6

Hypersequential Programming

— A Novel Paradigm for Concurrent Programming —

Naoshi Uchihira, Shinichi Honiden, and Toshibumi Seki
Systems & Software Engineering Laboratory, Toshiba Corporation
70, Yanagi-cho, Saiwai-ku, Kawasaki 210, Japan
Telephone: +81-44-548-5690, *Fax:*+81-44-520-5855, *E-mail:* uchi@ssel.toshiba.co.jp

Abstract

This paper proposes *hypersequential programming* which is a novel paradigm for concurrent programming to ease the difficulty of concurrent programming and make the concurrent program highly reliable. The difficulty of concurrent programming is due mainly to its nondeterminism; nondeterminism being the purpose of the concurrent program. We classify nondeterminism into 3 types: *intended*, *harmful*, and *persistent* nondeterminism. In traditional concurrent programming, a programmer first designs and implements programs so as to maximize concurrency, which may include the 3 types of nondeterminism. He then tries to detect harmful nondeterministic behaviors by testing and debugs them. However, it is actually very hard to remove all harmful nondeterministic behavior. On the contrary, in hypersequential programming the concurrent program is first serialized to remove all types of nondeterminism, and then the programmer tests and debugs it as a sequential program. Finally, it is parallelized by restoring only intended and persistent nondeterminism. With hypersequential programming, a highly-reliable concurrent program can be developed because the injection of harmful nondeterminism is precluded. This paper shows the generic concept and a simple embodiment of hypersequential programming.

Keywords

concurrent programming, nondeterminism, serialization, parallelization, dependence analysis, highly-reliable program, hypersequential programming, default sequential principle

1 INTRODUCTION

Generally speaking, we find it more difficult to develop concurrent programs than we do sequential programs. In testing and debugging of concurrent programs, the combination of data and timing variations causes an explosive increase of behavior complexity and often produces unexpected (probably harmful) behaviors. Moreover, concurrent programs do not always have reproducible behavior (McDowell and Helmbold, 1989).

These difficulties are due mainly to their capricious (i.e., nondeterministic) behaviors. Nondeterminism can be classified into the following 3 types.

- **Intended nondeterministic behavior:** Nondeterministic behavior which the programmer intends to implement.
- **Harmful nondeterministic behavior:** Nondeterministic behavior which the programmer does not intend to implement and does not expect.

I. Jelly et al. (eds.), *Software Engineering for Parallel and Distributed Systems*

- **Persistent nondeterministic behavior:** Nondeterministic behavior which is race-free, that is, has no effect on the results.

In conventional concurrent programming, the programmer tries to detect and remove harmful nondeterministic behavior in testing and debugging. However, it is actually very hard to remove all harmful behavior by testing.

This paper proposes a novel programming paradigm which is the reverse of the conventional programming and is applicable to actual concurrent programs. In hypersequential programming, all types of nondeterminism are removed at first by serialization, whereas only harmful nondeterminism is removed in the case of conventional programming. Then the programmer tests and debugs the serialized program in the conventional way. Finally, intended and persistent nondeterminism is restored by parallelization. While serialization and parallelization should be computer-aided, testing and debugging are basically done in the conventional way. Hypersequential programming makes concurrent programming as easy as sequential programming because testing and debugging are done for serialized programs, and thus high productivity and high reliability can be achieved.

The remainder of the paper is organized as follows. Section 2 explains three types of nondeterminism by example, and illustrates a concept of hypersequential programming. Then, we introduce a generic procedure of hypersequential programming and fundamental techniques for it in Section 3. Section 4 shows a concrete embodiment of hypersequential programming using a simple example. Finally, Section 5 mentions related works, and conclusions are presented in Section 6.

2 NEW PARADIGM FOR CONCURRENT PROGRAMMING

This section explains the concept of hypersequential programming. First, we classify nondeterminism of concurrent programs into 3 types in detail. Then, we illustrate how hypersequential programming differs from conventional concurrent programming with regard to manipulation of nondeterminism.

2.1 Concurrency and Nondeterminism

Even when a concurrent program runs with the same input, its behavior can be different. This is nondeterminism. Nondeterministic behaviors can be classified into three types: *intended*, *harmful*, *persistent* (Uchihira and Honiden 1995). We explain them using a simple example shown in Figure 1. The example is a simple Ada-like concurrent program "*Seat Booking*", where two processes read/write a shared memory "*seat*" to reserve one seat. This program has the following nondeterministic behavior, which can be classified into 3 types.

- $\theta_1 = l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow m_1 \rightarrow m_2 \rightarrow m_5$
  Result: $status_1 = OK, seat = OCCUPIED, status_2 = NG.$
- $\theta_2 = m_1 \rightarrow m_2 \rightarrow m_3 \rightarrow m_4 \rightarrow m_5 \rightarrow l_1 \rightarrow l_2 \rightarrow l_5$
  Result: $status_1 = NG, seat = OCCUPIED, status_2 = OK.$
- $\theta_3 = l_1 \rightarrow m_1 \rightarrow l_2 \rightarrow m_2 \rightarrow l_3 \rightarrow m_3 \rightarrow l_4 \rightarrow m_4 \rightarrow l_5 \rightarrow m_5$
  Result: $status_1 = OK, seat = OCCUPIED, status_2 = OK.$
- $\theta_4 = \mathbf{l_1} \rightarrow \mathbf{m_1} \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow m_2 \rightarrow m_5$
  Result: $status_1 = OK, seat = OCCUPIED, status_2 = NG.$
- $\theta_5 = \mathbf{m_1} \rightarrow \mathbf{l_1} \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow m_2 \rightarrow m_5$
  Result: $status_1 = OK, seat = OCCUPIED, status_2 = NG.$
- .....

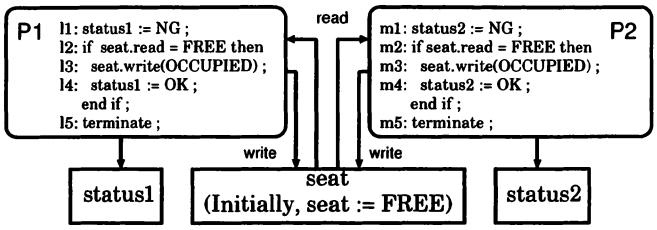

Figure 1 An example of a concurrent program

Intended nondeterminism The nondeterministic behaviors θ_1 and θ_2 derive different results: P_1 can book the seat ($status_1 = OK$) in θ_1, but cannot ($status_1 = NG$) in θ_2. However both are correct (intended behaviors).

Harmful nondeterminism The nondeterministic behavior θ_3 derives an incorrect result (double booking). Thus, this program has harmful nondeterminism.

Persistent nondeterminism The nondeterministic behaviors θ_4 and θ_5 have the same result because l_1 (write in $status_1$) and m_1 (write in $status_2$) are actions independent of each other. We call such a situation *persistent.*

Note that all intended nondeterministic behavior must be implemented, while persistent nondeterministic behavior is permitted but not necessarily implemented. Harmful nondeterministic behavior is wrongly injected when the programmer is implementing intended and persistent behaviors.

2.2 Paradigm Shift

Old Paradigm

In our observation of conventional concurrent program development, a programmer first tries to design and implement processes so as to maximize concurrency, which may include the 3 types of nondeterminism ($\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, ...$). He then tries to detect harmful nondeterministic behaviors (θ_3) in testing and debugs them by partially serializing the critical sections which interfere with each other using synchronization mechanisms (e.g., semaphore, rendezvous). Bugs due to harmful nondeterministic behavior often account for a considerable part of all timing bugs. In conventional concurrent programming, it is very difficult and requires heavy load to remove all harmful nondeterministic behavior for large-scale programs. Moreover, there is no assurance that all harmful behaviors are removed, and some bugs still remain more often than not.

This situation is illustrated in Figure 2. The dense tree denotes a concurrent program which has a lot of bugs (i.e., harmful behaviors). In conventional concurrent programming, bugs are removed one by one during testing and debugging. However, since the tree is dense, it is very hard to find all bugs, and some bugs remain.

New Paradigm

A novel programming paradigm, called *hypersequential programming*, is the reverse of the old paradigm. In a nutshell, the hypersequential programming consists of serialization and parallelization. In hypersequential programming the programmer first serializes the concurrent program to remove all types of nondeterminism, and then tests and debugs it as a sequential program. Finally, he parallelizes it by introducing only intended and persistent nondetermin-

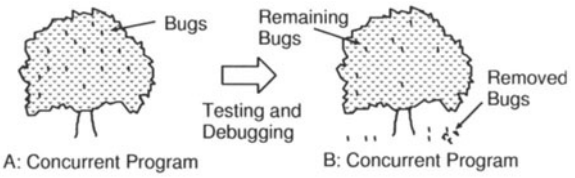

Figure 2 Conventional Concurrent Programming (Old Paradigm)

ism. We claim that hypersequential programming promotes a paradigm shift in concurrent programming.

In the case of the example (Figure 1), the program can be serialized by introducing a process priority ($P_1 > P_2$). The serialized program has only one behavior θ_1, which can be tested as easy as for a sequential program. Then, another intended behavior θ_2 is appended to the program. Finally, persistent nondeterministic behaviors $\theta_4, \theta_5, \ldots$ are restored by automatic parallelization.

This new paradigm is illustrated in Figure 3. After serializing the concurrent program represented by a dense tree A, only the trunk of the tree remains. This bare tree B illustrates the program stripped of all types of nondeterministic behaviors. It is easy to remove bugs from the bare tree as compared with the original dense tree, which implies that a lot of bugs (i.e., harmful nondeterministic behaviors) are removed together with serialization and further, the serialized program can be tested and debugged as easily as a sequential one for remaining bugs. However, this bare tree cannot fulfill the original requirements. It should be restored to its original dense condition. First, intended nondeterministic behaviors are explicitly introduced by the programmer. This program (we call it hypersequential program C) can fulfill the functional requirements but lacks the parallel speedup; thus the tree looks ill-formed. Then, persistent nondeterministic behaviors are restored. This introduction can be automatically done by using parallelization techniques. The programmer finally gets a new concurrent program D which may be slightly different from the original dense tree A but has no bugs.

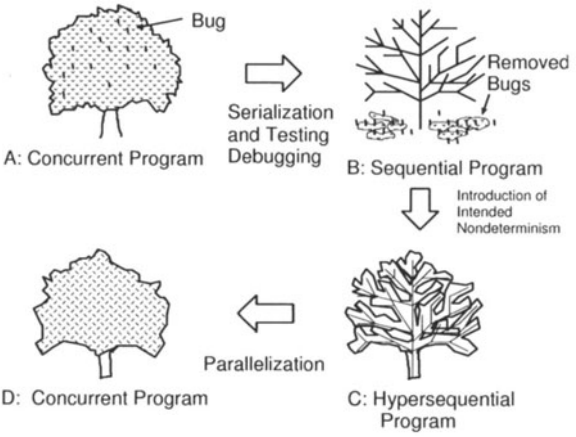

Figure 3 Hypersequential programming (New Paradigm)

3 HYPERSEQUENTIAL PROGRAMMING

Hypersequential programming consists of the following five steps.

Step 1: Modeling and Coding Model the target system as it is, using concurrency and nondeterminism naturally. Then code it with a concurrent programming language.

Step 2: Serialization (Projection) Serialize the concurrent program to remove all types of nondeterminism. The generated program can be viewed as a projection of the concurrent program onto a sequential program, which we call a *hypersequential program* *.

Step 3: Testing and Debugging
Test and debug the hypersequential program. Since the program is serialized, testing and debugging are as easy as with a sequential one.

Step 4: Introduction of Intended Nondeterminism
Introduce intended nondeterministic behaviors into the hypersequential program. After each introduction, the program should be tested again for added behaviors.

Step 5: Parallelization (Restoration)
Parallelize the program automatically by permitting persistent nondeterministic behaviors. This parallelization signifies the restoration of concurrency.

In a nutshell, a concurrent program is first projected into a sequential dimension which facilitates the programmer's job, and then the program is restored again into concurrent dimension. Each step is explained in detail in the following sections.

3.1 Serialization

The target concurrent program is serialized in the meta-level control and converted into a hypersequential program. There are several methods of the meta-level control for serialization as follows.

- **Serialization based on global priorities:** By introducing global priorities among processes, methods, or program sections, a concurrent program becomes deterministic for the same inputs. Note that the global priority control assumes a virtual single CPU environment; it is impractical to implement global control (i.e., global ordering of events) actually for parallel and distributed environments.
- **Serialization based on event histories:** The execution of a concurrent program can be recorded as an event history. A concurrent program can be executed deterministically under the control of the event history.
- **Serialization based on scenarios:** Instead of event histories, the programmer can specify execution order among program sections by giving a scenario.

Serialization techniques have been developed for debugging concurrent programs in order to solve the reproducibility problem (Bernstein and So, 1991). However, serialization information is used only for testing and debugging in these works, while serialization information (i.e., *default execution order* explained in 3.2) is also used for parallelization in hypersequential programming.

*A hypersequential program is a program which is serialized at the meta-control level and preserves the topology of the original concurrent program. After introducing intended nondeterminism in Step 4, the hypersequential program may have some partial concurrency.

3.2 Hypersequential Program

A hypersequential program is composed of *section information*, *section graph*, *program dependence graph*, and *serialization information*.

- **section information:** The original program (source code) is divided into program sections. A program section is a segment of a parallel program that is coded to be executed by one process. A program section may be a *basic block* which has no branching and synchronization except its start and end, and can be executed deterministically and be automatically extracted. The programmer can also specify a program section explicitly using some landmark point (we call it *section point*) as delimiter. In this case, program section may consist of several basic blocks and include some branching and synchronization, where process switching is allowed only at section points and prohibited during execution of the section.
- **section graph:** A section graph represents a topological control structure of the original program, where nodes correspond to program sections.
- **program dependence graph:** A program dependence graph (Ferrante, et al, 1987) represents both control dependence and data dependence between program sections in a single graph.
- **serialization information:** Serialization information specifies execution order among program sections. This execution order consists of two types of order: *a priori* execution order defined in the original program and *a posteriori* execution order given by serialization. The latter order is called *default execution order* which should be preserved as far as parallelization steps do not remove it explicitly. This execution order may be total order just after serialization, but will change to partial order after parallelization steps.

3.3 Introduction of Intended Nondeterminism

Introduction of intended nondeterminism is a new and important step which does not exist in the conventional programming. There are several methods to introduce intended nondeterminism.

- **Removing default execution orders:** The programmer can remove some default execution orders of the hypersequential program which generate intended nondeterministic behaviors. In other words, total ordering of sections is relaxed into partial ordering. Figure 4 illustrates a simple example of several removing default execution orders. Note that an a priori execution order cannot be removed.
- **Enumerating essential scenarios:** The programmer enumerates all essential scenarios, which correspond to test cases in conventional programming. The essential scenarios should be representative of all intended behaviors. A set of intended behaviors can be restored from these scenarios by parallelization.

3.4 Parallelization

Parallelization techniques have been extensively studied for compiler optimization for supercomputers (Zima and Chapman, 1990). In hypersequential programming, these parallelization techniques are used to detect persistent nondeterministic behaviors and restore them. This parallelization should preserve the original semantics, that is, it must not generate nondeterministic behavior which can produce results different from the original one.

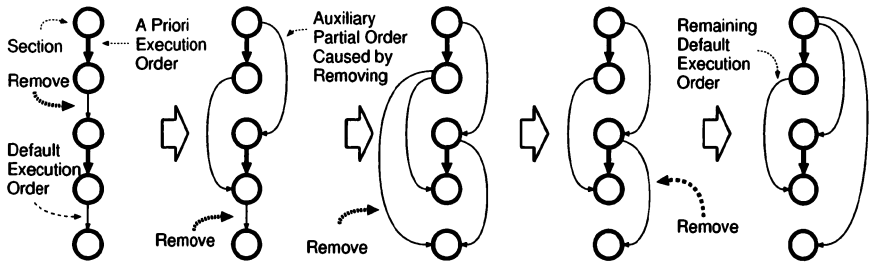

Figure 4 Removing Default Execution Orders

Concretely, a hypersequential program is parallelized as follows. First, precedence constraints between sections are automatically extracted from the dependence graph and the serialization information. If two sections have no precedence constraints, they can be parallelized. Then, default execution order between sections in the serialization information can be removed if they have no precedence constraints. Finally, a concurrent program is generated, where the remaining default execution order in the serialization information is preserved by inserting synchronization codes to implement it.† The resulting program involves only intended and persistent nondeterministic behaviors.

We want to emphasize that some of initial default execution orders by serialization still remain as far as they are not explicitly removed by introduction of intended nondeterminism or parallelization. These remaining default orders preclude the injection of harmful nondeterminism. We call it the *default sequential principle.*

4 SIMPLE EMBODIMENT

Since hypersequential programming is a rather conceptual paradigm, there are a variety of concrete procedures based on the concept. This section shows a simple embodiment of hypersequential programming using Petri nets.

4.1 Simple Example

We now explain hypersequential programming by a simple example. The target concurrent program consists of two processes P_1, P_2 on different processors, and two shared memories M_a and M_b. Although this program has no branch and loop, it can demonstrate essential features of hypersequential programming.

Step 1-1: Modeling and coding a target program

The target concurrent program $P = P_1 \| P_2$ is described as in Figure 5.

Step 2-1: Setting program sections

In this case, assume that each instruction forms one section. To simplify the description, an instruction code is itself used as a section ID.

†The remaining default execution order can be also implemented by run-time control instead of inserting synchronization codes into the program itself.

```
P1:                                          P2:
begin                                        begin
init1 ;     /* Initialize Memory Ma    */   init2 ;     /* Initialize Memory Mb    */
read1 ;     /* Read Data from Memory Mb */   read2 ;     /* Read Data from Memory Ma */
write1 ;    /* Write Data in Memory Ma  */   write2 ;    /* Write Data in Memory Mb  */
end                                          end
```

Be careful! M_a is accessed by $init_1, read_2, write_1$, and M_b is accessed by $init_2, read_1, write_2$.

Figure 5 Target Concurrent Program $P = P_1 \| P_2$ (Source Code)

Step 2-2: Serializing the concurrent program

The concurrent program P is serialized by introducing a process priority $P_1 > P_2$ which means P_1 is executed with the higher priority than P_2. In this case, the execution order of sections is represented as follows.

$$init_1 \rightarrow read_1 \rightarrow write_1 \rightarrow init_2 \rightarrow read_2 \rightarrow write_2$$

After serialization, the hypersequential program HSP is represented as shown in Figure 6, which consists of *section information*, *section graph*, *program dependence graph* and *serialization information*. The serialization information represents the above execution order of sections using Petri nets.

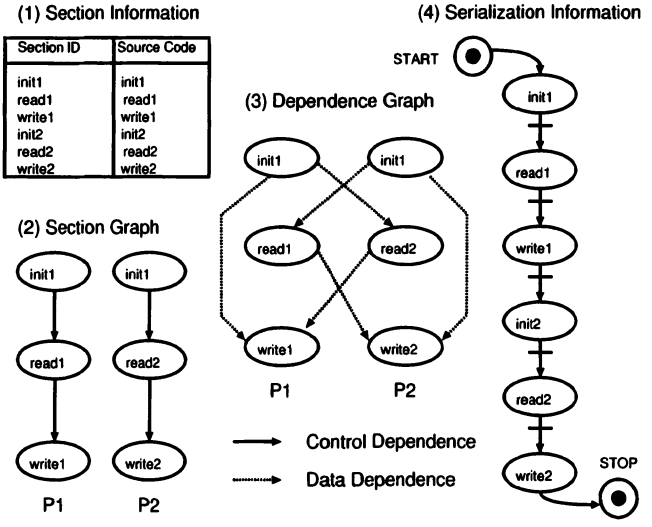

Figure 6 Hypersequential Program HSP

Step 3: Testing and debugging hypersequential program

The hypersequential program HSP is executed and tested by the programmer. If bugs are detected, the source code of sections is corrected. If a bug is related to the program structure, the original concurrent program is modified and serialized again. In this case, a bug such that $read_1$ accesses M_b before M_b is initialized is detected. Then the programmer debugs it by inserting synchronization commands (*send*, *wait*). The modified concurrent program is shown

in Figure 7, where $read_1$ accesses M_b after M_b is initialized. This program should be serialized and tested again.

```
P1:                                          P2:
begin                                        begin
init1 ;     /* Initialize Memory Ma    */   init2 ;     /* Initialize Memory Mb    */
wait(1) ;   /* Synchronization         */   send(1) ;   /* Synchronization         */
read1 ;     /* Read Data from Memory Mb */   read2 ;     /* Read Data from Memory Ma */
write1 ;    /* Write Data in Memory Ma */   write2 ;    /* Write Data in Memory Mb */
end                                          end
```

Figure 7 Modified Concurrent Program (Source Code)

Step 4-1: Introduction of intended nondeterminism

The Petri net representing the serialization information is displayed. The programmer introduces intended nondeterminism explicitly by removing default execution orders represented by arrows of Petri nets. In this case, a default execution order between $write_1$ and $read_2$ is removed (Figure 8(a)).

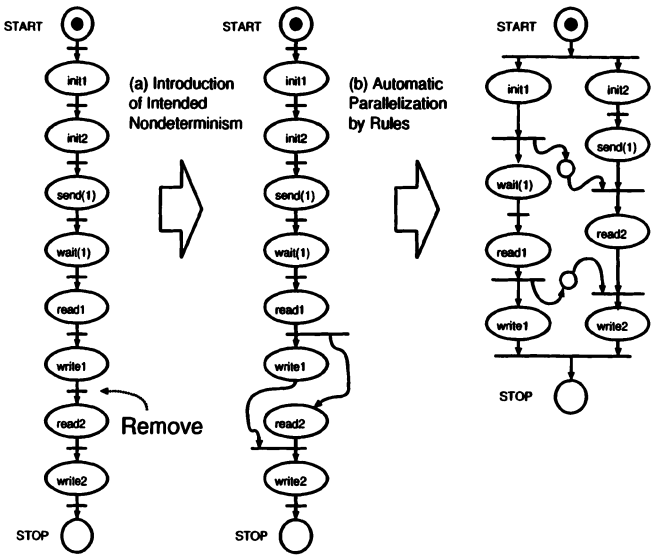

Figure 8 Introduction of Intended Nondeterminism

Step 4-2: Testing and debugging hypersequential program

The hypersequential program into which nondeterminism is introduced is executed and tested again. In this case, the program has two nondeterministic behaviors.

- $init_1 \rightarrow init_2 \rightarrow send(1) \rightarrow wait(1) \rightarrow read_1 \rightarrow \mathbf{write_1} \rightarrow \mathbf{read_2} \rightarrow write_2$, or
- $init_1 \rightarrow init_2 \rightarrow send(1) \rightarrow wait(1) \rightarrow read_1 \rightarrow \mathbf{read_2} \rightarrow \mathbf{write_1} \rightarrow write_2$.

If bugs are detected, the program should be corrected. In this case, these two behaviors are intended ones and no bugs are detected.

Step 5-1: Automatic parallelization

Precedence constraints among program sections can be automatically derived by analyzing the serialization information and the program dependence graph, especially data dependence among sections. Default execution orders with some precedence constraints should be preserved in parallelization because the execution result may change depending on the execution order of sections having data dependence. In this case, there are the following precedence constraints: $init_1 \rightarrow read_2$, $init_2 \rightarrow read_1$, $read_1 \rightarrow write_2$. For example, a value read by the process P_1 in $read_1$ is influenced by whether $write_2$ of the process P_2 occurs before or after $read_1$. Therefore, the default execution order $read_1 \rightarrow write_2$ in the serialization information should be kept in parallelization. Sections having no precedence constraints can be executed concurrently, and can be parallelized. For example, since two sections $read_1$ and $read_2$ have no precedence constraint, they can be parallelized, i.e., the default execution order $read_1 \rightarrow read_2$ is removed.

This parallelization can be done automatically by applying rules shown in Figure 9 to transform the target Petri net. Figure 8(b) shows a parallelized Petri net after rule application.

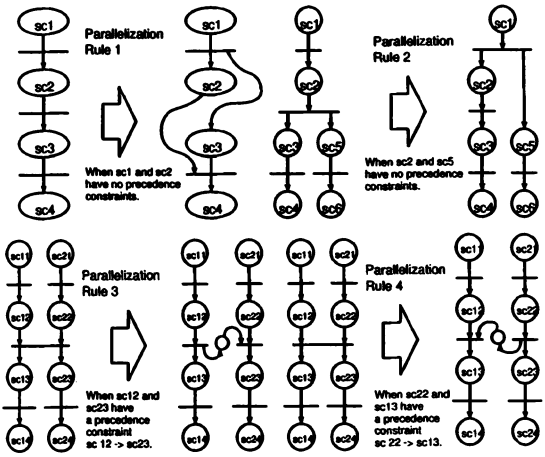

Figure 9 Rules for Parallelization

Step 5-2: Generation of Concurrent Program

The source code of the final concurrent program is generated from the hypersequential program after parallelization, where the remaining default execution order should be implemented. In this case, synchronization instructions (*send* and *wait*) are embedded in the source code for realizing the default execution order; $init_1 \rightarrow read_2$ and $read_1 \rightarrow write_2$. The generated concurrent program is shown in Figure 10.

4.2 Treatment of Branches and Loops

Ordinary programs have usually branches and loops. This section considers briefly treatment of branches and loops in hypersequential programming.

```
P1:                                          P2:
begin                                        begin
init1;      /* Initialize Memory Ma    */   init2;      /* Initialize Memory Mb    */
send(2);    /* synchronization         */   send(1);    /* synchronization         */
wait(1);    /* synchronization         */   wait(2);    /* synchronization         */
read1;      /* Read Data from Memory Mb */   read2;      /* Read Data from Memory Ma */
send(3);    /* synchronization         */   wait(3);    /* synchronization         */
write1;     /* Write Data in Memory Ma  */   write2;     /* Write Data in Memory Mb  */
end                                          end
```

Figure 10 Generated Concurrent Program (Source Code)

Steps concerning serialization, testing and debugging, and introduction of intended nondeterminism are done in the same way. Only the parallelization step requires additional consideration concerning branches and loops. In general, this parallelization has been well investigated and several techniques are available. In Petri-net-based parallelization, we use Petri net folding/unfolding techniques for branching. For example, the *rule*5-8 (Figure 11) shows parallelizing program fragments by folding/unfolding nets. With regard to loops, we basically adopt a hierarchical parallelization approach (Girkar and Polychronopoulos, 1992), in which loops are treated as single hierarchical sections and a program is represented as an acyclic section graph in each hierarchical level. Parallelization is done for each hierarchical level. Furthermore, loop unwinding rules are applied to promote parallelization effectively. The *rule*9 (Figure 11) is one of the loop unwinding rules.

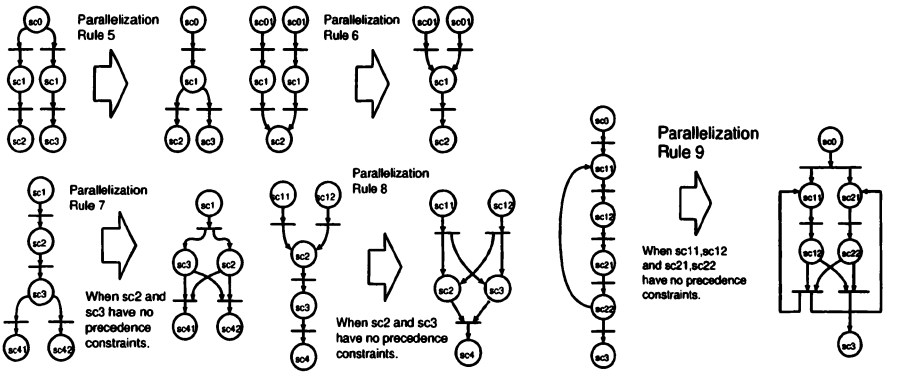

Figure 11 Rules for Parallelization (Branch and Loop)

5 RELATED WORKS

When developing concurrent programs, it is useful to test and debug them after serializing them and deleting their nondeterminism. For example, we usually test and debug concurrent programs in a single CPU environment acting as a pseudo-multi-CPU environment before doing so in an actual multi-CPU environment. Bernstein and So (1991) systematize this debugging know-how. They proposed a debugging method for concurrent programs by stepwise serializa-

tion. With regard to parallelization of sequential programs, a great deal of research has been done in the domain of compiler optimizations for parallel computers. Zima and Chapman (1990) summarized these techniques. Recently, automatic extraction of task-level parallelism has been studied by Girkar and Polychronopoulos (1992,1995). However, there are no reports of research into the combination of serialization and parallelization. The reader may ask why not starting with sequential program instead of a concurrent one which is serialized afterward. Answer is that topology of concurrent program is very natural for many target domains. In hypersequential programming, the topology of the original program is preserved at serialization, and it is restored at parallelization.

The concurrency control of database transactions (Bernstein and Goodman, 1981) is related to the techniques of hypersequential programming. The concurrency control is intended to remove harmful nondeterminism and leave intended and persistent nondeterminism as much as possible. However, it is intended only for database transactions, and not for ordinary concurrent programs. Hypersequential programming is aimed at testing and debugging of ordinary concurrent programs which may themselves have synchronization codes.

6 CONCLUDING REMARKS

Hypersequential programming can make concurrent programming as easy as sequential programming, and produce a highly reliable program. This paper shows the basic concept and approach of hypersequential programming and a simple embodiment of the concept. We think this concept is very general and there may be a lot of embodiments, and it can be applied to wide domains of concurrent programming. At the same time, since hypersequential programming is still in its infancy, a lot of techniques need to be developed in order to put hypersequential programming to practical use.

Acknowledgments: The concept of hypersequential programming is the fruit of a pooling of ideas and discussion in a certain research project being carried out by Toshiba corporation. We would like to thank our fellow researchers on the project: Hideaki Shiotani, Akihiko Ohsuga, Satoshi Itoh, Shinsuke Sawashima, Mitsuru Kakimoto, Yasuo Nagai, Yasuyuki Tahara, Katsuhiko Ueki, Keiichi Handa, and Tamiya Ochiai. We are also grateful to Sadakazu Watanabe and Kazuo Matsumura of the Systems & Software Engineering Laboratory, Toshiba Corporation, for their support throughout this work.

REFERENCES

McDowell, C.E. and Helmbold, D.P. (1989) Debugging Concurrent Programs, *ACM Computing Surveys*, Vol.21, No.4.

Uchihira, N. and Honiden, S. (1995) Compositional Adjustment of Concurrent Programs to Satisfy Temporal Logic Constraints in MENDELS ZONE, *28th Hawaii International Conference on System Science (HICSS)*.

Bernstein, D. and So, K. (1991) Debugging Parallel Programs by Serialization, *United States Patent* No. 5048018.

Ferrante, J., Ottenstein, K.J., and Warren, J.D. (1987) The Program Dependence Graph and Its Use in Optimization, *ACM Trans. on Programming Languages and Systems*, Vol.9, No.3.

Girkar, M. and Polychronopoulos, C.D. (1992) Automatic Extraction of Functional Parallelism from Ordinary Programs *IEEE Trans. Parall. Distrib. Syst.*, Vol.3, No.2.

Girkar, M. and Polychronopoulos, C.D. (1995) Extracting Task-Level Parallelism *ACM Trans. Prog. Lang. Syst.*, Vol.17, No.4.

Zima, H. and Chapman, B. (1990) *Supercompilers for Parallel and Vector Computers*, Addison-Wesley.

Bernstein, F.A. and Goodman, N. (1981) Concurrency Control in Distributed Database Systems, *ACM Computing Surveys*, Vol.13, No.2.