6 Applied Parallel Processing Technology

6-1 Distributed Processing Language Overlay GHC and its Application Possibilities

SAITO Kenji

Today's high-performance computers and high-speed networks allow sophisticated applications of overlay networks. Meanwhile, usage of multi-core processors has been spreading. The level of concurrency we need to handle in software systems has been rising rapidly, which necessitates a language that can express massive concurrency in a natural way, which can work with both tightly and loosely-coupled multiprocessor environments in parallel executions.

This article describes still work-in-progress design of Overlay GHC, an overlay network programming language based on concurrent logic language GHC[1] (Guarded Horn Clauses), as a candidate for such a language, and its application possibilities especially in traceable networks.

Keywords
Programming language, Concurrent programming, Logic programming, Overlay network, Parallel inference

1 Introduction

Today, parallel execution environments for computer systems, i.e., environments for concurrent executions of programs with multiple processors, are becoming both sophisticated and widely available. This has led to changes in the level of concurrency we need to handle, i.e., the level of logical parallelism with multiple processes.

As an example, sophisticated applications of the Internet using overlay networks, such as P2P (peer-to-peer) or grid computing, have become real as individual computers are equipped with massive excess resources such as large-scale storage devices or high-speed processors, and those computers are now connected with one another via high-speed networks.

The designs of P2P systems are characterized by their usage of overlay networks such that participants are symmetrical, and their roles can be dynamically switched with one another. In designing such systems, one needs to describe a system as a set of autonomous agents, which requires handling of a sophisticated level of concurrency. In designing grid computing systems, too, although autonomy is not weighted as much as in the case of P2P, one needs to program a dynamic discovery and utilization of excess resources in computers distributed over the networks, which requires handling of a sophisticated level of concurrency.

Meanwhile, all major CPU manufacturers are planning to move towards multi-core architectures as performance of single-core processors will not scale any longer. Usage of multi-core processors has already been widespread, but one needs to handle a sophisticated level of concurrency in software designs to extract such processors' full performance.

Under such situations, the level of concurrency we need to handle in software systems has been rising rapidly, and programming languages to describe massive concurrency in natural ways are expected to emerge. The parallel execution of concurrent programs written in such languages need to support both tightly and loosely-coupled multiprocessor environments, as tightly-coupled multiprocessor environments represented by multi-core processors and loosely-coupled multiprocessor environments represented by P2P or grid computing necessarily coexist today.

In this article, the author describes the design of a still-under-development language Overlay GHC as a massive concurrency/parallelism description language, and its application possibilities especially in traceable networks. This language is for overlay network programming under such an assumption that each node is a multi-core processor system. It is based on concurrent logic programming language GHC[1] (Guarded Horn Clauses). Overlay GHC allows programmers to program overlay networks rapidly from bird's-eye view, and to describe the whole logical network as one program.

2 Background

In this section, the author describes the syntax and semantics of GHC, and language KL 1[2] as an extension of GHC for tightly-coupled multiprocessor environments.

2.1 GHC: Guarded Horn Clauses

GHC is a concurrent logic language designed by Dr. Kazunori Ueda, and was published in 1985 as part of research activities in ICOT (Institute for new generation COmputer Technology) that propelled so-called Fifth Generation Computer Project.

2.1.1 Preliminaries

The basic elements of GHC programs are "terms". A term is formed from function symbols and variables. Function symbols are symbols beginning with small letters, and variables are those beginning with capital letters or '_'. A variable alone is a term. Otherwise, a term is of the following form:

$$f(\gamma_1, \gamma_2, \ldots, \gamma_n)(n \geq 0)$$

where f is a function symbol, and γ's are terms. If $n = 0$, f is specifically called a constant. For example, "X", "a", "cons (a, X)", "cons (a, cons(b, nil))" are all terms.

An "atom"(atomic formula) is of the following form:

$$p(\gamma_1, \gamma_2, \ldots, \gamma_n)(n \geq 0)$$

where p is a predicate symbol, and γ's are terms. Predicate symbols are symbols beginning with small letters. For example, "is_list (cons (a, X))" is an atom.

2.1.2 Syntax of GHC

A GHC program is a set of guarded Horn clauses of the following form:

$$H \leftarrow G_1, \ldots, G_m \mid B_1, \ldots, Bn. \ (m, n \geq 0)$$

where H, G's, and B's are atoms, and '|' is the "commit operator".

H is called the "head", and G's and B's are called *goals*, which are sometimes distinguished as "guard goals" and "body goals", respectively. A goal may be a unification goal:

$$\gamma_1 = \gamma_2$$

where γ_1 and γ_2 are terms. H and G's together are called the "guard", and B's as a group are often called the "body". If there are no goals in the guard or body, it is denoted by "true". A program is invoked by a goal clause:

$$\leftarrow B_1, \ldots, B_n. \ (\mathrm{n} \geq 0)$$

where B's are goals.

2.1.3 Declarative semantics of GHC

A guarded Horn clause can be read as follows:

If every goal in its guard and body is true, its head is true.

The result of every successful execution of a GHC program conforms the above semantics (*soundness*). There may be some clauses not applied in an execution, so that the result might not be the only solution (*incompleteness*).

2.1.4 Operational semantics of GHC

Intuitively, each guarded Horn clause is considered as a rewrite rule of a goal, where its guard specifies the conditions to be satisfied for the rule to be applied, and its body specifies the actual goals to replace with. If more than one clause can be applied, one of them is selected non-deterministically. This irreversible act of applying a clause is called "commitment".

Assignment to variables is called "binding". Bindings are produced after commitments, and any attempts to bind the bound variables with incompatible terms necessarily fail. A binding to a non-variable term is called "instantiation".

Two terms are "unified" when they become lexically identical by binding the variables in each with the corresponding terms of the other.

An informal operational semantics of GHC follows:

1. Goal execution: Every goal in the goal clause is executed concurrently by the following steps:
   (a) Head unification: Variables appearing in the head of a clause is analogous with formal parameters of procedural programming languages. Clauses whose heads are unifiable with the calling goal become the candidates for commitment; variables appearing in their heads are bound with the corresponding terms in the calling goal.
   (b) Suspension rule: Guard goals of the candidates are executed concurrently, with a restriction imposed by such a rule that any attempts to instantiate the calling goal are suspended.
   (c) Commitment: The execution of the calling goal commits to a clause whose guard succeeds; the body of the committed clause replaces the calling goal. Unification goals in the body may instantiate the calling goal.
2. Success: A unification goal succeeds if its arguments are unified. A non-unification goal succeeds if it is eventually replaced by unification goals that succeed, or by an empty body. A program succeeds if every goal in its goal clause succeeds.
3. Failure: A unification goal fails if their arguments are not unifiable. A non-unification goal fails if its execution has no candidates for commitment, or the guard of every candidate fails. A program fails if any goal in its goal clause fails.

2.1.5 Process interpretation of GHC programs

A GHC program defines a concurrent program as a set of processes in the following way:

1. Recursively defined predicates define processes.
2. Conjunction of processes defines a network of processes.
3. Arguments of goals define local states of processes.
4. Shared variables among goals define communication channels. These variables are often used to represent streams (sequences of data).

Figure 1 is a stack program written in GHC, based on an example from[3].

":-" is the coding expression for '←'. "[a|b]", "[a, b]" and "[ ]" are syntactic sugars for "cons (a, b)", "cons (a, cons (b, nil))" and "nil", respectively. "io: printstream (Os)" is a process provided in our Overlay GHC implementation that prints the elements of stream "Os". The program produces the sequence "[2, 1]", "2", "[1]", "1", and "[ ]".

Let the author follow the conventions in logic programming, and express a predicate in the form "name of predicate/number of arguments". In this example, a completed list as a command sequence is passed to stack/3 process in the goal clause. However, by using an incomplete list in which CDR part (the rest

```
% Definition of stack(CommandStream, Data, OutputStream).
stack([pop|Cs], [X|List], Os) :-true | Os = [X|Os1], stack(Cs, List, Os1).
stack([push(X)|Cs], List, Os) :-true | stack(Cs, [X|List], Os).
stack([list|Cs], List, Os) :-true | Os = [List|Os1], stack(Cs, List, Os1).
stack([], List, Os) :-true | Os= [].

:- stack([push(1), push(2), list, pop, list, pop, list], [], Os), io:printstream(Os).
```

Fig.1 *Example: stack program in GHC*

of the list) of the last CONS is a variable, stack/3 can be used as a data-driven process that waits for arrival of a new command.

2.2 KL 1: Kernel Language one

KL 1[2] is an extension of GHC, designed by Dr. Chikayama, et.al., as the system description language for parallel inference machines in the so-called Fifth Generation Computer Project. While GHC is concerned with concurrency only, KL 1 is also concerned with parallelism, or how to execute concurrent programs including how to allocate processes onto processors. Yet, the semantics of GHC is not broken by this, and concurrency and parallelism are clearly separated in the language specification; by removing pragmas (directives to the language system) from a KL 1 program, the corresponding GHC program is obtained.

3 Design of Overlay GHC

3.1 Principles

The author has designed an extension of GHC for distributed real-time environments in the past[4]. In the trial, the author introduced a new concept of “timed guard” to specify the timing constraint in the delay between placement of a goal and the occurrence of the commitment for reduction of the goal. It was necessary to modify the semantics of GHC programs partially in this approach.

In the case of the design of Overlay GHC, the author instead took the same approach as KL 1, i.e., how to execute a GHC program in a distributed real-time environment is to be described by pragmas as it refers to parallelism, and the author has left the language specification of GHC untouched. By doing this, the author expects that past research products such as translations of GHC programs, or accumulation of GHC programs themselves, will be directly available for us, in addition to the effects of easing extraction of implementation-dependent part of programs or verification of correctness of programs resulted from clear separation of concurrency and parallelism.

3.2 Pragmas

Currently, the pragmas shown in Table 1 are defined in Overlay GHC.

Among them, those pragmas concerned with ordering executions of goals and prioritizing applicabilities of clauses are common with KL 1. Overlay GHC is characterized by pragmas concerned with placement of goals and control of timing.

In KL 1, goals can be allocated onto processors by using “@node (Node_number)” pragma. It can be said that it is a specification suitable for tightly-coupled multiprocessors where the numbers of processors are rather fixed.

On the other hand, in Overlay GHC, environments are assumed such that the configurations of multiple processors are dynamically changed, and pragmas are introduced to specify the nodes in relative to the node from which the goal clause is cast, and a remote goal placement pragma “node_id (ID)” is introduced to specify the node with an identifier. “ID” in this pragma is an identifier expressed in the form of URL. For example, with “xmpp://” scheme, a Jabber ID can be specified, and with “pgp://” scheme, a PGP public key user ID can be specified. The language

Table 1 List of Pragmas in Overlay GHC

Type	Name	Function	Note
Ordering	@priority(Level)	Executes goal with the specified priority.	Common with KL1
	@lower_priority	Executes goal with a low priority.	
	alternatively.	Lowers priority of clauses to follow.	
Goal placement	@this_node	Executes goal at the calling node.	Specific to Overlay GHC
	@other_node	Executes goal at a non-calling node.	
	@node_id(ID)	Executes goal at the node with the specified ID.	
Real-time	@periodic(Msec)	Executes goal at the specified interval (msec).	

system needs to have mechanisms to map those identifiers onto actual targets of communication.

It is also noted that in GHC/KL 1, a single parallel inference machine was assumed to which just one console is attached, which has led to a specification that at most one built-in process is allowed in one program to handle the console I/O stream such as io: outstream/1.

On the other hand, in Overlay GHC, loosely-coupled multiprocessor environments are assumed in which each node is an independent computer (although each computer is often a tightly-coupled multiprocessor environment with one or more multi-core processors), which has led to a specification that one built-in process to handle the console I/O stream can be generated on each node. It has been made possible by this to describe as one program an overlay network where each participant can perform I/O from terminals in their vicinity such as a chat program shown in Fig. 2. It has also become possible to have multiple virtual nodes on one computer, where each node has an independent window to perform terminal I/O, which will be useful in simulations during development of programs such as P2P systems.

The periodic execution pragma "@periodic (Millisecond)" is to specify the interval between a placement of a goal and the commitment for the reduction of the goal. Using this, goals described as recursive processes can be periodically executed with specified intervals. This functionality can be used for polling sensor values, for example.

3.3 RGP: Remote Goal Placement

Goal placement pragmas in Overlay GHC automatically perform migrations of programs. If the node on which a goal is to be allocated does not have the corresponding program, the program is automatically transferred to the node before the placement of the goal happens (security considerations of this is described in section **3.6**).

"Remote Goal Placement (RGP)" is one of the techniques that characterize programming in Overlay GHC. By this technique, many interesting concepts in distributed programming can be realized. The author and associates are hoping that a new network programming culture using this will blossom because it is semantically different from Remote Procedure Call (RPC).

Figure 2 shows a symmetrical (without a server) group-chat program using this technique.

A new user can participate in the chat by invitation basis. In the program, "\=" is a built-in predicate to denote impossibility of unification, and "otherwise" is a built-in predi-

```
terminal(NickName, InStream, OutStream) :-true
  |  keyboard(NickName, InStream, OutStream),
     display(NickName, OutStream, DisplayStream),
     io:outstream([write('terminal started.'), nl|DisplayStream]).

keyboard(NickName, InStream, OutStream) :-io:read(X)
  | checkInput(NickName, InStream, OutStream, X).
keyboard(NickName, [Term|InStream], OutStream) :-Term \= line(NickName, _)
  | OutStream = [Term|Xs], keyboard(NickName, InStream, Xs).
keyboard(NickName, [line(NickName, X)|InStream], OutStream) :-X \= left
  | keyboard(NickName, InStream, OutStream).
keyboard(NickName, [line(NickName, left)|InStream], OutStream) :-true
  | InStream = OutStream.

checkInput(NickName, InStream, OutStream, X) :-Line := NickName + ': ' + X
  | OutStream = [line(NickName, Line)|Xs], keyboard(NickName, InStream, Xs).
checkInput(NickName, InStream, OutStream, join(Name, ID)) :-true
  | terminal(Name, InStream, Xs)@node_id(ID), keyboard(NickName, Xs, OutStream).
checkInput(NickName, InStream, OutStream, leave) :-true
  | OutStream = [line(NickName, left)|Xs], keyboard(NickName, InStream, Xs).
checkInput(NickName, InStream, OutStream, X) :-otherwise
  | keyboard(NickName, InStream, OutStream).

display(NickName, [line(_, Line)|OutStream], DisplayStream) :-Line \= left
  | DisplayStream = [write(Line), nl|Xs], display(NickName, OutStream, Xs).
display(NickName, [line(NickName, left)|OutStream], DisplayStream) :-true
  | DisplayStream = [write('I have left.'), nl].
display(NickName, [line(Someone, left)|OutStream], DisplayStream)
 :- NickName \= Someone, Line := Someone + ' has left.'
  | DisplayStream = [write(Line), nl|Xs], display(NickName, OutStream, Xs).

% Starts a group chat as the initiator whose nickname is 'foo'.
:-terminal('foo', X, X).
```

Fig.2 A simple group-chat program written in Overlay GHC

cate that succeeds when guards of all other candidate clauses fail.

In this program, terminal/3 is the main process to express a user. keyboard/3 is a process to accept inputs including those from other users, which forwards the inputs to the succeeding user in the ring-shaped overlay network. checkInput/4 belongs to keyboard/3, and interprets commands. If the input from the user is in the form “join(Name, ID)”, it makes a new terminal/3 process generated on the remote node using RGP. display/3 process generates the console output stream.

3.4 Detecting and handling failures

In distributed computing, an execution of a program may fail even if the program is logically sound, because of failures of processors or communications. The program in Fig. 2 needs to be rectified because it does not assume occurrences of failures.

The author and associates have been investigating how to naturally describe detection and handling of failures in Overlay GHC. In doing so, we have weighted on such a design constraint that the semantics of a GHC program remains unchanged even in the presence of failures. We have noted that a local goal placement and RGP can be equally treated without contradictions, if the failure model can be limited to fail-stop[5].

In the fail-stop failure model, failed processes stop computations without causing any side-effects. In Overlay GHC, this means that execution is perpetually suspended, which does not change the meaning of a GHC program because Horn clauses do not contain the concept of time (they can be regarded as very slow processes).

Many failures can be treated as fail-stop by detecting them with loose criteria, and forcibly stopping the processes in the case of false positives.

The author and associates has been investigating to introduce timeout/1 as a built-in predicate to be used in guards, in order to detect the perpetual suspensions resulted from fail-stop failures, and to recover from such situations.

Figure 3 shows an example of a program using timeout/1.

This program sends out “ping(Pong)” every 30 seconds, and detects a failure when variable “Pong” has not been instantiated for 10 seconds.

Byzantine failure[6] is a model of failure that cannot be made fallen back to fail-stop.

In Byzantine failure model, a failed processor may bind an arbitrary term to a variable. To detect such a failure, redundancy needs to be programmed. In Overlay GHC, redundancy can be easily programmed by placing multiple equivalent goals in the goal clause, which are allocated onto different remote nodes. When such a failure occurs, and if the same variable was bound to contradicting terms by equivalent multiple goal executions, the unification fails, and the whole program fails because the goals in the goal clause are executed in AND parallelism. If we would like to detect such a situation programmatically, and to recover from it, mechanisms such as meta-call of predicates will be necessitated.

3.5 Dynamic many-to-one communication

In Overlay GHC, instantiation of a variable may happen at most once. This makes it impossible for multiple writers to write simultaneously to a single stream.

In concurrent logic languages, this type of difficulties has been avoided by using merge/$(n+1)$ process that takes n independent streams and outputs one merged stream. This process can be easily implemented by the concurrent logic languages themselves. However, in overlay network programming, there may be requirements to support more dynamic many-to-one communications. For example, in P2P systems, each node may have to receive messages from a dynamically changing set of neighbor nodes.

The author and associates have been investigating to introduce“queue”data type to solve this problem. An example is shown in Fig. 4.

In the program, “<<” is a built-in predicate to add a message to a queue.

A queue is generated with a variable as an argument that represents an input stream to a

```
checkAlive(Stop, Os) :-timeout(30000)
  | Os = [ping(Pong)|Os1], checkPong(Pong, Stop), checkAlive(Stop, Os1).
checkAlive(stop, Os) :-true | process-failure-of-the-peer.

checkPong(pong, Stop) :-true | true.
checkPong(Pong, Stop) :-timeout(10000) | Stop = stop.
```

Fig.3 *A ping program written in Overlay GHC*

```
createStack(Q, Os) :-true | Q := createQueue(Cs), stack(Cs, [], Os).

stacker(Q, ...) :-true | Q << push(X), ..., stacker(Q, ...).

:-createStack(Q, Os), ..., stacker(Q, ...)@node id(ID1), stacker(Q, ...)@node id(ID2).
```

Fig.4 *Multiple writers to a single stream*

process (it is typically executed on the node that receives the inputs). Multiple writers can put data onto the queue concurrently, and the output is resulted as a merged single stream that uses the variable as the first CONS.

This data structure resides mainly in the local node. In the local node, it holds the variable to represent the output stream, and when a data arrives from a remote node, it sequentially writes the data to the stream using an exclusion mechanism such as a semaphore. In the remote nodes, it holds the reference to the main structure in the form of the node ID and queue ID, and upon putting a data to the queue, it transfers the data to the corresponding node.

3.6 Security considerations

Without an appropriate security design, RGP may be abused for transferring and executing malwares. Therefore, it is needless to mention that RGP needs to be allowed only when the specified remote node agrees to accept and execute the goal to be transferred. The author and associates have been investigating the best way to realize this (the current implementation has a list of Jabber IDs to accept remote goals from, as an easy tentative measure).

4 Application possibilities

4.1 Distributed computer

Various application possibilities can be considered for Overlay GHC with which programmers can program overlay networks dynamically. Those possibilities can be uniformly treated by the concept of “distributed computer”.

A distributed computer involves placements of part of a computer to remote processors for load distribution or routing I/O for specific purposes. In Overlay GHC, such a functional distribution can easily be described using RGP.

4.2 Application possibilities for traceable networks

As an application in the toolkit towards realization of traceable networks, we could describe a parallel inference engine with the language, as GHC was originally designed as a system description language for parallel inference machines. We expect that direct application of the past research products is possible as many research such as describing production systems with GHC were performed in the past as introduced in[7].

This inference engine dynamically generates rules based on the materials for deductions made available from the past observations, and infers appropriate actions against occurrences of incidents. This engine needs to realize iterations of inferences and investigations.

We hope that a possibility of an inference engine where multiple hypotheses are verified simultaneously can be pursued, utilizing the characteristics of the language designed for distributed processing.

As for the concrete design of the inference engine, we would like to proceed using the past research results in concurrent logic languages not only on production systems but also on various knowledge information processing.

As another application of Overlay GHC for traceable networks, we hope to use it for combining in a parallel way the parts of the toolkit as processes, as the language is suitable as a glue language to combine existing processes.

5 Implementation of the language system

As an example of language systems of Overlay GHC, the author and associates are developing an interpreter written in Java in an open source community. This interpreter is a plug-in for *wija*, an XMPP-based instant messaging system also written in Java and developed by the author. The latest version of the implementation is available from the following URLs:

- Latest release version:
  http://www.media-art-online.org/ghc/
- Latest development version:
  http://www2.media-art-online.org/nightly/

This implementation of Overlay GHC includes Package Programming Interface (PPI). This interface is for other plug-ins of *wija* to provide Overlay GHC primitives and processes, and supports programmers combining existing functionalities as Overlay GHC processes.

6 Related work

6.1 Distributed KL 1

Distributed KL 1 is a predecessor to Overlay GHC as an extension of GHC for distributed programming. It has been implemented as DKLIC[8] language system that is an extension to KLIC, a KL 1 to C compiler.

DKLIC has a notion similar to remote goal placement, but it is called by the term “remote predicate call”, which is rather similar to RPC, as predicate migration does not seem to have been implemented. Problems such as multiple writers to a stream or processor failures do not seem to have been addressed either.

6.2 P2/OverLog

P2[9] is a system using OverLog language to express overlay networks in a compact and reusable form.

OverLog and Overlay GHC share the similar goals (rapid and declarative implementations of overlay networks) and similar approaches (logic programming). They are different in that OverLog is a query language based on DataLog, a subset of Prolog, while Overlay GHC is a descendant of concurrent logic programming languages. Further comparison studies are planned between the two environments.

6.3 Erlang

Erlang[10] is a functional programming language designed for expressing massive concurrency.

Erlang and Overlay GHC are related in that concurrent computation in Erlang is based on Actor model[11], and the model is regarded as a special interpretation of concurrent logic programming[12]. Further comparison studies are also planned between those languages.

6.4 StreamIt

StreamIt[13] is a language for high performance streaming applications. It has two goals: to provide high-level stream abstractions, and to serve as a common machine language/high-level assembly language for grid-based architectures (instead of C as a common high-level assembly language for von Neumann architectures).

Because stream programming is one aspect of concurrent logic programming, many language features of StreamIt are also expressible in Overlay GHC. At this moment, Overlay GHC is not too concerned about performance of the generated overlay networks, but perhaps it is worth investigating to compile an Overlay GHC program into StreamIt code.

7 Future work

Overlay GHC is a programming language still under development. To improve its design specification, we need to accumulate experiences in programming with this language.

As an example of implementing existing overlay networks with this language, a trial to implement Kademlia[14], an instance of distributed hash tables, in Overlay GHC has been started. Trials to implement other distributed hash tables and various distributed algorithms will follow.

As for applicability to traceable networks, the author intend to describe an actual parallel inference engine to be embedded in the toolkit so that we can verify the usefulness of the language in the process of problem solving in the field by iterating inferences and investigations triggered by occurrences of incidents.

At the same time, the author and associates are planning to develop a scripting language we tentatively call OG. This language corresponds one-to-one with Overlay GHC. Overlay GHC is a high-level assembly language for non-von Neumann architectures in a different level from StreamIt (readers are reminded that KL 1 was a system description language for parallel inference machines). Through experiences of training a small number of programmers (mainly students of environmental informatics) to use Overlay GHC, the author and associates have gained an impression that higher-level languages are needed to ease the burdens of programmers who are not familiar with logic or concurrent programming (or programming in general).

8 Conclusions

In this article, the author introduced Overlay GHC, a still under-development overlay network programming language based on concurrent logic programming language GHC. The author also discussed as one of its application possibilities applicability of the language to a parallel inference engine for traceable networks.

The level of concurrency we need to handle has been rising rapidly, as parallel execution environment of computer systems are becoming sophisticated and widely available as seen in P2P, grid computing and wide acceptance of multi-core processors. Under such situations, many programming languages oriented towards expression of concurrency and parallel executions have been proposed and investigated. The research and development of Overlay GHC should be placed as one of those trials.

Among such languages, Overlay GHC is characterized by allowing programmers to grasp overlay networks from bird's-eye view, to rapidly program them, and to express the whole logical network in one program.

Currently, Kademlia is being implemented as an example of actual overlay network programming with this language. Trials for implementing other distributed hash tables and various distributed algorithms are also under way. At the same time, the author plans to describe the parallel inference engine as part of the toolkit for realizing traceable networks.

References

1 K. Ueda, Guarded Horn Clauses. Ph.D thesis, The University of Tokyo, 1986.

2 K. Ueda and T. Chikayama, "Design of the kernel language for the parallel inference machine", The Computer Journal, Vol.33, Dec. 1990.

3 S. -O. Nyström, "Guarded Horn Clauses, application and implementation", Tech. Rep. Report 41, Uppsala Univer-sitet, Nov. 1987.

4 K. Saito, "TGHC: Timed guarded horn clauses", in Proceedings of the Ninth TRON Project Symposium (Interna-tional), pp.122-134, IEEE Computer Society Press, Dec. 1992.

5 R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems", ACM Transactions on Computer Systems, Vol.1, Aug. 1983.

6 L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem", ACM Transactions on Programming Languages and Systems (TOPLAS), Vol.4, Jul. 1982.

7 K. Fuchi, K. Furukawa, and F. Mizoguchi, Parallel Logic Language GHC and its Applications. Kyoritsu Shuppan, 1987. (in Japanese).

8 R. Matsumura, H. Takayama, Y. Takagi, N. Kato, and K. Ueda, "Design and implementation of distributed language system DKLIC", in Proceedings of the 19th Conference of JSSST, Sep. 2002. (in Japanese).

9 B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays", in Proceedings of ACM Symposium on Operating System Principles (SOSP), Oct. 2005.

10 J. Armstrong, "The development of Erlang", in SIGPLAN International Conference on Functional Programming, 1997.

11 G. Agha, Actors: a model of concurrent computation in distributed systems. MIT Press, 1986.

12 K. M. Kahn and V. A. Saraswat, "Actors as a special case of concurrent constraint (logic) programming", ACM SIGPLAN Notices, Vol.25, Oct. 1990.

13 W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications", in Proceedings of the 2002 International Conference on Compiler Construction, Apr. 2002.

14 P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the XOR metric", in Proceedings of IPTPS02, Mar. 2002.

***SAITO Kenji**, Ph.D.*

Expert Researcher, Traceable Secure Network Group, Information Security Research Center

Distributed Systems, Real-time Systems, Computer Communication