SOOC-94: An Experimental Language toward Building Real-World Computing Systems*

Yasusi Kanada

Tsukuba Research Center, Real World Computing Partnership
Takezono 1-6-1, Tsukuba, Ibaraki 305, Japan
Phone: +81-298-53-1663
Fax: +81-298-53-1652
E-mail: kanada@trc.rwcp.or.jp

Abstract

Today's computer applications often require on-line input and output and sometimes requires change of computation methods while processing complex tasks. In future, they will probably be required to change even the objectives of computation. Such a new style of information processing is called *real-world computing* in the present paper. Real-world computing requires computation languages to work without fixed sequence of control nor fixed data flow, to work without a whole plan of computation, to be always open to environment at any time, to have possibility of distributed and parallel processing, to have possibility to respond to nondeterministic actions, and so on. Because today's programming languages are considered not to satisfy these requirements, a new computation model called CCM (chemical casting model) and a language called SOOC-94, which partially satisfies the requirements, are developed as the first step. Computation is regarded as biased random walk in CCM. The random walk is realized by a randomly acting forward-chaining production system, and the bias is realized by locally computed evaluation functions

* Part of this work was done at Central Research Laboratory, Hitachi Ltd.

in SOOC-94. I have developed a preliminary version of compiler and interpreter, and analyzed the computation processes and results of graph coloring problems for example.

1. Introduction

Today's high performance computers are used for more and more complex applications, such as large-scale expert systems or simulation of complex natural or social phenomena. In the period of batch processing, computers performed closed processing based on input supplied beforehand, and outputted the result after the completion of computation. However, today's computer applications often require on-line input and output, and sometimes require change of computation methods while processing complex tasks. In other words, on-line and real-time processings have become much more important. In future, they will probably be required to change even the objectives of computation. Such a new style of information processing is called *real-world computing* in the present paper.[1]

Not only hardware nor operating systems but also programming languages must meet the requirements of the real-world computing. Because, implementation of the information processing based on new paradigms using programming languages based on old paradigms is similar to implementation using machine languages. However, do today's programming languages try to meet the requirements? Although various new languages, such as logic or functional programming languages, have been developed, older languages, such as Fortran or C, are still major in writing applications. One of the reasons is considered to be that the programming languages have been developed toward different directions from meeting the processing style changes, and thus they have not tried to meet the requirements.

The objective of this research is to build a new computation model and a language, which are suitable for describing complex real-world computing systems that are open to their environment. In this model and this language, input from the environment or output to it must be allowed while computing, and even change of computation methods should be allowed. As mentioned before, one of the features of real-world computing is that it is real-time, and including real-time processing facilities into conventional languages has been researched. However,

[1] This definition of real-world computing is not necessarily popular in the community of programming languages, nor in our organization, the Real World Computing Partnership.

the objective of our research is to reexamine the basic mechanism of computation radically or to ask the question by Backus [Bac 78] again, i.e., “Can programming be liberated from von Neumann style?,” and to go far beyond the solution given by Backus.

The backgrounds of this research are the results given by the researches of self-organizing systems, i.e., Prigogine [Pri 87], Haken [Hak 78], and so on, which are excellently surveyed by Jantsch [Jan 80]. This research is also influenced by the researches on action-based robots [Bro 86] and Artificial Life [Lan 89, Lan 92], which have *emergent* properties [For 91].

This paper reports a trial towards the establishment of new style of computation languages. A computation language SOOC-94 is proposed as a prototype. SOOC-94 is based on the chemical casting model (CCM) [Kan 92, Kan 94].[1] The purpose of the present paper is to analyze the requirements of real-world computing, to explain the features of SOOC-94, and to verify that this language partially satisfies the requirements. The requirements are analyzed in Section 2. Several new language features of SOOC-94 are explained in Section 3, and a simple example using SOOC-94 is shown in Section 4. Whether the requirements described in Section 2 are satisfied is verified, compared to conventional programming languages, in Section 5. Several implementation issues are explained in Section 6. Conclusions are given in Section 7.

2. Requirements of Real-World Computing

Six requirements of new style of computation models or languages by the new style of information processing, or real-world computing, are analyzed in this section.

(1) Openness to the environment

Computation must be always open to environment when using the new language. Namely, the system must be able to accept sudden and unexpected input of data or program from the environment of the computation, such as users or networks. Although it is not impossible to satisfy this requirement using procedural programming, it is similar to assembly program-

[1] “Casting” is used instead of “programming” in CCM because “programming” means making a complete plan, which is not the target of CCM or SOOC-94.

ming. Better features must be supported by the new language. Conventional languages are not quite suitable for satisfying this requirement because they are not prepared to accept sudden and unexpected input or change, even if they have parallel processing features, which are good for accepting *expected* input.

(2) Possibility of distributed/parallel processing (DPP)

DPP must be possible when using the new language. Distributed processing means conceptually distributed processing here. Namely, it means processing by cooperating agents that sparsely communicate each other, but not necessarily physically distributed. Such distributed processing is important for realizing computation without a whole plan, which will be described in item (5). Many researches aim such distributed processing by extending conventional programming languages. However, I think it is doubtful for such researches to perform the objective without reexamining the microscopic (procedural or functional) mechanism of computation. Although the purpose of parallel processing and distributed processing are different, they require similar conceptual properties to languages.

(3) Respondability to nondeterminacy

The computation must respond to nondeterministic actions caused by the environment or DPP when using the new language. Unexpected information may be inputted as described in item (1) and a change of computation purpose may be required, even when the system is busy at computation, and a type of nondeterminacy is inevitable in DPP. For example, the timing and contents of communication between agents are not known beforehand, and the computation is sometimes dependent on the order of computation, that is not decidable beforehand. Respondability to nondeterminacy assures robustness of the system.

(4) Unfixed control flow and data flow

Computation must be able to perform even when the sequences of control flow or the data flow are not fixed in the new language. Because, if they are fixed, flexible computation required by the real-world computing is not realized. Ashby advanced *the law of requisite* *variety*, which states that a system must have the same or more amount of variety than that of the environment [Ash 61]. This law can be interpreted that there must be more internal variety, i.e., paths of control flow and data flow that can be switched by external causes, than the external variety, i.e., the nondeterminacy caused by the environment. For example, if the sequence of control is fixed, the system cannot meet the requirement on a change of processing order. Although this requirement has been tried to be satisfied by conventional programming paradigms, such as logic programming, constraint programming or production system, they meet only very limited part of this requirement.

(5) Computability from partial and local plan

Computation must be able to be performed even when only a partial and local computation plan is given, if using the new language. In other words, the new language must support emergent computation [For 91]. When using conventional languages, basically, computation cannot be started unless a whole and global plan is given. That is, a specification must be given first, and a program, which follows the specification, must be given before starting computation. However, the objective of computation is not usually very clear when computational systems begin to work, if they are large, complex and open to their environment. The plan may be added or it may be changed after the computation starts. Even a plan or objective may become clear by the computation. Thus, the computation must be able to run without a whole plan.

(6) Real-time processing

Real-time applications should be able to be implemented in the new language. High performance is needed to satisfy this requirement.

(7) General-purposeness

New languages should be general-purpose as well as conventional programming languages. Less general-purpose languages will not be successful unless its application is very much popular.

3. SOOC-94: A Language Based on Biased Random Walk

A computation model based on biased random walk and a language called SOOC-94, which realizes the model, is explained in this section.

3.1 Computation as biased random walk

A computation process can be regarded as a sequence of state transitions. State transitions are caused by the system itself (by the program) or by an input from the environment. This model is basically sequential. DPP cannot be modeled precisely by this model because state transitions may occur independently and in parallel in each part of the system in DPP. However, this model is considered to be better than models with explicit parallelism. The reason is explained later in Section 5.

The set of computation states is called *state space* (See **Figure 1** (a)). Although it is called "set," it is not asserted that all the states are given beforehand. Thus it does not mean the mathematical sense of set. Computation is a sequence of transitions between the elements of state space. If the elements of a state space are regarded as vertexes, and each pair of two vertexes is connected by a directed edge if the transition from the first vertex to the second is possible, a directed graph, such as illustrated in Figure 1 (b) is created. This graph is called a *computation graph*.[1]

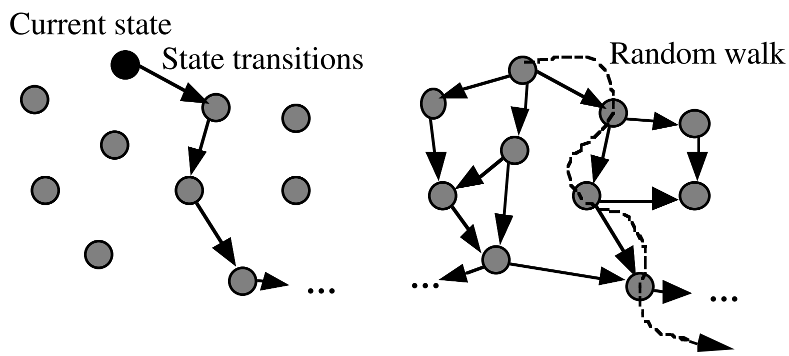
It is asserted that an edge is selected from all the edges connected to a vertex, a state, randomly in computation. Thus, computation is basically a *random walk* on a computation graph in our model. The reason that the random walk is used is explained later. However, the objective of computation is not usually performed or it is too slow by a complete random walk. So the random walk is *biased* by an appropriate method, and it is induced to a better direction.

The reason why computation is based on a biased random walk is that this mechanism is considered to satisfy the requirements explained in the last section. How the model and language satisfy the requirements is examined in Section 5, and it will be the better solution to this ques-

[1] If the computation is goal-oriented, the computation is called *search*, and a computation graph is called a search graph [Kan 93b]. If it is a tree, it is called a search tree.

tion. So, only the most major reason is explained here. This mechanism is used because the computation can be performed only by giving a computation graph without giving whole plan nor sequence of control.

Kanada [Kan 92, Kan 94] proposed the chemical casting model (CCM), which is based on biased random walks on computation graphs. CCM has an analogy to chemical reaction systems. The semantics of SOOC-94, the language described below, is based on CCM.



(a) State space (b) Computation graph and random walk

Figure 1. State space and computation graph

3.2 Reaction rules — description of partial computation plan

Edges of computation graphs, or possible state transitions, are given by forward-chaining production rules in SOOC-94. Namely, a computational system is expressed as a type of production system. However, this type of production system is quite different from that used for building expert systems or for the researches of cognition, for example, OPS5 [For 81].

Production rules are given by user and they are fixed in SOOC-94. As long as production rules are not changed (or learned) by computation, the requirements explained in the previous section may partially be remained unsatisfied. However, it seems to be good as a starting point. Because production rules are fixed in SOOC-94, all the states that there exist transitions from a known state, or all the neighbor vertexes of a known vertex on the computation graph, can be known beforehand. An appropriate state is selected from these states, and transition to this state is caused. The state is selected randomly. Thus the computation is basically a random walk.

To explain the syntax and the semantics of reaction rule definition in SOOC-94, a simple and probably meaningless example of rule is given here.

```
(defrule change-color
  (var V)                              ; the declaration part
  (exist V vertex (:color 'Blue))      ; the LHS condition
 -->
  (exist V vertex (:color 'Red)))      ; the RHS condition
```

The name of the rule is **`change-color`**. A variable named **`V`** is declared and used in this rule. A syntactic element that begins with "**`(exist`**" is called an *existence condition*. Existence conditions before the arrow (**`-->`**) are called the left-hand side (LHS) conditions, and those after the arrow are called the right-hand side (RHS) conditions. In the above rule, there is only one LHS condition, and it matches an object of type **`vertex`**. The matched vertex is referred to as **`V`** in the rule. There is also only one RHS condition in the above rule. The same vertex, **`V`**, is matched to the above RHS condition. There may be two or more LHS and RHS conditions in general.

Data are put in the working memory (WM) in SOOC-94. A state transition caused by a rule application is called *reaction* in SOOC-94 because of analogy to chemical reaction system. LHS conditions are the preconditions of the reaction, and RHS conditions are the post conditions. The above LHS condition can match a vertex in WM if the color of the vertex is **`Blue`**, and the above RHS condition can match a vertex in WM if it is **`Red`**. Thus, when the LHS of this rule is tested, a vertex is randomly selected, and the color of the vertex is changed from **`Blue`** to **`Red`** by the reaction.

If two rules, one of which is the above rule that changes the color from **`Blue`** to **`Red`**, and one of which is a rule that changes the color from **`Red`** to **`Blue`**, are given, and there are two vertexes in WM, the computation graph is as illustrated in **Figure 2**. Because this graph is strongly connected, i.e., any two vertexes are connected by a directed path, and no bias has yet given, there is no absorbing (i.e., termination) state and the computation on this graph will continue forever.

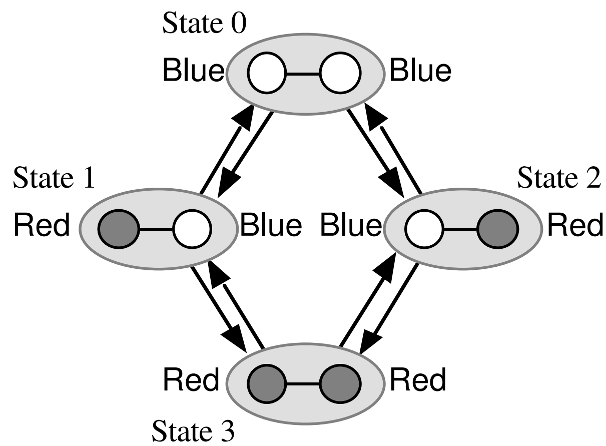


Figure 2. A computation graph for simple coloring

3.3 Local order degree — description of non-sequencing control

Bias to random walk is described by the user using the form of locally computed evaluation function, which is called *local order degree* (LOD) in SOOC-94.[1] The reason that biases are specified by evaluation functions but the direction of transition is not specified directly is to make the control flexible.

Bias works as follows. The sum of the LODs of the object is called *instance order degree* (IOD). When a set of objects, which match the conditions of a rule, is selected, the IOD before the reaction and that after it are computed and the values are compared (before actually causing the reaction). The current method of biasing is that the reaction occurs only when the IOD increases by the reaction. This means that bias is binary, i.e., whether it is existent or non-existent. Non-binary, or continuous bias is also possible, but not yet tried.

A simple example of LOD is given here.

```
(deforder ((V vertex))                          ; Definition of a self order degree[2]
  (if (eql (vertex-color V) `Blue)
    0
    1))
```

[1] It is called LOD because the state in which LOD is higher can be considered to have more order.
[2] An LOD defined for one object is called a *self order degree*.

The syntax of LOD definitions is similar to that of method definitions in Common Lisp (CLOS). This LOD is defined for an object of type **`vertex`**. The LOD of a vertex is 0 if its color is equal to **`Blue`**, and it is 1 otherwise. If this definition is used with two rules described in the previous subsection, only reactions caused by the first rule, **`Blue`** to **`Red`**, occur, because a reaction that is caused by the first rule increases the LOD, which is equal to the IOD of the matched vertex, from 0 to 1, but a reaction that is caused by the second rule decreases it from 1 to 0. Thus this LOD biases the random walk strongly and thus the biased computation deterministically terminates in this case.

If a weaker bias is required, it can be realized by modifying the semantics of rule applications. The computation will be stochastically terminates under a weaker bias. A language feature that implements probabilistic reactions caused by bias will be added in a future version of SOOC-94.

Another example is given.

```
(deforder ((V1 vertex) (V2 vertex))  ; Definition of a mutual order degree[1]
  (if (eql (vertex-color V1) (vertex-color V2))
    0
    1))
```

This LOD is defined between two objects of type **`vertex`**. The LOD is zero if the colors of the two vertexes, **`V1`** and **`V2`**, are equal, and it is one otherwise.

4. An Example using SOOC-94 and its Execution

A set of rules, LODs and a WM is called a *system* in CCM. A system to solve a coloring problem is described and its execution process and result is shown in this section. Other examples in CCM, i.e., an *N* queens system, a TSP system, and so on, are described or mentioned by Kanada and Hirokawa [Kan 94], though their codes in SOOC-94 are not shown there.

[1] An LOD defined between two objects is called a *mutual order degree*.

4.1 A coloring system by CCM

The graph coloring problem is a problem to color the vertexes of an undirected graph using specified number of colors, for example, four colors. For example, a problem of coloring the graph with five vertexes, is shown in **Figure 3**. Although the simplest version of this problem is not real-world, several extensions of this problem to real-world are discussed in Section 5.

The data structure for solving the problem is as follows. Both vertexes and edges are represented by objects. Then the graph shown in Figure 3 is represented by the data in **Figure 4**. An object of type vertex has a color as its internal state. C1, C2, C3 and C4 are the colors in Figure 4. Two data types, **`vertex`** and **`edge`**, must be declared in SOOC-94 by the following declarations, because there are two kinds of objects.

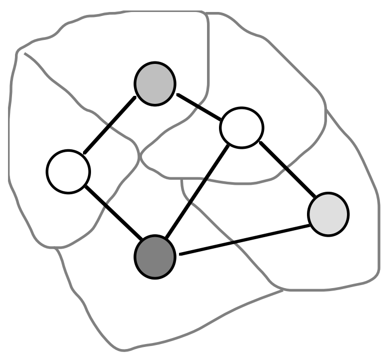


Figure 3. An example of graph coloring problem

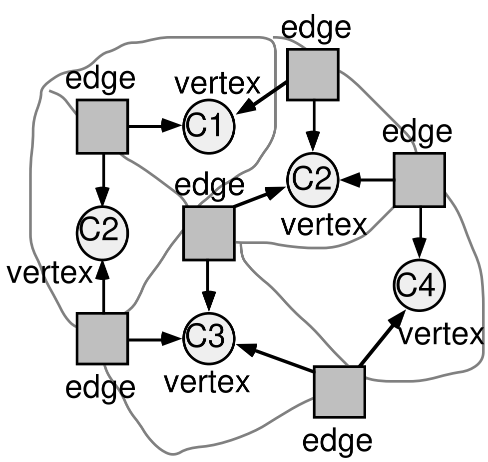


Figure 4. The data structure for solving the graph coloring problem

```
(defelement vertex      ; A vertex of graphs ("element" means data type in SOOC-94 )
  id                      ; the identifier of the vertex
  color)                  ; the color of the vertex

(defelement edge        ; An edge of graphs
  (* vertex))             ; the vertexes connected to the both end of the edge
```

The semantics of these declarations are similar to **`defstruct`** macro of Common Lisp. Thus they define data structures, **`vertex`** and **`edge`**, and define component access functions; **`vertex-color`**, **`edge-vertex`**, and so on.

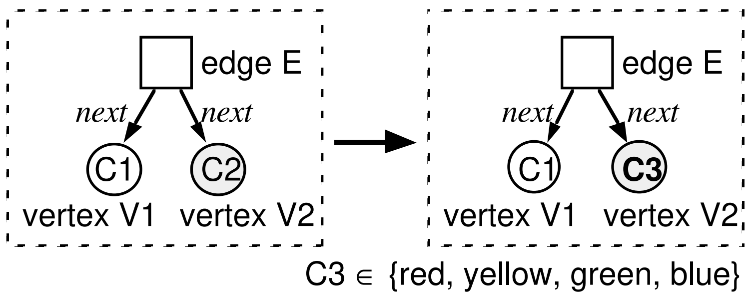
Type **`vertex`** has two components, **`id`** and **`color`**. An edge always has two neighbor vertexes, though type **`edge`** may have any number of components named **`vertex`** by the semantics of SOOC-94. They have the same name because they must not be distinguished in the rule because edges are undirected. If these components have different names, they implement a directed edge, where the head and tail are distinguished. A graph to be colored, such as shown in Figure 4, must be created using these data types.[1]

The only reaction rule to solve the problem is shown in SOOC-94 and in a visual form in **Figure 5**. This rule refers only to neighboring two vertexes and the edge between them and change the color of one of the vertexes randomly. Although only one rule is used, the computation graph is strongly connected and there is no absorbing state if LOD is zero, i.e., not defined.

[1] Lisp expressions are used for building initial data structures in SOOC-92, a previous version of SOOC-94, because language features for this purpose have not yet been given in SOOC-92.

```
(defrule change-color
  (var E V1 selected)                          ; three variables are declared
  (exist E edge (:vertex V1))                  ; the first LHS condition
  (exist V1 vertex)                            ; the second LHS condition
 -->
  (exist E edge (:vertex V1))                  ; the first RHS condition
  (oneof '(red yellow green blue) selected)
  (exist V1 vertex (:color selected)))         ; the second RHS condition
```

(a) The reaction rule in SOOC-94



(b) A visual representation of the reaction rule

Figure 5. The reaction rule for the graph coloring system

Two data types are used in this caster, but the LOD of type **vertex** is not defined because it is zero at any time. The LOD of type **edge** can be defined in an informal form as follows:

$$O_{edge}(x) = \begin{array}{ll} 1 & \textbf{if } x.next[0] \neq x.next[1].color \\ 0 & \textbf{if } x.next[0] = x.next[1].color \end{array}$$

The same LOD can be expressed in SOOC-94 as follows:

```
(deforder ((e edge))                         ; Definition of a self order degree
  (if (eql (vertex-color (edge-vertex e 0))
           (vertex-color (edge-vertex e 1)))
    0
    1))
```

The informal definition means that the LOD is equal to 0 if the head and tail of the edge are vertexes with the same color, and equal to 1 if they are with different colors. The latter is greater because it is a better (higher order) state. Expressions, *x.next*[0].*color* and *x.next*[1].*color,* represent the internal state, named *color*, of the objects that are pointed by the links, *next*[0] and *next*[1], from object x. The subscripts of the links are usually hidden, but they are specified here because they must be distinguished each other.

The nature of computation in SOOC-94 is non-terminating and user input is always acceptable. Thus, there is no need for the system to test the termination of computation. However, user may want to make sure the correctness of the solution computed using SOOC-94. For this purpose, the user can define a rule with **`depth-first`** option as follows.

```
(defrule (check-color :strategy depth-first)
   (var C E V1 V2)
   (exist E edge (:vertex V1) (:vertex V2))
   (exist V1 vertex (:color C))
   (exist V2 vertex (:color C))        ; The color of V1 and V2 are the same.
   -->
   ; Appropriate actions to be taken.
   )
```

This option means that the system must prove that there is no set of objects that can be matched, by testing the rules systematically (in depth-first order) instead of testing them randomly.[1]

4.2 Execution of the coloring system

The rule and LOD are coded in SOOC-92, a preliminary version of SOOC-94. The system is applied to the map of the main part of the USA, consists of 48 states [Tak 92]. Every time the system ran, a solution was found within reasonable time, though the execution time varies time to time because of the stochastic nature. The number of reactions, the number of LHS matching (including failed cases), and the execution time is measured on SOOC-92 interpreter on Macintosh Common Lisp on Macintosh Quadra 700 Computer. Their averages are 940, 34729 and 6.01 seconds.

Reactions occur successively so that the IODs increase, when the caster shown above is run. However, a reaction may decrease LODs of the neighboring edges. The sum of LODs of all the

---

[1] The user may specify this option in the coloring rule. Then there will be no need to supply a separate rule for testing the correctness. However, then the computation becomes non-random, and thus a limit cycle (looping) may be caused.

objects in WM is called *global order degree* (GOD). The GOD is not linearly increased, and the system does not necessarily find a solution in finite time, though the average time is finite. An example transition path of GOD is shown in **Figure 6**. The final value of the path is the maximum value of GOD, 105, in the USA map. A distribution of total number of matchings, which is roughly proportional to the computation time, is shown in **Figure 7**. The frequency decreases almost exponentially when the total number of matchings is more than 30,000.[1] Although this means that there is no upper limit in computation time, the probability decreases very fast.

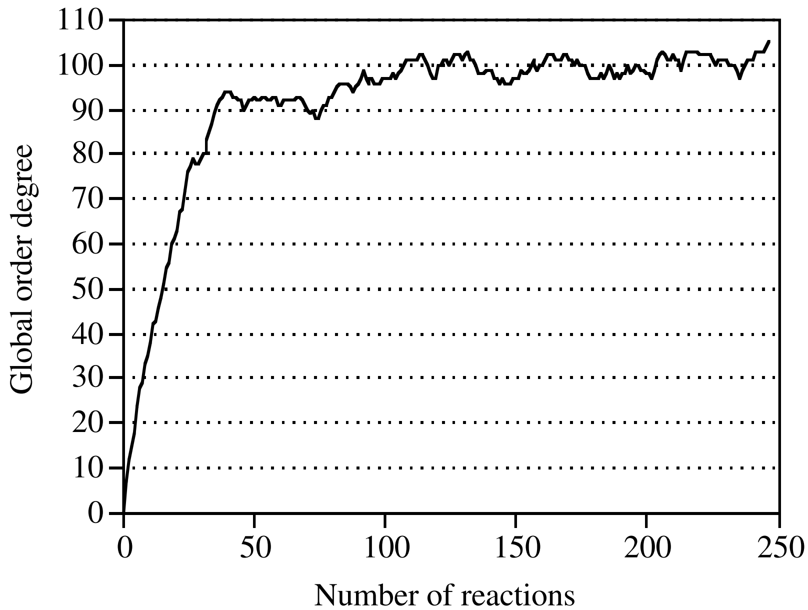


Figure 6. An example transition path of global order degree

[1] This distribution and the GOD transition can be explained using Markov chain model [Kan 93a].

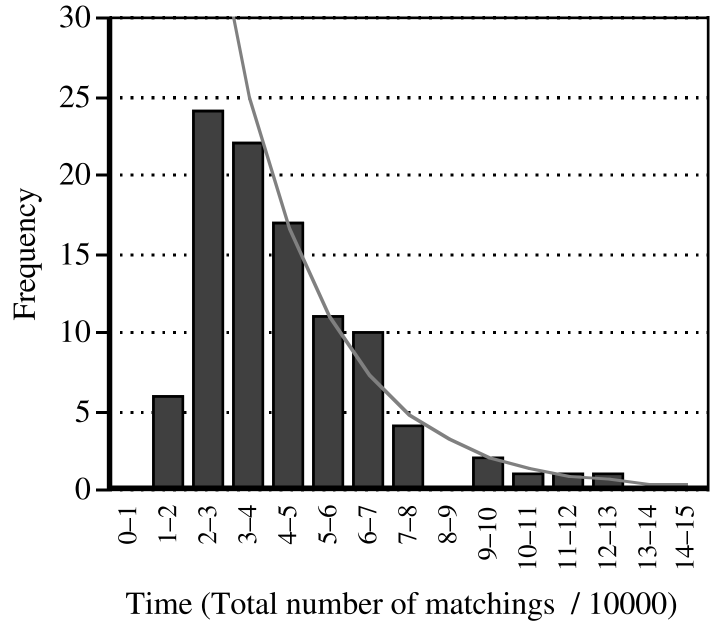


Figure 7. Distribution of computation time

5. Satisfaction of requirements from the real-world computing

Whether SOOC-94 meets the requirements explained in Section 2 is verified, compared to conventional programming languages, in this section. However, SOOC-94 is currently applied only to closed and relatively simple problems such as graph coloring problems, though most of them are NP-hard. The statements in this subsection must be proved in open and more complex applications in future.

(1) Openness to the environment

This requirement can relatively easily be satisfied when using SOOC-94, because WM (working memory) is always open to the environment. Input from the environment is performed by creating objects in WM, output to the environment is performed by moving (or removing) data from WM, and they are performed by a separate process in SOOC-94. Thus the system can always accept input from the environment and respond to it.[1] In addition, the

[1] However, the current impelmentation, SOOC-92, has not yet satisfied this requirement, because it is a strictly sequential implementation.

execution does not terminate conceptually. Thus, the system waits for input even when there is currently no more task to perform.

For example, vertexes and edges can be freely added or removed dynamically in the coloring system, and then the system finds a new solution correctly. If there are vertexes named V1 and V2, which are not connected, the user can input the following command at any time, and then, a new edge that connects V1 and V2 is created.

```
(make edge :vertex V1 :vertex V2)
```

If vertexes V1 and V2 are connected, the user can input the following command at any time (the user does not have to wait), and then, the link between V1 and V2 is removed.

```
(remove edge :vertex V1 :vertex V2)
```

A vertex can be created or removed in the same way as an edge.

(2) Possibility of distributed/parallel processing (DPP)

This requirement is satisfied because DPP is theoretically and practically made possible by SOOC-94. DPP is theoretically made possible because DPP can be modeled by random walk, because nondeterminacy caused by DPP can be modeled by stochastic processes. If DPP is modeled by explicit parallelism or by precise logic, the model becomes too complicated for human to understand, and/or the requirements, such as computability from partial and local plan, are difficult to be satisfied.

DPP can be practically made possible by physically or logically distributing objects in SOOC-94. For example, objects may be distributed to parallel processors. They are possible in principle because computation is performed only using local information. However, concrete and better methods of object distribution must be found by future research, because load balancing and object migration are difficult problems to be solved in a distributed or parallel implementation of SOOC-94.

There are many languages that target DPP, of course. However, most of them target coarse-grained parallelism, in which agents are implemented using procedural programming.

Fine-grained parallelism is targeted in SOOC-94, as same as in CCAP (committed-choice and-parallel) languages, such as Concurrent Prolog [Sha 87].

(3) Respondability to nondeterminacy

This requirement can be satisfied by giving appropriate biases to random walk in SOOC-94. If an object in WM is nondeterministically changed by an environmental change or by an action of another agent, the new situation can be appropriately processed with changing the bias value. Thus, the mechanism of biased random walk also improves robustness. In the above example, if a new edge between V1 and V2 is added and the colors of V1 and V2 are the same, the system works again so that the new graph will be correctly colored. An object removal may also cause reactions in general, though the removal of an edge in the above example does not cause reactions. If the computation graph is strongly connected, the bias value change may cause a sequence of transitions to an expected state, regardless of the previous state. However, a further research on real-world computing systems is needed to realize the solution.

(4) Unfixed control flow and data flow

This requirement is satisfied by the mechanism of biased random walk. The system works only if the computation graph, where no sequential control flow nor data flow is given, is supplied. In addition, the system does not fall into a limit cycle (infinite loop) without specifying control flow because of the randomness, even if the computation graph is not designed so ingeniously and thus it is a strongly connected graph, which causes limit cycles if the computation is a systematic walk on the graph. It will also be possible to supply a computation graph partially in the initial state and to restructure it dynamically in a future version of SOOC-94. On the contrary, in conventional programming languages, control flow and data flow must be given explicitly or implicitly even in logic programming or production system programming languages, and incomplete control flow easily causes a limit cycle.

(5) Computability from partial and local plan

This requirement is also satisfied by the mechanism of biased random walk. No global plan is given in SOOC-94.

(6) Real-time processing

This requirement is not satisfied in SOOC-94. Although high performance is needed for real-time processing, the performance of current implementation techniques, which are partially described in Section 6, is not good for real-time processing. The performance improvement is a future work.

(7) General-purposeness

This requirement is satisfied because SOOC-94 is based on production system, which is a general-purpose mechanism. Although it uses random walk, it is also possible for user to grasp the control almost completely. It is also possible for user to use Lisp expression within rules of SOOC-94, because it is built using Common Lisp. Thus, it is easy for user to use procedural expression partially, and it makes SOOC-94 more general-purpose.

6. Implementation Issues

The basic structure and several techniques of SOOC-94 implementation are discussed in this section.

The basic structure of SOOC-94 implementation is shown in **Figure 8**. The major parts of the system are a compiler and an interpreter. Both the compiler and interpreter are built on top of Common Lisp. Each rule or LOD declaration, which is given by the user, is compiled into a Lisp function by the SOOC compiler and each function is compiled to native code by the Lisp compiler, which is a part of Lisp system. The interpreter consists of a rule interpreter and an I/O interpreter. These interpreters work independently. The rule interpreter may consist of parallel processes. However, sequential implementation is asserted here. The compiled rules and LODs are called from the rule interpreter. The I/O interpreter is event-driven. If an input comes from the environment (from the user for example) through an input device or sensor, the I/O interpreter is invoked and WM is modified. If an object to be outputted is written into WM by the rule interpreter, the I/O interpreter is invoked and an output goes to the environment through an output device or actuator. Mutual exclusion is necessary when modifying WM because the rule and I/O interpreters work concurrently.[1]

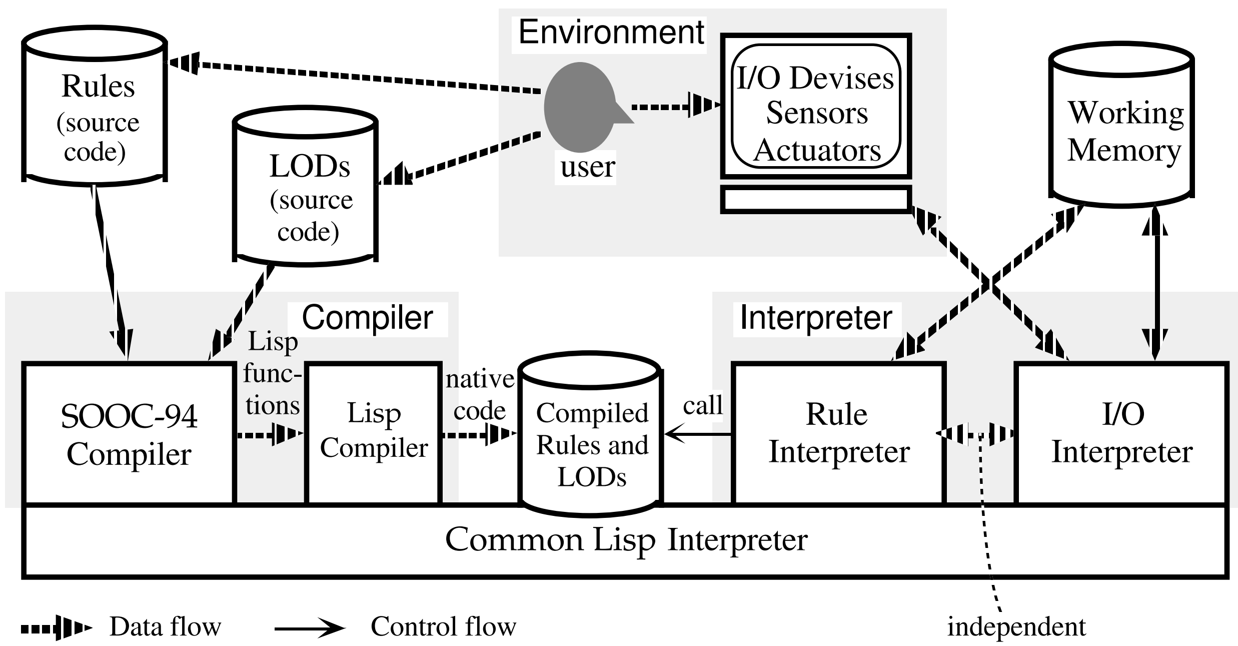


Figure 8. Basic implementation structure of SOOC-94

A set of objects and a rule to be reacted are selected in the following way in SOOC-94. The objects in WM are classified by their data types, and they are stored in extensible arrays. The interpreter selects a rule, and it selects an object from the array for each LHS condition. They are selected using random numbers. The LHS conditions are tested, the LODs are computed for each selected object or each pair of selected objects, and the reaction occurs if all the preconditions are satisfied. Thus the scheduling, or the conflict resolution, of concurrently activatable reactions is done by the first-fit basis. The interpreter repeats the above process until there is no set of objects and a rule, which can react. This implementation is completely different from

---

[1] The top level loop of the Lisp interpreter is replaced by the rule interpreter in SOOC-92, the current and preliminary implementation. Thus, rules and LODs can be defined through the rule interpreter, and the SOOC compiler is called from the rule interpreter. Rules and LODs cannot be defined unless the rule interpreter terminates, for the same reason.

conventional production systems. Thus, the method of improving performance of matching, called RETE [For 79], cannot be applied. Performance improvement is a future work.

Links are implemented using pointers in SOOC-94. For example, in the rule shown in Figure 5, variable **`V1`** is bound to the pointer to a vertex in the first LHS condition. Thus the interpreter can find the object that matches the second LHS condition immediately. Because conventional production systems do not support pointers, the same function needs much more computation time, where a "join" operation is needed.

Parallel implementation, which parallelizes the rule interpreter and may parallelize the I/O interpreter, has not yet tried for SOOC-94. However, CCM is promising as a model of massively parallel computation. Reactions can occur in parallel similar to chemical reactions. However, if two instances contain the same object, they may not be reacted in parallel.

7. Conclusion

This paper analyzes the requirements of real-world computing, and explains a computation language SOOC-94 based on computation model CCM, and its implementation techniques. If SOOC-94 is refined, it will be a powerful tool for researching methods of building real-world computing systems, and it may become a prototype of new real-world computation languages.

The main focuses of future work are as follows. First, a complete implementation of SOOC-94 must be developed, based on the preliminary version, SOOC-92. Second, the design and implementation of SOOC-94 should be refined so that they will be suitable for wide range of real-world applications. Development of new applications, especially, simple but real-world computing systems, is necessary for reaching this target.

Acknowledgment

The author wishes to thank the following persons. Mr. C. Bando of Hitachi Research Laboratory gave the author a chance to study the theories of self-organizing systems and to begin this research. The discussion with Dr. Hirokawa of Hitachi Advanced Research Laboratory provided the author with many useful ideas. Dr. R. Oka of Real World Computing Partnership gives the author a chance to continue this research.

References

[Ash 61] Ashby, W. R.: *An Introduction to Cybernetics*, Chapman & Hall, 1961.

[Bac 78] Backus, J.: Can Programming Be Liberated from von Neumann Style? A Functional Style and its Algebra of Programs, *Comm. ACM*, Vol. 21, No. 8, 1978.

[Bro 86] Brooks, R. A.: A Robust Layered Control System for a Mobile Robot, *IEEE J. Robotics and Automation*, Vol. RA-2, No. 1, pp. 14–23, 1986.

[For 79] Forgy, C. L.: *On the Efficient Implementation of Production Systems*, Ph.D. Thesis, Department of Computer Sci., Carnegie-Mellon University, 1979.

[For 81] Forgy, C. L.: *OPS5 User's Manual*, Technical Report CMU-CS-81-135, Carnegie Mellon University, Dept. of Computer Science, 1981.

[For 91] Forrest, Stephanie, ed.: *Emergent Computation*, MIT Press, 1991.

[Hak 78] Haken, H.: *Synergetics — An Introduction*, Springer-Verlag GmbH & Co. KG, 1978.

[Jan 80] Jantsch, E.: *The Self-organizing Universe: Scientific and Human Implications of the Emerging Paradigm of Evolution*, 1980.

[Kan 92] Kanada, Y.: Toward Self-organization by Computers, *33rd Programming Symposium*, Information Processing Society of Japan, 1992 (in Japanese).

[Kan 93a] Kanada, Y.: Symbol Processing as Stochastic Processes — Macroscopic Models of Computation Processes —, *Technical Report of IEICE*, COMP92–93, SS92-40, March 1993 (in Japanese).

[Kan 93b] Kanada, Y.: Features of Problem-Solving Method using Computation Model CCM, based on Production Rules and Local Evaluation Functions, *Summer United Workshops on Parallel, Distributed, and Cooperative Processing '93 (SWoPP '93)*, Tomonoura, Japan, 1993 (in Japanese).

[Kan 94] Kanada, Y., and Hirokawa, M.: Stochastic Problem Solving by Local Computation based on Self-organization Paradigm, *27th Hawaii International Conference on System Sciences,* 1994.

[Lan 89] Langton, C. G., ed.: *Artificial Life*, Addison-Wesley, Redwood City, CA, 1989.

[Lan 92] Langton, G. G. et al., ed.: *Artificial Life II*, Addison-Wesley, Redwood City, CA, 1992.

[Pri 89] Nicolis, G., and Prigogine, I.: *Exploring Complexity — An Introduction*, R. Piper GmbH & Co. KG Verlag, Munich, 1989.

[Sha 87] Shapiro, E., ed.: *Concurrent Prolog: Collected Papers*, Volumes 1 & 2, MIT Press, 1987.

[Tak 92] Takefuji, Y.: *Neural Network Parallel Processing*, Kluwer Academic Publishers, 1992.