Sorting on Skip Chains

Ajoy K. Datta, Lawrence L. Larmore
School of Computer Science, UNLV, USA
Firstname.Lastname@unlv.edu

Stéphane Devismes
VERIMAG, Université Joseph Fourier, Grenoble, France
Stephane.Devismes@imag.fr

Abstract—We introduce a generalization of the distributed sorting problem on chain network. Our problem consists of sorting values in processes that are separated from each other by any number of intermediate processes which can relay values but do not have their own values. We solve this problem in a chain network by proposing a silent self-stabilizing distributed algorithm.

I. INTRODUCTION

Sorting values on a chain of processes is a well-known problem, and a number of algorithms has been published [1], [2]. We propose a generalization of this problem, where the processes that have values, called major processes, are separated from each other by any number of intermediate processes, called relay processes, which do not have their own values, although they can read and write the major values while doing their job of relaying those values.

More precisely, we consider a chain network of processes. Some of those processes, including the two end processes, are major processes, and the rest are relay processes. We call this structure a skip chain. The problem is then to sort the values held by the major processes. We call this problem the skip chain sorting problem.

Contributions. We give a silent self-stabilizing distributed algorithm for the skip chain sorting problem. Our algorithm is written in the locally shared memory model and works under an unfair daemon. Its stabilization time is $O(md)$ rounds, where $m$ is the number of major processes and $d$ is the maximum number of processes in the chain from one major process to the next. Note that $md = O(n)$ if the spacing between major processes is roughly equal.

Related Works. To the best of our knowledge, there is no prior work on the skip chain sorting problem. However, note that the distributing sorting has been investigated in numerous papers, e.g., [2], [3], [1], [4].

Self-stabilization [5], [6] is a versatile property, enabling an algorithm to withstand transient faults in a distributed system. A distributed algorithm is self-stabilizing if, after transient faults hit the system and place it in some arbitrary global state, the system recovers without external intervention in finite time. In [4], asymptotically space and time optimal self-stabilizing sorting algorithms for oriented chains were given.

Roadmap. In the next section, we define the model used throughout this paper. We also define the specification of skip chain sorting problem in this section. Our self-stabilizing skip chain sorting algorithm is proposed in Section III. Due to lack of space, the correctness proof has been omitted, see the technical report online for details:

www-verimag.imag.fr/~devismes/WWW/rapports/trSkip.pdf

II. PRELIMINARIES

Computational model. We consider networks of chain topology whose two extremities are distinguished as Left and Right. Each process of the chain has the finite set of shared variables (henceforth, referred to as variables) whose domains are finite. A process $P$ can read its own variables and that of its neighbors, but can write only to its own variables. Each process writes its variables according to its program. We call algorithm a collection of programs, each one operating on a single process. The program of each process is a finite set of guarded commands or actions: If $\langle$guard$\rangle$ then $\langle$statement$\rangle$. The guard of an action in the program of a process $P$ is a Boolean expression involving the variables of $P$ and its neighbors. The statement of an action of $P$ updates one or more variables of $P$. An action can be executed only if it is enabled, i.e., its guard evaluates to true. A process is said to be enabled if at least one of its actions is enabled. The evaluations of all guards and executions of all statements of those actions are presumed to take place in one atomic step; this model is called composite atomicity [6].

The values of the variables at some process $P$ define the state of $P$. A configuration is an instance of the states of all processes. Let $\rightarrow$ be the binary relation over configurations such that $\gamma \rightarrow \gamma'$ if and only if it is possible for the network to change from configuration $\gamma$ to configuration $\gamma'$ in one step of the algorithm. A computation is a maximal sequence of configurations $\varrho = \gamma_0\gamma_1\ldots\gamma_i\ldots$ such that $\gamma_{i-1} \rightarrow \gamma_i$ for all $i > 0$. The term “maximal” means that the computation is either infinite, or ends at a terminal configuration in which no action of any process is enabled. Each step $\gamma_i \rightarrow \gamma_{i+1}$ consists of one or more enabled processes executing an action.

We assume that each step from a configuration to another is driven by a scheduler, also called a daemon. If one or more processes are enabled, the scheduler selects at least one of these enabled processes to execute an action. We assume that the scheduler is also unfair, meaning that, even if a process $P$ is continuously enabled, $P$ might never be selected by the scheduler unless $P$ is the only enabled process.

We say that a process $P$ is neutralized in the step $\gamma_i \rightarrow \gamma_{i+1}$ if $P$ is enabled in $\gamma_i$ and not enabled in $\gamma_{i+1}$, but does not execute any action between these two configurations. The
neutralization of a process represents the following situation: at least one neighbor of \( P \) changes its state between \( \gamma_i \) and \( \gamma_{i+1} \), and this change effectively makes the guard of all actions of \( P \) false.

We use the notion of round, defined by Dolev et al in [7]. The first round of a computation \( \varrho \), noted \( \varrho' \), is the minimal prefix of \( \varrho \) in which every process that is enabled in the initial configuration either executes an action or becomes neutralized. Let \( \varrho'' \) be the suffix of \( \varrho \) starting from the last configuration of \( \varrho' \). The second round of \( \varrho \) is the first round of \( \varrho'' \), the third round of \( \varrho \) is the second round of \( \varrho'' \), and so forth.

**Self-stabilization and Silence.** A configuration conforms to a predicate if the predicate is satisfied in the configuration; otherwise the configuration violates the predicate. By this definition, every configuration conforms to the predicate TRUE and none conforms to the predicate FALSE. Let \( R \) and \( S \) be predicates on configurations of the algorithm. Predicate \( R \) is closed with respect to the algorithm if every configuration of any computation of the algorithm that starts at a configuration conforming to \( R \) also conforms to \( R \). Predicate \( R \) converges to \( S \) if \( R \) and \( S \) are closed, and every computation starting from a configuration conforming to \( R \) contains a configuration conforming to \( S \).

A distributed algorithm is self-stabilizing with respect to a predicate \( R \) if and only if TRUE converges to \( R \). In that case, any configuration that satisfies \( R \) is said to be legitimate, and all other configurations are called illegitimate.

We say that an algorithm is silent, if all its computations are finite. In other words, starting from an arbitrary configuration, the network will eventually reach a configuration where no process is enabled.

**Formal Statement of the Problem.** We are given a chain of processes. Some of those processes, including the two end processes (which we call Left and Right) are major processes, and the rest are relay processes. We call this structure a skip chain. We assume that only major processes have values, and the problem is to sort those values. The specification of the skip chain sorting problem is given below.

1) In an arbitrary configuration of a skip chain, there is a canonical value \( V(x) \) associated with each major process \( x \). This value may or may not be stored at \( x \).
2) At each step, the multiset of canonical values does not change, although the canonical values of two different major processes can be exchanged.
3) Every computation eventually results in a legitimate configuration, where the following conditions hold:
   a) The canonical values of the major processes are in increasing order from left to right.
   b) The canonical value of each major process \( x \) is stored at \( x \).
   c) No action is enabled.

We assume that \( n \) is the number of processes in the chain, \( m \) is the number of major processes, and \( d \) is the relay chain length, which is the maximum number of processes in the chain from one major process to the next. For example, if all processes are major processes, then \( d = 2 \), and \( d = n \) if only the two end processes are major.

**III. Skip Chain Sorting**

We give an algorithm, skip chain sort (SCS) essentially a distributed version of the well-known algorithm bubblesort, which satisfies the requirements listed above.

**A. High Level Overview of SCS**

SCS is self-stabilizing, which implies that it converges to a legitimate configuration regardless of the initial configuration. Given any skip chain \( S \), let \( C \) be the set of all configurations of SCS on \( S \). A certain subset \( N \subseteq C \) consists of what we call normal configurations. These configurations are those where the states of all processes are correct, except that the canonical values may not be sorted. \( N \) is closed under the actions of SCS and is an attractor of \( C \).

The first phase of SCS, which we call error correction, results in a normal configuration. The second phase of SCS sorts the canonical values of the major processes, and eventually halts in a legitimate configuration, where each major process stores its own canonical value, and no process is enabled to execute.

**Data Structure.** Every major process \( x \), except Left, contains two embedded relay processes, which we call \( x.L._{relay} \) (left embedded relay process) and \( x.R._{relay} \) (right embedded relay process). Left contains only one embedded relay process, \( Left.R._{relay} \). Note that at the end, each major node stores its canonical value in its right relay. We call the other relay processes free relay processes. If \( x \) is any process, then we define \( Right.Major(x) \) and \( Left.Major(x) \) to be the nearest major processes to the right and left of \( x \) (if any) respectively.

If \( x \) is a major process, we define the right relay chain of \( x \) to be the chain of relay processes starting with \( x.R._{relay} \) and ending with \( Right.Major(x).L._{relay} \); the left relay chain of \( x \) is simply defined to be the right relay chain of \( Left.Major(x) \).

Two values can only be swapped by a major process if it holds both. If \( x \) is a major process and \( y = Right.Major(x) \), then \( V(x) \) and \( V(y) \) can be compared, and possibly swapped, by \( y \). The mechanism is to move \( V(x) \) along the right relay chain of \( x \) to \( y.L._{relay} \), while \( V(y) \) is at \( y.R._{relay} \). The values are then compared and possibly swapped. Afterward, the new value of \( V(x) \) can move back to \( x \), while the new value of \( V(y) \) can move to \( Right.Major(y) \). After at most \( \binom{d}{2} \) such comparisons, the canonical values will be sorted.

**Colors.** SCS uses color waves to control the movement of the values along the relay chains. A value moves to the left at the crest of a wave of color 0, and to the right at the crest of a wave of color 1. Two additional colors, 2 and 3, complete the color wave cycle to avoid ambiguity between waves. Additionally, there is an “error color,” E.

**Silence.** When the canonical values are sorted, a silence wave, generated by the rightmost process, moves to the left, eventually causing all execution to cease.
B. Variables and Functions of SCS

Each relay process \( x \) has the following variables.

1) \( x.\text{color} \in \{0, 1, 2, 3, \text{E}\} \), the color of \( x \). The color \( \text{E} \) indicates an error.

2) \( x.\text{value} \), of value type, the value of \( x \). If \( x \) is the canonical location of a major process \( y \), then \( x.\text{value} \) is the canonical value of \( y \).

3) \( x.\text{done} \), Boolean, meaning that \( x \) is done. This flag (roughly) indicates that the canonical values to the right of \( x \) have already been sorted. In a legitimate (silent) configuration, all relay processes are done and have color 0.

Recall that major processes contain relay processes (so-called embedded relay processes). Any embedded relay process maintains the aforementioned variables. However, an embedded relay process is controlled by the code of its master major process, and does not necessarily emulate the action of a free relay process.

In addition, each major process has the following variable:

4) \( x.\text{status} \in \{\text{S, U}\} \), the status of \( x \). \( \text{S} \) stands for swapped, and \( \text{U} \) stands for unswapped. This variable is only important during the error correction phase. If \( x.\text{status} = \text{S} \), it means that \( x.l.\text{relay}.\text{value} \) and \( x.r.\text{relay}.\text{value} \) have been exchanged. During error correction it can happen that \( x.r.\text{relay}.\text{value} \) is the canonical value of \( x \), but that \( x.l.\text{relay}.\text{value} \) is not the canonical value of any major process. In this case, the values should not be exchanged, but there is no way to prevent the swap if \( x \) is not aware of the fact that the configuration is still in an erroneous configuration. If \( x \) actually exchanges those values, \( x.\text{status} \leftarrow \text{S} \), and if it merely compares and does not exchange, \( x.\text{status} \leftarrow \text{U} \). Later, when \( x \) realizes that one of those values was not canonical, it will undo the swap if \( x.\text{status} = \text{S} \).

Left and Right Neighbors. Each process \( x \) sees each of its neighbors as a relay process. They are respectively denoted \( x.left \) and \( x.right \).

5) \( x.left \). If \( x \) is the leftmost process then \( x.left \) is undefined; otherwise, let \( y \) be the left neighbor of \( x \) in the chain.

Then \( x.left = \begin{cases} \emptyset & \text{if } y \text{ is a relay process} \\ y.r.\text{relay} & \text{if } y \text{ is a major process} \end{cases} \)

6) \( x.right \). If \( x \) is the rightmost process then \( x.right \) is undefined; otherwise, let \( y \) be the right neighbor of \( x \) in the chain.

Then \( x.right = \begin{cases} \emptyset & \text{if } y \text{ is a relay process} \\ y.l.\text{relay} & \text{if } y \text{ is a major process} \end{cases} \)

We also define the left and right neighbors of embedded relay processes.

7) If \( x \) is a major process other than \( \text{Left} \), we define \( x.l.\text{relay}.\text{left} \) to be \( x.left \), \( x.l.\text{relay}.\text{right} \) to be \( x.r.\text{relay} \), and \( x.r.\text{relay}.\text{left} \) to be \( x.l.\text{relay} \). If \( x \neq \text{Right} \), define \( x.r.\text{relay}.\text{right} \) to be \( x.right \).

When a relay process looks at its neighbors, it sees its relay chain neighbors, and cannot determine whether it is a free relay process or an embedded relay process. This way, all free relay processes can have identical programs. Likewise, a major process looking either left or right can only see the next relay process in the relay chain, and does not know whether it is a free relay process or an embedded relay process. Thus, all major processes, except for \( \text{Left} \) and \( \text{Right} \), can have identical programs.

Figure 1 illustrates the structure of a skip chain with three major processes. Each process sees each neighbor only as a relay process. For example, \( R_2.left = R_1 \), while \( R_2.right = M_2.l.\text{relay} \). Red dashed rectangles enclose the right relay chains of \( M_1 \) and \( M_3 \), while the blue dashed rectangle encloses the right relay chain of \( M_2 \). The left relay chains of \( M_2 \) and \( M_3 \) are the right relay chains of \( M_1 \) and \( M_2 \), respectively.

Validity.

8) If \( x \) is any free relay process, or the left relay process of any major process, we say that \( x \) is \( \text{valid} \), and write \( \text{Valid}(x) \), if \( x.left.\text{color} \) and \( x.\text{color} \) satisfy the conditions in any one of the eight rows of the following table.

<table>
<thead>
<tr>
<th>( x.left.\text{color} )</th>
<th>( x.\text{color} )</th>
<th>other condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>( x.left.\text{value} = x.\text{value} )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( x.left.\text{value} = x.\text{value} )</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>( x.left.\text{value} = x.\text{value} )</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>( x.left.\text{value} = x.\text{value} )</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>( x.left.\text{value} = x.\text{value} )</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>( x.left.\text{value} = x.\text{value} )</td>
</tr>
</tbody>
</table>

9) We define \( \text{Valid}(x.r.\text{relay}) \) to be false if \( x.r.\text{relay}.\text{color} = \text{E} \), true otherwise.

10) If \( x \) is a free relay process, we say that \( x \) is \( \text{protected} \) if \( x \) is not valid, \( x.\text{color} = 1 \), and \( x.right.\text{color} = 0 \). Note that if \( x \) is protected, then \( \sim \text{Valid}(x) \). This special designation is used to maintain Specification 2, as we shall see below. The basic idea is that \( x.\text{value} \) could be the canonical value of \( \text{Right_Major}(x) \), and there must not be deleted.

11) \( \text{Done}(x) \), Boolean. If \( x \) is any relay process, then

\[
\text{Done}(x) = \begin{cases} \text{TRUE} & \text{if } x = \text{Right.r.\text{relay}} \\
& \text{and } x.\text{color} = 0 \\
& \text{TRUE if } \text{Valid}(x), x.\text{color} = 0, \\
& x.right.\text{done}, \\
& \text{and } x.\text{value} = x.right.\text{value} \\
& \text{FALSE otherwise} \end{cases}
\]

C. Actions of SCS

We now list the actions of SCS.

Error Handling Actions. The actions listed here will clear away errors introduced by arbitrary initialization. However, the set of canonical values will not be altered.

Basic Idea: Invalidity caused by color or value error is cleared
from left to right in the skip chain, while errors in the done field are cleared from right to left.

Any relay process $x$ where either $x\text{.color}$ or $x\text{.value}$ is inconsistent with $x\text{.left}$ is invalid, $x$ changes its color to $E$ by executing Action 1, unless $x$ is protected. At a later step, after $x$ is sure that the error wave has been properly propagated to the right, $x$ will copy the color and value of $x\text{.left}$ by executing Action 5.

If $x$ is a free relay process, $x\text{.color} = 1$ and $x\text{.right}.color = 0$, we do not allow $x$ to execute Action 1, because that change could inadvertently change the canonical value of $Right Major(x)$. The reason is that $x$ could be the canonical location of $Right Major(x)$. If $x$ is not valid, it must then wait until $x\text{.right}$ executes Action 7a, copying the color and value of $x$. If $x$ is a canonical location, this action will shift that canonical location to $x\text{.right}$, permitting $x$ to execute Action 1 safely.

For any relay process $x$, the correct value of $Done(x)$ can be computed from $x\text{.color}$ and $x\text{.value}$, and also by the variables of $x\text{.right}$. In Action 6, $x\text{.done}$ is corrected to match $Done(x)$. This wave of corrections moves to the left.

After one pass of the color correction wave to the right followed by one pass of the done correction wave to the left, no further action of any process will create a new error of either kind.

**Actions:**

1) If $x$ is a free relay process, $\neg Valid(x)$, $x\text{.color} \neq E$, and $x$ is not protected, then $x\text{.color} \leftarrow E$.

2) If $x$ is a free relay process, $x\text{.color} = E$, $x\text{.right}.color = E$, and $x\text{.left}.color \neq E$, then $x\text{.color} \leftarrow x\text{.left}.color$ and $x\text{.value} \leftarrow x\text{.left}.value$.

3) If $x$ is a major process, $x\text{.l relay}.color \neq E$, $x\text{.left}.color = E$ we break into two cases.

   a) If $x\text{.status} = S$, $x\text{.l relay}.color = 0$, $x\text{.r relay}.color \in \{1, 2\}$, and $x\text{.l relay}.value < x\text{.r relay}.value$, then $x\text{.l relay}.color \leftarrow E$ and $x\text{.r relay}.value \leftarrow x\text{.l relay}.value$.

   b) Otherwise, $x\text{.l relay}.color \leftarrow E$.

   The purpose of Action 3a to reverse a possible exchange of the true canonical value of $x$ with a false canonical value of $Left Major(x)$. In this situation, $x\text{.status} = S$ indicates that the exchange has taken place, and thus it must be reversed by overwriting the false value of $V(x)$ by the true value.

4) If $x$ is a major process, $x\text{.r relay}.color = E$, and either $x = \text{Right}$ or $x\text{.right}.color = E$, then $x\text{.r relay}.color \leftarrow 0$.

5) If $x$ is a major process, $x\text{.l relay}.color = E$, and $x\text{.left}.color \neq E$, then $x\text{.l relay}.color \leftarrow x\text{.left}.color$ and $x\text{.l relay}.value \leftarrow x\text{.left}.value$.

   The following action corrects errors in the done fields of processes.

6) If $x$ is a relay process and $x\text{.done} \neq Done(x)$, then $x\text{.done} \leftarrow Done(x)$.

**Normal Actions.** The actions listed below have lower priority than those given above, i.e., no action listed in this section is enabled if any error handling action is enabled.

7) We first consider actions of a relay process $x$ which is either free relay process or the left relay process of a major process. Assume that $Valid(x)$ and $x\text{.left}.color \neq E$. If $x = y\text{.l relay}$, then an action of $x$ is in fact an action of $y$.

   a) If $x\text{.color} = 0$ and $x\text{.left}.color = 1$, then $x\text{.color} \leftarrow 1$, $x\text{.value} \leftarrow x\text{.left}.value$, and $x\text{.done} \leftarrow false$.

   b) If $x\text{.color} = 1$ and $x\text{.right}.color = 2$, then $x\text{.color} \leftarrow 2$.

   c) If $x\text{.color} = 2$ and $x\text{.left}.color = 3$, then $x\text{.color} \leftarrow 3$.

   d) If $x\text{.color} = 3$ and $x\text{.right}.color = 0$, then $x\text{.color} \leftarrow 0$, $x\text{.value} \leftarrow x\text{.right}.value$, and $x\text{.done} \leftarrow x\text{.right}.done$.

8) We now consider actions of a major process $x$. For each of these actions, we assume that $x$ is not enabled to execute any error correcting action.

   a) The following actions assume that $x \neq \text{Left}$.

      i) If $x\text{.l relay}.color = 0$, and $x\text{.left}.color = 1$, then $x\text{.l relay}.color \leftarrow 1$ and $x\text{.l relay}.value \leftarrow x\text{.left}.value$.

      ii) If $x\text{.l relay}.color = 2$, and $x\text{.left}.color = 3$, then $x\text{.l relay}.color \leftarrow 3$.

      iii) If $x\text{.l relay}.color = 1$ and $x\text{.r relay}.color \in \{0, 3\}$, then $x\text{.l relay}.color \leftarrow 2$.

      iv) If $x\text{.r relay}.color = 2$ and $x\text{.l relay}.color \in \{2, 3\}$, then $x\text{.r relay}.color \leftarrow 3$.

      v) If $x\text{.l relay}.color = 1$ and $x\text{.r relay}.color = 2$, then $x\text{.l relay}.color \leftarrow 2$ and $x\text{.r relay}.color \leftarrow 3$.

      vi) Assume $x \neq \text{Right}$. If $x\text{.l relay}.color = 3$ and $x\text{.r relay}.color = 0$, we break into three cases:

         A) if $x\text{.l relay}.value \leq x\text{.r relay}.value$ and $x\text{.r relay}.done$, then
A configuration is a relay process, which causes values of its embedded relay processes to be exchanged. The number inside each triangle is the color of the processes in those places at those times.

![Figure 2: Schematic of a computation of SCS. Major processes are represented by vertical lines. Colored lines trace the canonical locations of the values. Shading represents done. Black circles enclose points where values are exchanged.](image)

**Figure 2:**

We begin by giving a detailed description of computation of SCS where all configurations are normal, i.e., free of errors. In such a computation, Actions 1 through 6 are never enabled.

**Normal and Legitimate Configurations.** We define a configuration to be normal if

1) all relay processes are valid,
2) \(x\).done = Done\((x)\) for every relay process \(x\).

We say that a relay process is finished if \(x\).color = 0 and \(x\).done. We define a configuration to be legitimate if all relay processes are finished, and if values are sorted in left-to-right order; that is, if \(x\) is a major process and \(y = \text{Right}_\text{Major}(x)\), then \(V(x) \leq V(y)\).

**Canonical Locations and Canonical Values.** We now give a precise definition of Location\((x)\), the canonical location of a major process \(x\). The canonical location is a relay process, and the value of that relay process is \(V(x)\), the canonical value of \(x\).

We define the right valid chain of a major process \(x\) to be the maximum prefix of the right relay chain of \(x\) which
consists entirely of valid relay processes. For example, the right valid chain equals the right relay chain if it contains no invalid relay process, and it is empty if \( \neg \text{Valid}(x.r_{\text{relay}}) \). We define the relay process \( \text{Right Location}(x) \) as follows.

1) If \( x.r_{\text{relay}} \) is not valid, then \( \text{Right Location}(x) = x.r_{\text{relay}} \).
2) \( \text{Right Location}(x) \) is the rightmost process of color 1 in the right valid chain of \( x \), if there is any such process.
3) \( \text{Right Location}(x) \) is the leftmost process of color 0 in the right valid chain of \( x \), if there is any such process, and if there is no process of color 1 in the right valid chain of \( x \).
4) In all other cases, \( \text{Right Location}(x) \) is the rightmost process in the right valid chain of \( x \).

If \( x \) is a major process, we define \( \text{Off Side}(x) \), Boolean, to be true provided all following conditions hold, and false otherwise.

1) \( x \neq \text{Left} \)
2) \( x.r_{\text{relay}}.\text{color} \in \{1, 2\} \)
3) \( x.\text{status} = \text{S} \)
4) \( x.l_{\text{relay}}.\text{value} < x.r_{\text{relay}}.\text{value} \)
5) \( x.l_{\text{relay}}.\text{color} = 0 \)
6) There is some invalid process in the left relay chain of \( x \).
7) Let \( y \) be the rightmost invalid process in the left relay chain of \( x \). Then \( y.\text{color} \in \{0, 3\} \).

\( \text{Off Side}(x) \) holds if \( V(x) \) has been swapped with a value which is not the canonical value of \( \text{Left Major}(x) \). This could happen before the errors are cleared out.

We now define \( \text{Left Location}(x) \) and \( \text{Location}(x) \) for a major process \( x \) as follows.

\[
\text{Left Location}(x) = \begin{cases} 
  x.l_{\text{relay}} & \text{if } \neg \text{Valid}(x.l_{\text{relay}}) \\
  \text{undefined} & \text{if } \neg \text{Off Side}(x) \\
  \text{leftmost valid process of color 0} & \text{in the left relay chain of } x \\
  \text{otherwise}
\end{cases}
\]

\[
\text{Location}(x) = \begin{cases} 
  \text{Left Location}(x) & \text{if } \text{Off Side}(x) \\
  \text{Right Location}(x) & \text{otherwise}
\end{cases}
\]

\[
V(x) = \text{Location}(x).\text{value}
\]

**Color Cycles.** A color cycle consists of four waves along a relay chain. Suppose that \( x, y, \) and \( z \) are major processes, where \( y \) and \( z \) are the nearest major processes to the right of \( x \) and \( y \), respectively. For convenience, assume that none of those three are end processes.

We will start in a configuration where \( y.l_{\text{relay}}.\text{color} = 3 \) and \( y.r_{\text{relay}}.\text{color} = 0 \). Thus, the canonical locations of both \( x \) and \( y \) are at \( y \)'s relay processes. After the two values at \( y \) are compared and possibly exchanged, by Action 8(a)viA, 8(a)viB or 8(a)viC, the (possibly exchanged) values are carried to the left of a 0 wave starting at \( y.l_{\text{relay}} \), and to the right at the crest of a 1 wave starting at \( y.r_{\text{relay}} \). When the 0 wave reaches \( x \), it will wait for a 3 wave to reach \( x \) from its left, after which the two values will compare and possibly exchange, sending a 1 wave back to \( y \). When the 1 wave reaches \( z \), it will wait for a 2 wave to reach \( z \) from its right, after which \( z \) will execute Action 8(a)v, which starts a 2 wave back to \( y \). The canonical location of \( y \), however, remains at \( z \).

When the 1 wave from \( x \) reaches the 2 wave from \( z \) at \( y \), \( \text{Location}(x) = y.l_{\text{relay}} \) and \( \text{Location}(y) = z.r_{\text{relay}} \). Values do not compare and exchange at \( y \). Rather, \( y \) executes Action 8(a)v, sending a 2 wave from \( y \) to \( x \) and \( x \) a 3 wave from \( y \) to \( z \), and the canonical locations do not change. When the 2 wave reaches \( x \), it will eventually return as a 3 wave, and when the 1 wave reaches \( z \), it will eventually return as a 0 wave, carrying \( \text{Location}(y) \) back to \( y.r_{\text{relay}} \) with a possibly new canonical value \( V(y) \). This completes the color cycle.

If \( x = \text{Left} \) or \( z = \text{Right} \), the color cycle is essentially the same. The actions of these processes are the same or simpler versions of the actions of an interior major process.

**Figure 3** illustrates how color cycles work.

**Figure 3:** Partial computation of SCS. Red circles enclose values of \( M_i \), for \( i = 1, 3, 5 \), in their canonical locations; blue circles for \( i = 2, 4, 6 \). The canonical location of \( V(M_i) \) can change, but, if the configuration is normal, stays within the right relay chain of \( M_i \).

Finally, **Figure 4** shows each normal action in greater detail. The done fields are not shown in that figure, except for Action 8(a)viA.
when it encounters the larger value F. Starting at that point, the finishing wave again moves to the left until it meets the larger value G at the fourth (middle) major process. The wave then retracts all the way back to Right again, and then moves to the left again, eventually reaching Left.

E. Detailed Overview of Error Correction Computation of SCS

We now give a detailed description of error correction computation of SCS. Starting from an arbitrary configuration, SCS will execute Actions 1 through 6 until a normal configuration is reached, after which the computation of SCS is as described in Section III-D.

The error correcting actions, namely Actions 1 through 6, are designed to meet Specification 2.

If there is an invalid item in a relay chain, the invalidity will propagate to the right, eventually terminating at x.relay for some major process x. When a relay process sees that it is invalid, it changes its color to E, causing its right neighbor to become invalid as well. We make one exception to that rule; if a process is invalid, but protected, it does not change color until it is no longer protected. If a process x is protected, then x.color = 1 and x.right.color = 0. In this case, x is not the canonical location of y = Right_Major(x), and if it changed its color to E, that canonical location could suddenly jump from the right relay chain of Right_Major(x) to x.right, possibly violating Specification 2 of the skip chain sorting problem. To prevent this jump, we do not allow x to change its color to E until its right neighbor has color 1. The role of protected process moves rightward along the chain until it reaches the rightmost free relay process of the relay chain, after which execution of Action 3 resolves the problem.

The problem described above is caused by the possibility that is possible that y executed Action 8(a)viC or 8(d)iiiB before the invalidity wave reaches y. At that point, Location(y) will switch sides, from the right of y to the left side of y. When the invalidity wave reaches y, Location(y) is reset to the right of y, by y executing Action 3a. The purpose of the register y.status is to tell whether it has executed Action 8(a)viC or 8(d)iiiB.

**References**