has been implemented by SWI-Prolog ver. 2.9.6 with XPCE ver. 4.9.7(Figure 16). The size of communication and cooperation facilities is 42 KB. The size of models of four expert systems is only 10 KB.

Table 1: Many Differences between the Results of FIMCOES and those of CPA

Advise ID	8	18	50	33	42	36	38	51	13	56	31	40	58
CPA		✓		✓		✓	✓	✓	✓	✓		✓	✓
FIMCOES	✓	✓	✓	✓	✓		✓	✓			✓		

Table 2: The Number of messages generated by interoperation

	Troubleshooting	Scheduling	EV-design	Total
1st	14	14	19	47
2nd	15	15	19	49

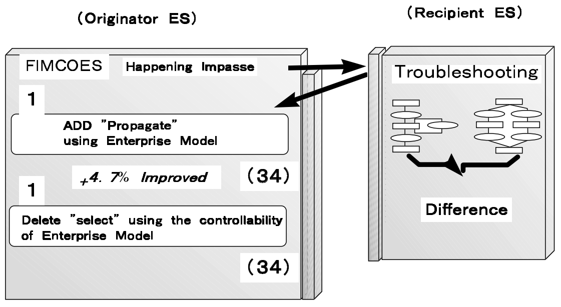


Figure 17: The first interoperation among FIMCOES and other expert systems

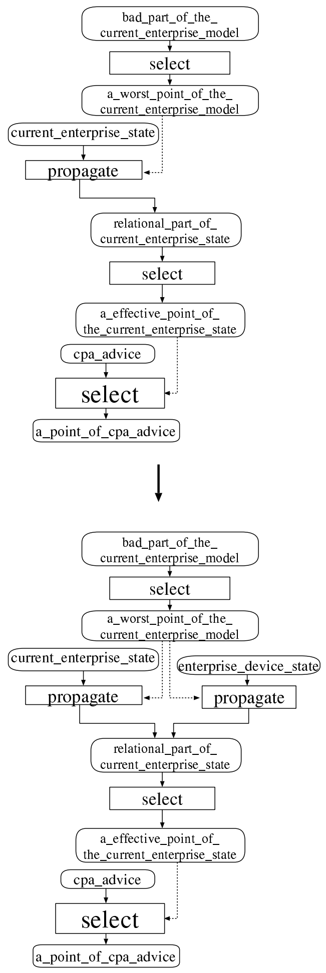

bad_part_of_the_current_enterprise_model
select
a_worst_point_of_the_current_enterprise_model
current_enterprise_state
propagate
relational_part_of_current_enterprise_state
select
a_effective_point_of_the_current_enterprise_state
cpa_advice
select
a_point_of_cpa_advice
bad_part_of_the_current_enterprise_model
select
a_worst_point_of_the_current_enterprise_model
current_enterprise_state
enterprise_device_state
propagate
propagate
relational_part_of_current_enterprise_state
select
a_effective_point_of_the_current_enterprise_state
cpa_advice
select
a_point_of_cpa_advice

Figure 18: Refinement of FIMCOES inference structure

5 Experimental Results and Discussion

Experiments have been done to how the financial management expert system FIMCOES (originator)[Gra94] is supported by a troubleshooting, an electric power management job scheduling and an elevator design expert systems through the interoperation in INDIES. In Table 1, the checks mean the advises recommended by a CPA (Certified Public Accountant) or FIMCOES.

Afterwards, the originator gets into INDIES and sends its models and correspondence value desired (24 in this experiment) to the other three expert systems (recipients) in INDIES. At the first interoperation, the originator received 47 reply messages from other three recipients, as shown in Table 2.

Although the originator got two messages with highest correspondence value from the troubleshooting ES as shown in Figure 17, the reply message of ‘ ADD “Propagate using an enterprise model” ’ has been selected by a simple conflict resolution strategy that ADD has priority over Delete. According to the reply message, the new inference primitive of “propagate using an enterprise model” has been put just before “compare” primitive in FIMCOES inference structure, as shown in Figure 18. The enterprise model is a model of fund flow through six departments and has been given by a user. After implementing a new FIMCOES with the modification, we found that the modified FIMCOES had 4.7% performance gain compared to the original FIMCOES.

At the second interoperation, the originator received 49 reply messages from other three recipients, as shown in Table 2. In spite of that FIMCOES has been modified once, the reply messages with highest correspondence were the same as ones at the first interoperation. Further examination showed that two messages, got in the interoperation except ones got in the first interoperation, brought 11.1% performance gain to FIMCOES but other reply messages not.

6 Related Work

In the field of CDES, much work focuses on strategies of unifying solutions that include uncertainty from multiple expert systems that use different representation of uncertainty. However, few systems try to deal with the management of semantics, at present.

On the other hand, R.Dieng's work [Die94] manages issues in cooperative knowledge-based systems. The specification to interoperate knowledge-based systems from the point of multi-agent systems has been analyzed, but not yet launched into full implementations and evaluation in real task-domains, as shown here.

As issues in software engineering, much work has been done on a framework of reusable data structures[Fow97, Pre95]. At present, little work pays attention to method structure(ontology) and relation between data and methods.

Recently, detailed consideration has been done on visualizing information about conceptual hierarchies, specification descriptions and implementation codes[Jon98, Luk98, RMe98, TMe98]. On the other hand, A lot of frameworks have been proposed including Molina's work on integration of different models of PROTEGE-II and KSM[Mol96]. Comparing those frameworks, our approach is characterized by an integration of specifications and implementations in a unified framework reinforced with ontologies.

As work on consideration about a structure of integrating ontologies, deep investigation about Construction of ontologies has been done[Set98] which argues the necessity of having two different kinds of ontologies on different conceptual levels. It can be a theoretical background of our construction for domain ontologies and a standard data hierarchy.

As compared with our previous work without REPOSIT[YMG98b], more than two weeks are needed as a cost of reflecting received messages in the originator's implementation, while REPOSIT reduces the cost into at most two days.

7 Summary, Discussion and Future Work

In order for distributed expert systems to exchange information at a conceptually acceptable level of details for a better individual performance, REPOSIT has been proposed. Furthermore, a cooperation method based on specification sharing has been presented, which focuses on the difference in context of the correspondence between inference primitives of an originator and those of a recipient. The wrapper with conversion facilities has also been provided, using a common domain ontology manually constructed. After putting them into an integrated deep interoperation environment among distributed expert systems,INDIES, the experiment results have shown that the interoperation works on the four heterogeneous expert systems. Message selection facilities are still static and so should change into dynamic ones, using inductive learning techniques. Furthermore, we will pay much more attention to automatic construction of a common domain ontology.

References

[Bre94] J.Breuker and W.Van de Velde. Common KADS Library for Expertise Modeling, IOS Press (1994).

[Die94] Rose Dieng: Agent-Based Method for Building a Cooperative Knowledge-Based System, Workshop on Heterogeneous Cooperative Knowledge-Bases.International Symposium on Fifth Generation Computer Systems (1994) 237-251.

[Fow97] M.Fowler, "Analysis Patterns: Peusable Object Models", Addison-Wesley, 1997.

[Gra94] Garcia, P. V. D., Yamaguchi, T.: A Financial Management Consultation Expert System with Constraint Satisfaction a nd Knowledge Refinement. The Third Pacific Rim International Conference on Artificial Intelligence. (199 4) 979-985

[Gen94] M.R.Genesereth and S.P.Ketchpcl: Software Agents, CACM.ol.37.No.7. (1994) 48–53.

[Jon98] C.Jonker, R.Kremer, P.van Leeuwen, D.Pan, J.Treur, "Mapping Visual to Textual Representation of Knowledge in DESIRE" Proc. 11th Knowledge Acquisition for Knowledge-Based Systems Workshop, 1998

[Luk98] D.Lukose, G.W.Mineau, "A Comparative Study of Dynamic Conceptual Graphs" Proc. 11th Knowledge Acquisition for Knowledge-Based Systems Workshop, 1998

[RMe98] R.A.F.Mendez, P.van Leeuwen, D.Lukose, "Modeling Expertise Using KADS and MODEL-ECS" Proc. 11th Knowledge Acquisition for Knowledge-Based Systems Workshop, 1998

[TMe98] T.Menzies, "Evaluation Issues for Visual Programming Languages" Proc. 11th Knowledge Acquisition for Knowledge-Based Systems Workshop, 1998

[Mol96] M.Molina, Y.Shahar, J.Cuena, M.Musen, "A Structure of Problem-Solving Methods for Real-time Decision Support: Modeling Approaches Using PROTEGE-II and KSM" Proc. 10th Knowledge Acquisition for Knowledge-Based Systems Workshop, 1996

[Pre95] W.Pree, "Design Patterns for Object–Oriented Software Development", ACM-Press, 1995.

[Set98] K.Seta, M.Ikeda, O.Lakusho, R.Mizoguchi: "Construction of a Problem Solving Ontology – A Scheduling Task Ontology as an Example –" Japanese Society of Artificial Intelligence.Vol.13.No.7. (1998) 597-608 .(in Japanese)

[YMG98b] T.Yamaguchi: DESIRE: An Interoperative Environment for Distributed Expert Systems. ECAI'98 Wrokshop on Applications of Ontologies and Problem-Solving Methods. (1998) 120-125.