Heterogeneous Distributed Cooperative Problem Solving System HELIOS

Akira Aiba, Kazumasa Yokota, and Hiroshi Tsuda
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
{aiba, kyokota, tsuda}@icot.or.jp

Abstract

This paper describes a heterogeneous distributed cooperative problem solving system HELIOS that is being developed at ICOT.

HELIOS is a framework for constructing a heterogeneous distributed cooperative problem solving system having those three kinds of flexibilities. Each problem solver is encapsulated by a module called a *capsule* to absorb heterogeneity on types of data or knowledge that are used to communicate with other problem solver. Encapsulated problem solvers, we call them *agents*, are placed in a common space called an *environment*. Through an environment, agents can communicate by sending messages to each other. An environment provides global information such as a common data type system, common message protocol, and global constraints, for agents in it. Furthermore, even if agents are placed on more than one machines, the environment takes care of communication between them.

To verify our idea, we implemented several versions of HELIOS on distributed environment. The HELIOS system will be released as ICOT Free Software by the end of March 1995.

1 Introduction

As on knowledge information processing applications become more advanced and complex, problems becomes increasing hard to represent and solve using just a single problem solver such as a database, a constraint solver, or an application program. Any system for the advanced, complex and vat problems of knowledge information processing must take into account model heterogeneity, spatial heterogeneity and temporal heterogeneity.

Problems in those areas are hard to model using a single paradigm, and hard to represent with single knowledge representation. This is model heterogeneity. Huge systems such as databases are hard to migrate or transport to other machines because of their sizes. So these systems must be used on the machine on which they are currently running. This is spatial heterogeneity. Repeated expansion or modification of existing languages, or the never-ending progress for new programming languages causes perpetual rewriting of software to use new programming language. This is temporal heterogeneity. Thus, a system for solving those advanced, complex and vast problems should have three kinds of flexibilities for handling those three kinds of heterogeneity. The point is to integrate various application programs, databases, or constraint solvers effectively.

At ICOT, we have developed many knowledge representation languages, constraint solvers, database management systems (DBMS), and application programs throughout the FGCS project and its follow-on project. These include knowledge representation language called *Quixote* [Yokota *et al.* 1993], parallel constraint logic programming language called GDCC [Terasaki et al. 92], database management system called Kappa [Kawamura *et al.* 1992], and applications on genetic information processing and legal reasoning [Nitta *et al.* 1994]. In order to spread these applications on knowledge information processing, it is crucial that we provide a framework for combining and cooperating various knowledge representation languages and problem solvers.

The heterogeneous distributed cooperative problem solving system HELIOS is intend as a way to achieve flexible and advanced problem solving.

To achieve this, the important issue is to absorb heterogeneity between various knowledge representation languages and problem solvers and to add functionality for cooperative problem solving. For this purpose, each problem solver is encapsulated to enable it to communicate by messages with other problem solvers by translating its internal data type of message content into a common data type. To integrate global information for agents that are communicating with each other, a common space is introduced with provides a common data type, a common message protocol, and a message delivering functions. With this common space, separate problem solvers can be combined and cooperate with each

other even when these problem solvers are implemented on different languages, data types in a distributed environment.

Next, a problem solving system can be constructed in a bottom-up manner. That is, we were able to construct a problem solving system by combining several problem solvers in a common space. In HELIOS, the combined problem solving system comprising a number of problem solvers with a common space, can be treated as an independent a problem solver again. Thus, it can be combined with other problem solvers in a larger common space. This means that a problem solving system in HELIOS can be hierarchical.

These form the basic model for HELIOS. Based on it, in early 1994 we first implemented an experimental version of HELIOS on a UNIX workstation to check communications between problem solvers. We implemented the first version of HELIOS with certain encapsulation functionalities in the mid-summer of 1994 on UNIX workstations connected by an Ethernet. We implemented the second and the latest version of HELIOS in November 1994 on UNIX workstations connected by an Ethernet.

The structure of this paper is as follows. In section 2, we describe the basic model for HELIOS. In section 3, we describe a distributed implementation of the second version of HELIOS with a small example.

2 Logical Model for HELIOS

2.1 Basic Model

To realize a heterogeneous distributed cooperative problem solving system with model heterogeneity, spatial heterogeneity, and temporal heterogeneity, multiple problem solvers are combined as follows.

In general, each problem solver utilizes various data type systems. For establishing communication between those problem solvers, the contents of communication should be mutually understandable.

In HELIOS, communication between problem solvers is performed in restricted form. Each problem solver exports problems that it cannot solve. If the answers to the problems are obtained from some other problem solver, they are sent back to the original problem solver. That is, communication in HELIOS is either querying or answering. This communication is done by message passing between problem solvers. To preserve flexibility, names of problem solvers that can answer certain queries should not be fixed. That is, a problem solver that can answer a query can be replaced by another problem solver with the same problem solving ability with different data type without changing the other problem solvers. Thus, a common message protocol, common data types of contents of messages are required. Therefore, each problem solver must have a module for translating between common data types, and its own local data types.

A module contains a translator called a *capsule*, and each problem solver is enclosed in a capsule. An encapsulated problem solver is called an *agent*, and a problem solver is called a *substance*. Each agent has its own logical name that is unique in the environment.

A common space for agents is called an *environment*. An environment takes care of message passing between agents in it, and manages global information for those agents. Global information contains a common data type system. An environment with agents in it can be considered as a problem solver again, and so it can be encapsulated and placed in a larger environment. This type of agent is called a *complex agent*, while an agent having no internal structure is called a *simple agent* (Figure 1).

An environment-agents structure can be nested in HELIOS. This allows us the following:

- Bottom up construction of agents
- Possibility of expansion and openness of problem solvers
- Classification of agents
- Reuse of agents

2.2 Message Protocol

A message between agents contains the followings:

- **Message identifier**
  An identifier used for identifying a message. This field is unique within an environment.
- **Message type**
  As described in the previous section, a message is either a method invocation or an answer. The former message is called a *query message*, and the latter message is called a *reply message*. This field is used to distinguish a query message from a reply message.
- **Sender agent identifier**
  This field contains a logical name of the agent that sends this message.
- **Designation of destination agents**
  The methods of designating destination agents in a query message are described in Section 2.4. In a reply message, this field contains the logical name of the agent that is the sender of the corresponding query message.
- **Transaction identifier**
  If update of a content of a destination problem solver is attendant on invocation of a message, then a transaction identifier is required to control it. This field contains a transaction identifier. For nested transaction, a transaction identifier with a nested structure is used.
- **Status**
  This field contains information on the status of invoked methods for error handling.
- **Message content**
  In a query message, this field contains a method in-

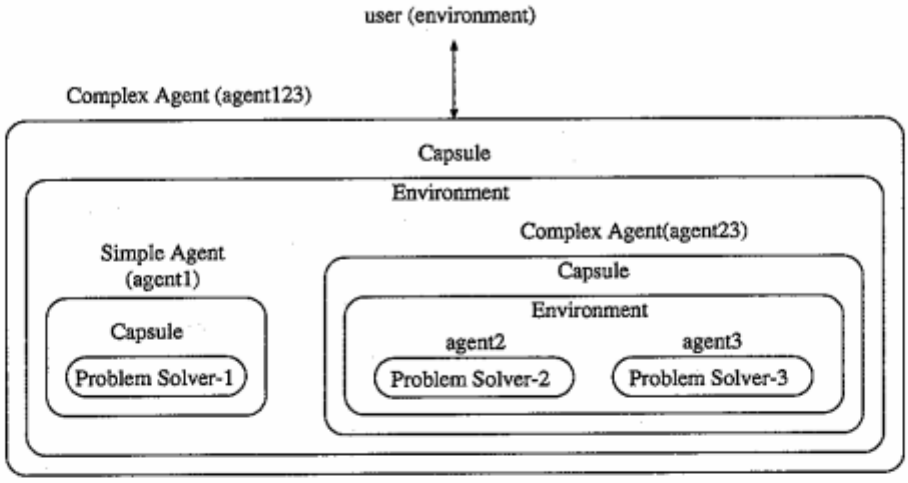

Figure 1: Basic Model for HELIOS

vocation, and in a reply message, this field contains the answer to the invocation.

2.3 Agent

An agent has the following two functions: solving problems that are asked by other agents, and asking problems that cannot be solved internally to other agents. An agent with only the former function is called a *passive agent*, while an agent with both functions is called an *active agent* (Figure 2).

To make an agent active, a substance should have the following functions:

- The ability to accept a problem including sub-problems that cannot be solved in it,
- The ability to transmit those sub-problems to other agents, and to receive answers from those agents, and
- The ability to accept and interpret answers.

Note that to realize cooperation between agents, those agents should be active.

The role of an agent capsule is to wrap a problem solver to make it an agent.

In a capsule, the following are defined:

- **the name of the agent**
  The name of the agent is the logical name of the agent.
- **import methods/export methods**
  Each agent provides methods to the public. In a capsule, methods that the agent make public to other agents, and methods that the agent may invoke are declared. The former methods are called *import methods*, while the later methods are called *export methods*. That is, messages between agents are restricted to invocations of export methods and answers to those invocations. Note that by declaring both methods, translation is restricted to parameters of methods and their answers.
- **self model**
  An environment uses various information on agents in it for delivering messages. In each agent capsule, functions provided by the agent is defined in terms

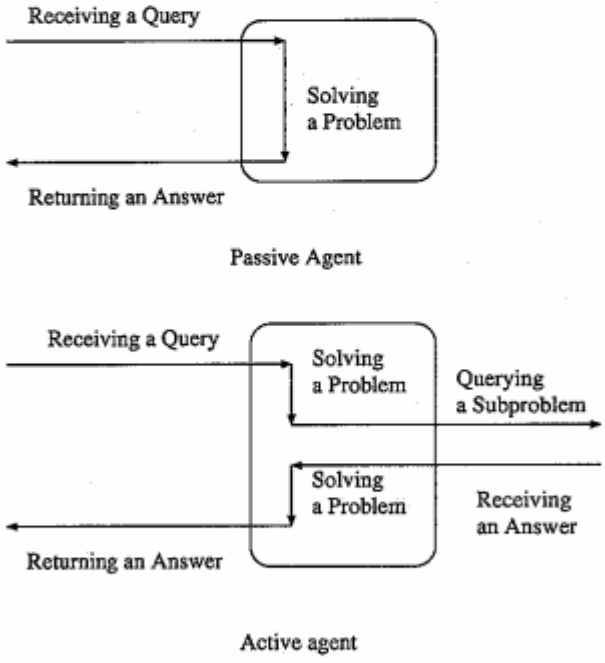

Figure 2: Passive Agent and Active Agent

of a *self model*. The use of functions in the self model is described in the section 2.4.

- **concurrency control for the substance**
  If a substance has the concurrency control function, then the capsule can send all messages as capsule receives. However, if a substance has no concurrency control function, then the capsule should lock/unlock the substance, or make copies of substances to control concurrent execution of the substance.
- **translation rules**
  As described in Section 2.1, translation between common type system and own type system defines translation of types.
- **negotiation strategy**
  Besides these definitions, a capsule should determine the strategy for negotiating with other agents. To make agents mutually negotiable, those agents should be active, and a negotiation protocol should be defined. A negotiation protocol is defined in an environment. A negotiation strategy defines what message should be sent when the agent receives a certain message according to the negotiation protocol. The relationship between a negotiation strategy and a negotiation protocol are described in Section 2.4. The export methods in the agent and the export methods of other agents are used to construct message parameters.
- **others**
  Except for the definitions listed in the above, the following are defined in the capsule description language called *CAPL* (CAPsule Language).
  The method used to connect a capsule with a substance is defined in the CAPL program. There are two relationship between a capsule and a substance: a substance and a capsule executed in the same process, and a substance and a capsule executed in distinct processes. This categorization is defined in a capsule. In the former case, the name of the file that contains the substance to be linked with a capsule is defined, and in the latter case, the name of the executable file that is connected with a capsule is defined.

2.4 Environment

The roles of an environment are handling messages between agents, and managing global information for agents. Global information can be categorized as follows. Certain global information is used to make an agent enable to communicate: common data type, and ontology are included in this category. Then, certain other information is used to control the destination of messages. Directories in an agent server are included in this category. Other global information is used to control agent behavior. Negotiation protocol for agents, and global constraints are included in this category. This global information is described below.

Handling of messages

All messages between agents are strongly typed by a common type system in the environment. An environment may receive a message in two ways: one is from an agent in it, and the other is from a capsule of that environment. That is, one is from inside the environment and the other is from the outside. In both cases, the contents of the message is represented using the common type system in both environments. Remark that the content of the message from the outside is converted by the capsule of that environment.

Then the environment attempt to find agents that the message should be delivered to by referring to the destination field of the message. If the environment can find agents that the message should be delivered to, then the environment sends the message to them. But if the environment cannot find those agents, then the environment sends the message to the outside through its capsule.

Global information for communication

Common data type is used to represent the contents of messages understandable to all agents in an environment. Thus, common data type is defined in an environment.

The vocabulary used in the contents of a message is also converted, if necessary, using on ontology that is defined in that environment.

Global information for message control

In a message, the destination agents should be cleared. The simplest way of designating destination is clearly using a logical name of an agent. However, this method assumes the existence of the agent with that logical name in the same environment. Thus, although this way is effective, it gives little flexibility for combining agents in an environment.

In HELIOS, three methods are prepared to designate the destination of a message: a method to designate the destination by a *logical name*, a method to designate the destinations by *agents' functionalities*, and a method to designate the destinations by a *pair of agents' functionalities and a method name*.

Let us recall the primary motivation for requesting to problems from other agents. Utilizing functionality provided by other agents is an important issue, and the logical names of other agents are not a issue. That is, the designation of destination agents by their function makes possible communication between agent without knowing their names, places, etc.

Figure 3 shows modules in an environment.

Control messages in an environment are carried out by a module called the *agent server*.

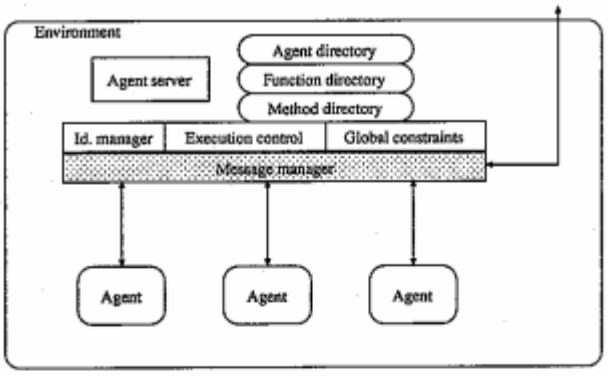

Figure 3: Modules in an environment

To implement the above described control methods, the agent server utilizes the following three directories.

- Agent Directory
- Function Directory
- Method Directory

An *agent directory* is a table whose entry is the pair comprising a logical name of an agent and its physical address. This directory is used to deliver a message in which the destination is designated by the logical name of an agent.

A *function directory* is a table whose entry is a pair comprising a function and a list of logical names of agents that have that function. For instance, if agents A_1, A_2, and A_3 have the same function *Addition* in an environment, then the function directory in the environment has an entry consisting of the function *Addition* and a list of A_1, A_2, and A_3. Ontology is used to select words used to represent functions. A function directory is constructed by gathering the self model of each agent.

A *method directory* is a table whose entry is a pair comprising a function and method name, and a list of a pair comprising a logical name of an agent and a method.

The simplest and the most naive method to designate a destination, by a logical name, is handled by an agent server using an agent directory. Using the directory, an environment gets information on the physical address of a destination agent, then the environment can send a message to it.

When a destination is designated using agent functions, then more than two agents with the same functions will all be designated. For instance, agents A_1, A_2, and A_3 are selected when the function *Addition* is designated. In such cases, collected answers from those agents are processes using aggregate functions to construct an answer. Aggregate functions includes bringing together all answers into one list, similar to bag_of in Prolog, and selects all answers that satisfy global constraints etc.

Global information for control agents

A global constraint can be used not only to aggregate functions, but also to control agents. When a problem solving system solves a problem like a board game or a maze, the board or the maze restricts the behavior of agents. In this sense, the board and the maze controls the agents. Those boards or mazes are represented using global constraints in an environment.

A negotiation protocol between agents and negotiation strategies for agents are required to allow negotiation between agents. As we describe in Section 2.3, the negotiation strategy for each agent is defined in the capsule. The negotiation protocol is defined in the environment.

There are so many protocols for negotiations. Syntax, number of parameters, and keywords to define the type of message are included in the definition of a negotiation protocol. Let us take the contract net protocol as an example. A contract net protocol consists of at least three types of messages: messages used for announcement of tasks, those used for bidding, and those used for awarding. Thus, "announcement", "bidding", and "award" would be keywords to distinguish these types of messages from each other. Those message cannot be sent in arbitrary situations. That is, a "bidding" message cannot be used before "announcement", and "award" cannot be used before "bidding", etc. These restrictions can be represented in terms of a state transition. These restrictions are also included in the definition of a negotiation protocol.

On the other hand, a capsule has to select one message even if the selected negotiation protocol gives the restrictions described above. In many cases, several messages can be send in a certain situations according to the negotiation protocol definition. Selection from these alternatives is done by the negotiation strategy.

Since there may be more than one negotiation protocols in an environment, more than one negotiation protocols can be defined in an environment and distinguished by a *negotiation protocol identifier*. To participate in a negotiation, agents should declare the negotiation protocol to be used by the negotiation protocol identifier,

If agents participating in a negotiation declare different negotiation protocol identifiers, the negotiation fails. This is the user's responsibility. Since an environment has negotiation protocols, it can check whether messages between agents use the selected protocol or not. Thus, messages that do not fit the protocol can be filtered out by the environment. Selecting an appropriate negotiation protocol and using the corresponding negotiation strategy in all the participating agents are also the user's responsibility.

The environment description language called *ENVL* (ENVironment Language) is used to define the environment.

3 Distributed Implementation of HELIOS

3.1 HELIOS System

Implementation of HELIOS started in 1993. At the beginning of 1994, we first implemented an experimental version of HELIOS to check communications between agents on a UNIX workstation using Prolog. We then implemented the first version of HELIOS with certain functionalities of capsules and environments in the midsummer of 1994 on UNIX workstations connected by an Ethernet. In November 1994, we implemented the second version of HELIOS with the CAPL and ENVL compiler with restrictions on functions on UNIX workstations connected by an Ethernet. The main focus of the second version is implementation of the environment, agents and their hierarchical structures, to avoid environment communication bottlenecks.

In the following sections, we describe our implementation for the second version based on agent processes, messages passing between agent processes, and examples of the CAPL and ENVL programs.

In CAPL and ENVL in the second version, some functionalities listed in the logical model are restricted as follows. In CAPL in the second version, the following restrictions are placed on functions. First of all, the self model of an agent is restricted. In the logical model, the self model is used for constructing directories for the agent server. This is not implemented in CAPL in the second version. Directories for the agent server are defined manually in the ENVL in the second version. For concurrency control for the substance, CAPL in the second version provides only a lock mechanism. The negotiation strategy can be described in CAPL in the second version experimentally. That is, the negotiation protocol is defined in CAPL with the definition of negotiation strategy in terms of a set of rules.

On the other hand, the following restrictions on functions are placed in the ENVL of the second version. As we described above, the function directory and the method directory in each agent server are created manually in ENVL in the second version. The negotiation protocol is not described in ENVL in the second version.

3.2 Agent Processes

Implementing a naive logical model for HELIOS described in the previous section is problematic in terms of execution efficiency. That is,

- difficulty of concentrating processes to environments,
- mapping from the hierarchical structure of environment and agents to processes, and
- mapping to a distributed computation environment

The first issue is solved by copying the environment. The second issue is solved by implementing mapping, thereby decreasing the number of processes. The third issue is solved by implementing distributed management for agents and process information. From our experience with constructing the very first experimental HELIOS system, we concentrated on the second issue.

The central concept of the distributed implementation of HELIOS is that each simple agent is implemented by a distinct process with all environments that involve the simple agent (Figure 4).

This figure corresponds to Figure 1. In general there are three kinds of processes.

3.3 Message Passing between Agent Processes

When an agent transmits a query message, it passes a message to an environment that is in the same process. An environment also passes a message to an outside environment in the same process. When an environment transmits a message to an agent that is in the environment, then the environment passes it to a process that includes the destination agent. When the agent that receives the message is a complex agent, then there are more than two processes representing the complex agent. Thus, for a complex agent, a *representative process* is selected, and the environment passes a message to a representative process.

In Figure 4, suppose that *Agent Process-2* is the representative process for a complex agent 23. When agent 1 transmits a query message to agent 3, then *Capsule-1* passes the message to *Environment 123* in *Agent Process-1*. Then, Agent Process-1 transmits it to *Capsule 23* in *Agent Process 2*. Then *Environment 23* in Agent Process 2 transmits it to *Capsule 3* in *Agent Process-3*. Then finally, *Problem Solver-3* in the problem solver process can get the message by means of communication between Capsule-3 and the process.

We implemented an experimental system for HELIOS on a distributed network of UNIX workstations.

As shown in Figure 4, HELIOS consists of a *Daemon process*, and a *User Interface Process* apart from the agent processes. Each process is implemented as a UNIX process, and communication between processes occurs as inter-process communication using sockets.

- **Daemon Process**
  There is a daemon process on each machine in the distributed environment for HELIOS. The daemon process manages the creation and deletion of agent processes, and provides location information on the agent processes. In the distributed environment, communication between daemon processes is required.

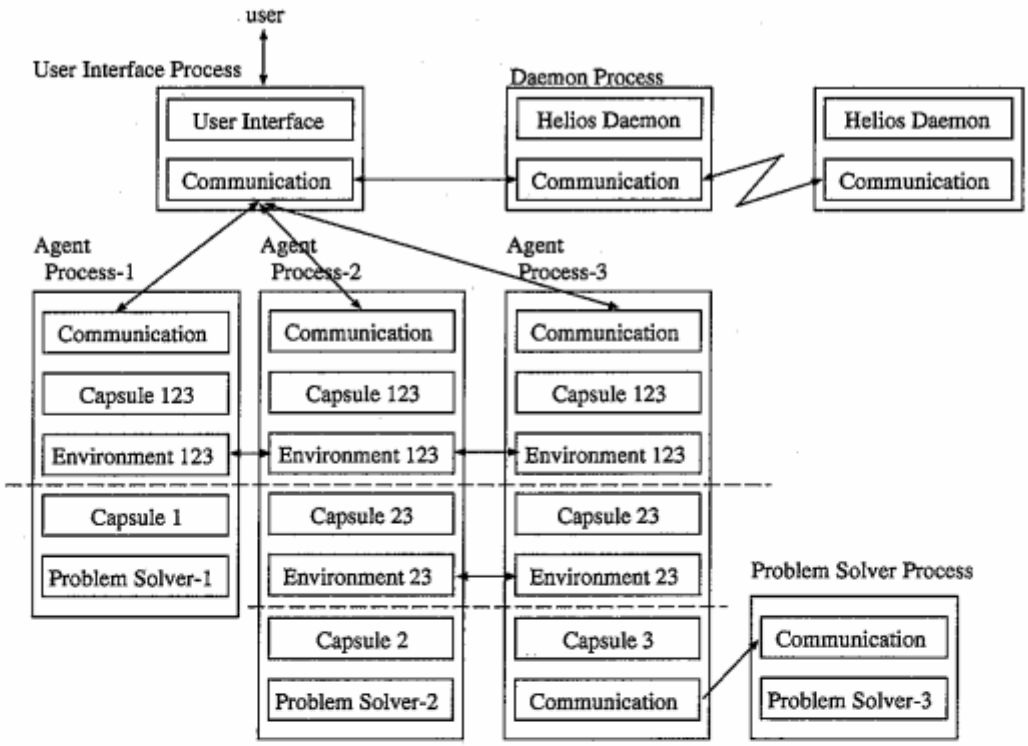

Figure 4: Implementation Model for HELIOS

- **User Interface Process**
  An interface process is prepared for each user to
  - establish communication between the user and the daemon process (*Daemon Session*),
  - establish communication between the user and the agent process (*Agent Session*).

  In a daemon session, a list of agents, and precise information on designated agents are presented, and connection to an agent takes place. In an agent session, messages are sent to the selected agent, precise information on the agent is presented, and the connection to the agent is cut.

Apart from those processes, when a problem solver is a single process separated from the process which implements its capsule, it is called a *Problem Solver Process* (see Problem Solver-3 in Figure 4).

To map a logical model to this implementation model, the following should be considered.

- A single agent in the logical model is sometimes divided into more than one processes in the implementation. Thus, one process should be selected to send messages. This can be done by scheduling which takes account of loads of processes, or dispatch which takes account of message contents. Currently, the representative process is used when the agent is started up for ease of implementation.
- In the logical model, a route of a message is uniquely determined except for nondeterministic message processing. In the implementation, however, the route to a problem solver passes through the representative process, but the route from the problem solver tends to remain on the same process. Thus, interprocess communication decreases because of this route asymmetry.

Figure 5 shows what files should be prepared to use HELIOS, and how agent processes are constructed from those files.

In Figure 5, `*.cpl` is a file for a CAPL program, `*.cty` is a file for common type declaration, and `*.evl` is a file for an ENVL program. CAPL programs are compiled by the CAPL compiler into C programs. Then those programs are compiled by the C compiler into object programs. ENVL programs, with reference to files containing common type definitions, are compiled by the ENVL compiler into C programs. Then those programs are also compiled by the C compiler into object programs. On the other hand, a file with an extension `hia`, called an *hia file* which specifies hierarchies of agent processes, is automatically generated from ENVL programs and CAPL programs. This file contains the physical address of all

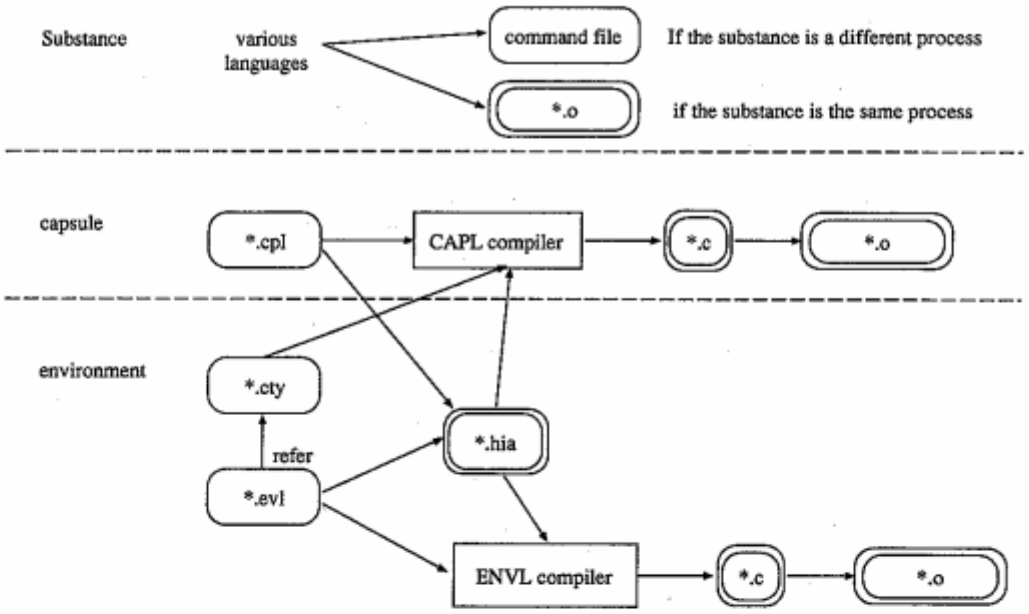

Figure 5: File Configuration of HELIOS

agents, and the command name for invoking processes for those agents generated from those object files.

3.4 CAPL and ENVL

In this section, CAPL programs, a single example of an ENVL program of HELIOS version two are presented to explain the functions.

Simple Scheduling Problem

The problem is to have M meetings in a group consisting of N members. Each meeting has planned participants, and gives a point to each of them. On the other hand, each member has their own priority for participating in each of the meetings. As far as possible, each member's preference should be respected.

That is, each meeting should include as many of those planned participants to acquire points as much as possible, while each member aims at participating in the most preferable meeting to acquire points as much as possible.

The aim of the system is to maximize the sum of all points acquired by all meetings and all members.

Agents

To solve the problem, we use two kinds of agents: meeting agents and member agents.

Each meeting is defined as an agent, and we name it a *meeting agent*. In each meeting agent, a point is given to a member if it is planned that they attend that meeting. Thus, each meeting agent has at most N different points since there are N participants. If the meeting agent succeeds in collecting a participant, then the meeting agent acquires the points of that member. The aim of each meeting agent is to collect members so as to maximize acquired points.

Each member in the group is defined as an agent, and we call it a *member agent*. A member agent cannot participate in more than one meeting if they are held simultaneously. Each member agent also has their own priority for participating in each of the meetings, but if a meeting designates the member, then the member should participate in the meeting.

Problem Solving in HELIOS

The following is an outline of the solution to the problem. First of all, each meeting agent collects participants by offering the meeting to all of the planned participants. The collected participants are called initial participants. If a member agent receives more than one invitation to meetings, then the agent selects the one that has the highest priority. After collecting initial participants, each meeting agent calculates the point acquired.

To maximize the sum of points acquired by meeting agents, each meeting agent gets one of participants of other meetings each other. First of all, let us suppose that all meeting agents are given global ordering. The first meeting agent makes plans to increase its own acquired points. Then, those plans with estimated increase of points are sent to other meeting agents having desired participants one by one. When a meeting agent receives a plan, then it calculate decrease of points and add it to

the increase given with a plan. If the sum is positive, then the plan is accepted, otherwise, the plan is rejected. Next, the second meeting agent can make and send plans, then third, etc. When no participants cannot move for increasing the sum of points, then the whole system gets the maximum points.

The configuration of agents and environments for solving this problem is represented in Figure 6.

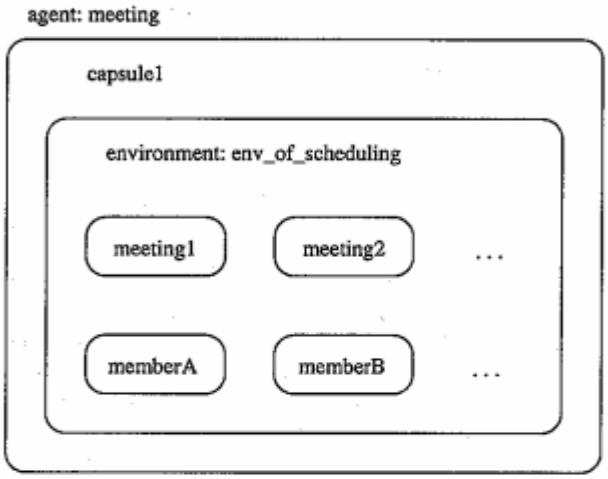

Figure 6: Agents and environments for solving scheduling problem

Figure 7 is a fragment of an ENVL program for the environment env_of_scheduling.

```
1    &environment  env_of_scheduling;
2       &common_type   scheduling.cty;
3       &agent_dir     meeting1, meeting2, ...
4                       memberA, memberB, ...
```

Figure 7: ENVL program for an environment env_of_scheduling

In Figure 7, numbers attached to lines are line identifiers.

Line 1 is the definition of the name of this environment. In this problem, the name of the environment is `env_of_scheduling`. In line 2, the common type is defined. In this case, a file named `scheduling` is defined. Files defining common types have the extension `cty`. The content of the file `scheduling.cty` is given in Figure 8.

In the definition, `string`, and `int` are primitive types for common types given a priori.

The third and the fourth line in Figure 7 define agents that are used in this environment. The agent directory is constructed from this definition.

```
meeting  ::=     string;
 member  ::=     string;
   date  ::=  <  month=int,
                 date=int,
                 start=time,
                 finish=time >;
   time  ::=  <  hour=int,
                 minute=int >;
```

Figure 8: Common type definition in scheduling.cty

On the other hand, an example of the CAPL description to describe the problem includes the following. Figure 9 is a fragment of the CAPL program for a capsule of the environment env_of_scheduling shown as capsule1 in Figure 6.

```
1    &type complex;
2    &agent meeting;
3    &inside env_of_scheduling meeting.evl;
4    &import_method
5     scheduling  #1:[meeting] ->
6      [<meeting=#2:meeting,
7       date=#3:date,
8       member=#4:[member]>]
9  => all, bag_of !  schedule #1:[meeting] ->
10      <#2:meeting,
11       #3:date,
12       #4:[member]>;
```

Figure 9: CAPL program for capsule1

Line 1 defines the type of the content of this agent. In this case, the keyword `complex` declares that this agent is a complex agent. In line 2, the name of this agent `meeting` is defined. Line 3 is the definition of the content of this agent. Since this agent is a complex agent, the content is an environment named `env_of_scheduleing` as described in the above ENVL fragment. The ENVL program for env_of_scheduling is stored in a file named `meeting.evl`. `evl` is the extension given to the ENVL program. Lines 4 through 12 are used to define import method for this complex agent. In the above definition, a single import method named `scheduling` is defined. Let us describe the method definition more precisely.

`scheduling` is the name of this import method with an input parameter and an output parameter. An input parameter identified as `#1` is of the type "list of `meeting`". Remark that the user is the outermost environment, the user should have its own common type. This is given by the user model. For instance, list is a data type defined using a list constructor in the common data type for the user. `meeting` is also a common data type for the user.

An output parameter is a list of tuples consisting of three elements identified by #2, #3, and #4. A tuple is also a data type defined using a tuple constructor in the common data type for the user, and `date` and `member` are also common data types for the user.

This import method definition is translated into a method invocation in the inside environment `env_of_scheduling`. In this case, method `scheduling` is translated into method invocation of the `schedule` of certain agents in the environment. `all` means that messages including this method invocation should be delivered to all the agents in the environment, and the next `bag_of` means that answers from all of those agents should be collected by the aggregate function `bag_of`. The input parameter for the method `schedule` is also a list of `meetings`, and the output parameter is a tuple of `meetings`, `dates`, and a list of `members`. To make the description simple, the common data type for the user and that in the environment `env_of_scheduling` are identical.

Besides the CAPL program for `capsule1`, CAPL programs for meeting agents and member agents are also required.

The following figure (Figure 10) is a fragment of a CAPL program for a meeting agent `meeting1`.

```
1    &type simple;
2    &agent meeting1;
3    &parameter threshold:int = 80,
4    &inside prolog -l meeting.pl;
5    &substance_type prolog;
6    &connect pipe;
7    &lock always on;
8    &import_method
     ...
9    &export_method
     ...
```

Figure 10: CAPL program for meeting1

In the definition in Figure 10, line 1 defines that this agent is a simple agent. The definition with keywords `&agent` is the same as that for capsule1. Information stored for local usage is defined with the keyword `&parameter`. In this case, the threshold of the meeting1, 80 is defined. The keyword `&inside` defines that the substance of the agent is a prolog program, that the program is stored in the file named `meeting.pl`, and that the substance is invoked by a command `prolog -l meeting.pl`. `&substance_type prolog` is used to determine the substance type definition. Line 6 determines the method used to connect the capsule and the substance. In this case, a pipe is used. In line 7, the declaration `&lock always on` states that all messages received by the agent should be serialized before passing them to the substance, and turns on the transaction control. That is, if a side-effect occurs in the substance and the substance cannot manage the transaction because the implementation language for the substance is prolog, then the capsule should manage the transaction control. As in the CAPL program for capsule1, import methods and export methods are defined.

4 Concluding Remarks

In this paper, we describe the heterogeneous distributed cooperative problem solving system HELIOS developed at ICOT during the Follow-on Project. To verify our idea on HELIOS, we have implemented several versions of HELIOS since 1993, and the final version will be released as ICOT Free Software by the end of March 1995.

We are now trying to use HELIOS for natural language processing applications. In [Tsuda 1994b], HELIOS is applied to natural language processing using negotiating agents. Natural language understanding includes two kinds of heterogeneity: a variety of constraint domains, and a variety of natural language processes. Constraint-based grammars such as HPSG and JPSG are examples of the former, where understanding corresponds to constraints solving in heterogeneous domains of constraints such as unification [Tsuda 1994a], temporal logic, subsumption relations and so on. HELIOS implements a negotiation-based distributed model to natural language phenomena such as garden-path sentence recognition, syntactic/semantic interaction [Marcus 1980], ill-formed sentences, and disambiguation, all examples of the latter.

Our research and development provided a comprehensive testbed for heterogeneous distributed cooperative problem solving for dealing with three types of heterogeneity: model heterogeneity, spatial heterogeneity, and temporal heterogeneity. By making capsules and environments programmable by CAPL and ENVL, HELIOS provides a flexible and extensible common space and communications. The resulting system uses HELIOS as a vast system for knowledge information processing, tackling the issues of sharing knowledge and reusability of knowledge/programs. Existing programs can be used by making it an agent, and this is required for temporal heterogeneity. Introducing capsule and environment is required for model and spatial heterogeneity. Cooperation is required for resource bound environment. When vast systems for knowledge information processing are considered, considering the above three kinds of heterogeneity and resource bound environment is crucial.

HELIOS has many aspects: it can, for example, be seen as a multi database, and a multi agent system.

To construct a system including autonomous heterogeneous databases under a distributed environment, the relations between databases are often classified as follows [Manola *et al.* 1992, Papazoglou *et al.* 1992]:

- Interconnectivity
- Interoperability
- Cooperation

Although there are many systems for multidatabases, they all meet all of these three classes. For autonomous databases, the third class is especially important in large scale distributed environments. Many attempts have been carried out to adopt suitable technologies in distributed artificial intelligence, especially multi agent technologies [Manola *et al.* 1992, Papazoglou *et al.* 1992, Kambayashi *et al.* 1991, Shek *et al.* 1993].

Although HELIOS ass described in this paper is not restricted to a system for constructing multidatabases, it is along these lines [Yokota 1994].

To encapsulate heterogeneous data and programs in a distributed environment, a distributed object management system has been proposed [Manola *et al.* 1992], and it is expected to be a system for the next generation of distributed databases. In comparison, HELIOS expands agents in the following ways:

- HELIOS can dissolve heterogeneity in the contents of messages,
- hierarchical agent structure can be constructed in a bottom-up manner, and
- HELIOS can utilize cooperation/negotiation strategies such as constraint satisfaction problem

As a multi-agent system, HELIOS has the ability to make agents from problem solvers, databases, constraint solvers, and application programs by wrapping them by capsules and placing them in a common space, an environment. These agents have the ability to negotiate each other under a defined negotiation protocol in an environment with negotiation strategies defined in the capsules. For instance, many negotiation in a multi-agent system such as *contract net protocol* can easily be implemented on HELIOS. To utilize agents having constraint solvers in their substances, *distributed constraint satisfaction* and *constraint relaxation in distributed constraint satisfaction* [Yokoo 1993] can also be implemented.

HELIOS contributes various methods of constraint solving. One way is distributed constraint satisfaction and distributed constraint relaxation as we have described above. Another possibility is to make a constraint solver an agent. For instance, if we make a constraint solver for CAL [Aiba *et al.* 1988], and GDCC [Terasaki et al. 92] then we can obtain an agent that can solve algebraic equations including non-linear equations. By combining those constraint solving agents with different domains and introducing a utility function for satisfying constraints, then constraint solving and relaxation of constraints can be handled on multiple domains in HELIOS.

There is one other computation models for distributed problem solving: the distributed computation model of concurrent objects. We concluded that HELIOS has a different viewpoint to this distributed computation model. That is, HELIOS concentrates upon entities of computation or systems, while the distributed computation model concentrates upon computation or processes. This is why dynamism becomes an important issue in the distributed computation model. We aim to do a more precise comparison, and establish a computation model for HELIOS.

To increase the efficiency of the system, communication between agents should be decreased. Introducing proxy agents is one interesting idea that we concern. To verify the usefulness of HELIOS with respect to its various aspects, we need to implement large-scale application systems. Since current implementation does not meet the full specification of the logical model that we described in this paper, we need to look at some functions such as ontologies that have not yet been implemented.

Acknowledgments

We wish to acknowledge all members of HELIOS project for many discussions about the languages, the systems, and for many proposals of their applications. We would also like to acknowledge members of Heterogeneous Knowledgebase Task Group for their useful suggestions and fruitful discussions. Above all, we particularly thank Shunichi Uchida, the director of the ICOT research center for his successive encouragements.

References

[Aiba *et al.* 1988] A. Aiba, K. Sakai, Y. Sato, D. J. Hawley, and R. Hasegawa. Constraint Logic Programming Language CAL. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, November 1988.

[Cutkosky *et al.* 1993] M. R. Cutkosky, R. S. Engelmore, R. E. Fikes, M. R. Genesereth, T. R. Gruber, W. S. Mark, J. M. Tenenbaum, and J. C. Weber, "PACT: An experiment in integrating concurrent engineering systems," *IEEE Computer*, pp. 28–38, January 1993.

[Kambayashi *et al.* 1991] Y. Kambayashi, M. Rusinkiewicz, and A. Sheth (eds.), *Proc. of First International Workshop on Interoperability in Multidatabase Systems*, IEEE Computer Society, Kyoto, 1991.

[Kawamura *et al.* 1992] M. Kawamura, H. Sato, K. Naganuma, and K. Yokota, "Parallel Database Management System: Kappa-P," In *Proc. of the Inter-*

national Conference on FGCS 1992, Tokyo, June 1-5, 1992.

[Manola *et al.* 1992] F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie, "Distributed Object Management," *Int. J. of Intelligent and Cooperative Information Systems*, vol. 1, No. 1, pp. 5–42, 1992.

[Marcus 1980] M. P. Marcus, "A Theory of Syntactic Recognition for Natural Language," MIT Press, ISBN 0-262-13149-8, Cambridge, Mass. 1980.

[Nitta *et al.* 1994] K. Nitta, K. Yokota, A. Aiba, and M. Ishikawa, "Knowledge Processing Software," In *Proc. International Symposium on Fifth Generation Computer Systems 1994*, Tokyo, December 1994.

[Nitta *et al.* 1994] K. Nitta, M. Shibasaki, T. Sakata, T. Yamaji, W. Xianchang, H. Ohsaki, S. Tojo, I. Kokubo, and T. Suzuki, "A Legal Reasoning System: new HELIC-II," In *Proc. International Symposium on Fifth Generation Computer Systems 1994*, Tokyo, December 1994.

[Nishida 1994] T. Nishida, "The Knowledge Community (in Japanese)," In *Proc. of the 8th Annual Conference of JSAI*, pp.77–80, 1994.

[Papazoglou *et al.* 1992] M. P. Papazoglou, S. C. Laufmann, and T. K. Sellis, "An Organizational Framework for Cooperating Intelligent Systems," *Int. J. of Intelligent and Cooperative Information Systems*, vol. 1, No. 1, pp. 169–202, 1992.

[Papazoglou 1993] M. P. Papazoglou, "On the Duality of Distributed Database and Distributed AI Systems," Proc. CIKM, Washington DC, 1993.

[Rosenschein *et al.*] J. S. Rosenschein, and G. Zlotkin. "Rules of Encounter," MIT Press, ISBN 0-262-18159-2, 1994.

[Shek *et al.* 1993] H-J. Shek, A. P. Sheth, and B. D. Czejdo (eds.), *Proc. of Second International Workshop on Interoperability in Multidatabase Systems*, IEEE Computer Society, Vienna, 1993.

[Takeda 1994] H. Takeda, K. Iino, and T. Nishida, "Knowledge-sharing mechanism in the Knowledge Community KC_0," In *Proc. of the 8th Annual Conference of JSAI*, pp.279–282, 1994.

[Terasaki et al. 92] S. Terasaki, D. Hawley, H. Sawada, K. Satoh, S. Menju, T. Kawagishi, N. Iwayama, A. Aiba. Parallel Constraint Logic Programming Language GDCC and its Parallel Constraint Solvers. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 330–346, June 1992.

[Tsuda 1994a] H. Tsuda. "cu-Prolog for Constraint-Based Natural Language Processing," *IEICE Transaction on Information and Systems*, Vol. E77-D, No.2, pp. 171–180, February 1994.

[Tsuda 1994b] H. Tsuda, and A. Aiba. "Heterogeneous Natural Language Understanding in HELIOS," *FGCS'94 Workshop on Heterogeneous Knowledgebases*, December 1994, Tokyo.

[Yokoo 1993] M. Yokoo, "Constraint Relaxation in Distributed Constraint Satisfaction Problem," *Proc. of ICTAI*, pp. 56–67, 1993.

[Yokota *et al.* 1993] K. Yokota, H. Tsuda, and Y. Morita, "Specific Features of a Deductive Object-Oriented Database Language *Quixote*," Workshop on Combining Declarative and Object-Oriented Databases, (ACM SIGMOD'93 Workshop), Washington DC, May 29, 1993.

[Yokota *et al.* 1993] K. Yokota, and M. Shibasaki, "Can database predict judgments? (In Japanese)," EDWIN, Information Processing Society of Japan, July 21 – 23, 1993.

[Yokota 1994] K. Yokota, "From Database to Knowledge-Bases: Kappa, Quixote, Helios," In *Proc. International Symposium on Fifth Generation Computer Systems 1994*, Tokyo, December 1994.