Software Maintenance Management Method for Large-scale Distributed Industrial Control Systems

Yoshiki Kakumoto

Eiji Nishijima

kakumoto@sdl.hitachi.co.jp

eiji@sdl.hitachi.co.jp

1st Research Dept., Systems Development Laboratory,

Hitachi, Ltd. 292, Yoshida, Totsuka, Yokohama, Kanagawa, 244-0817,

JAPAN

Kuniyuki Kikuchi *Senior Director*

ku-kikuchi@itg.hitachi.co.jp

Transportation Information Systems Dept.2, Information & Control Systems Division,

Hitachi, Ltd. 2-15-2, Konan, Minato, Tokyo, 108-6113,

JAPAN

Norihisa Komoda ***Professor***

komoda@ist.osaka-u.ac.jp

Dept. of Multimedia Eng., Graduate School of Information Science and Technology,

Osaka University 2-1, Yamadaoka, Suita, Osaka, 565-0871,

JAPAN

Abstract: Although the functions of subsystems in large-scale distributed industrial control systems seem to be identical, they differ slightly because they change when hardware is replaced and when the functions are upgraded. When new standard systems are generated by merging the differences between subsystems, subsystems become different when their programs are adjusted in accordance with the conditions under which each subsystem operates. Software maintenance therefore becomes very difficult. This paper proposes a two-phase software maintenance management assuring the integrity and consistency of the software in large-scale distributed industrial control systems. In the generation phase, individual and standard programs are managed separately and the subsystem is completely constructed according to the system definition file. In the execution phase, programs loaded into the execution environment are recorded in the registration data file in each subsystem. This is method increases the effectiveness and reliability of software maintenance.

Keywords: distributed control system, software maintenance, integrity, consistency

Yoshiki Kakumoto has been a researcher of Systems Development Laboratory, Hitachi, Ltd. since 1989. He is now a senior researcher in 1st Research Dept. He received the B. Eng. and M. Eng. degrees in Systems Science from Kyoto University in 1987, 1989 respectively. His research interests include transportation systems, control systems, and information processing systems.

Eiji Nishijima has been a researcher of Systems Development Laboratory, Hitachi, Ltd since 1986. He received the B. Eng. degree in Information Science and Technologies from Hitachi Keihin Institute of Technology. His research interests include train systems and information systems.

Kuniyuki Kikuchi has been an engineer of Information & Control Systems Division, Hitachi, Ltd. since 1979. He is now a senior director in Transportation Information Systems Dept.2. He received the M. Eng. degree in Gunma University in 1979. His research interests include transportation systems, control systems, and information systems.

Norihisa Komoda: has been a professor of the Department of Multimedia Engineering, Graduate School of Information Science and Technologies, Osaka University since 2002. He received the B. Eng. and M. Eng. degrees in Electrical Engineering and the Dr. Eng. from Osaka University in 1972, 1974, and 1982, respectively. His research interests include business information systems, electronic commerce systems, and knowledge information processing. Dr. Komoda received several awards such as IEEJ Technical Development Award, IEEJ in 2000.

1. Introduction

The subsystems in large-scale distributed industrial control systems differ from each other, are distributed over a wide area, and gradually change as a result of continuous maintenance. All of these things make software maintenance very difficult. A train-traffic-control system, for example, has many station-level subsystems doing the traffic control, several train-line-level systems managing the station-level subsystems, and a central supervising system [5]. Although a process computer executes the traffic control in each station-level subsystem, the functions of the station-level subsystems are a little different because the subsystems are

operating under different conditions. Each function is also continuously changing because hardware is replaced and because the function itself is being gradually upgraded. Since software developers cannot generate common programs for all parts of every subsystem, they generate temporary standard programs and install them in each subsystem. The maintenance of each subsystem must be performed in a way appropriate to the conditions under which that subsystem is operating, so the software of each subsystem becomes different.

Computer-Aided Software Engineering (CASE) tools [8] have a software version management function, and Concurrent Versions System (CVS) [10] is one version management tool widely used in developing open source software. These software management tools manage the sequential versions of software products. The existing software distribution tools [3] are mainly for personal computers and work stations. Since they deploy commercial software products to client machines uniformly, they can manage only those machines in which the same software products are distributed. These tools are thus not suitable for large-scale distributed industrial control systems where the software management is necessary for the individual program installed in the subsystem and the subsystem itself.

This paper proposes a two-phase software maintenance management method assuring the integrity and consistency of the software in large-scale distributed industrial control systems. In the generation phase, individual and standard programs are managed separately and the subsystem is completely constructed according to the system definition file. In the execution phase, programs loaded into the execution environment are recorded in the registration data file in each subsystem. This is method increases the effectiveness and reliability of software maintenance.

The rest of this paper is organized as follows. Section 2 describes features of large-scale distributed industrial control systems. Section 3 describes requirements for software maintenance. Section 4 describes the proposed maintenance management method. Section 5 discusses evaluation of the proposed methods. Section 6 concludes the paper.

2. Features of Target Systems

2.1. Functional Features

There are several types of distributed industrial control system. A purely distributed system is one in which not only devices but also machines are distributed [1], and a hybrid system is one in which a centralized machine controls distributed devices and controllers [4]. The recently introduced cellar manufacturing systems can be rapidly reconfigured in response to changes in market demand [9], and the autonomous components of holonic manufacturing systems [2] cooperate with each other to efficiently produce the wide variety of products needed to meet increasingly diverse demands.

The large-scale distributed industrial control system is a purely distributed one. Although the basic functions of its subsystems seem to be identical, they differ slightly according to the conditions under which the various subsystems operate (i.e., differences in device arrangement and hardware performance). In large-scale train traffic control systems [5], for example, the process computer in each station-level subsystem executes "route control" by setting routes for arriving and departing trains. Although the basic functions of "route control" are the same, the detailed functions differ slightly between subsystems according to the railroad structures and device performance at the individual stations.

2.2. Subsystem Construction Process

As shown in Fig. 1, all the subsystems in a distributed system are not constructed at the same time. Instead, new subsystems are constructed and connected to the ones constructed earlier. The first step is thus to construct a subsystem and connecting it to another (Fig. 1(a)). The second step is to construct a local system by repeating the fist step (Fig. 1(b)). The third step is to connect the local systems (Fig. 1(c)) because there are some cooperative functions between them.

As construction process is progressing, a subsystem upgrades its execution-mode: The first is a single operational mode where a subsystem executes the degraded operation by itself. The second is an adjacent operational mode where a subsystem, connected to adjacent subsystems, executes the operational control by

communicating with each other. The third is an operational mode where a subsystem, connected to management machines in a local system, executes the operational control according to the schedule and commands from the management machines.

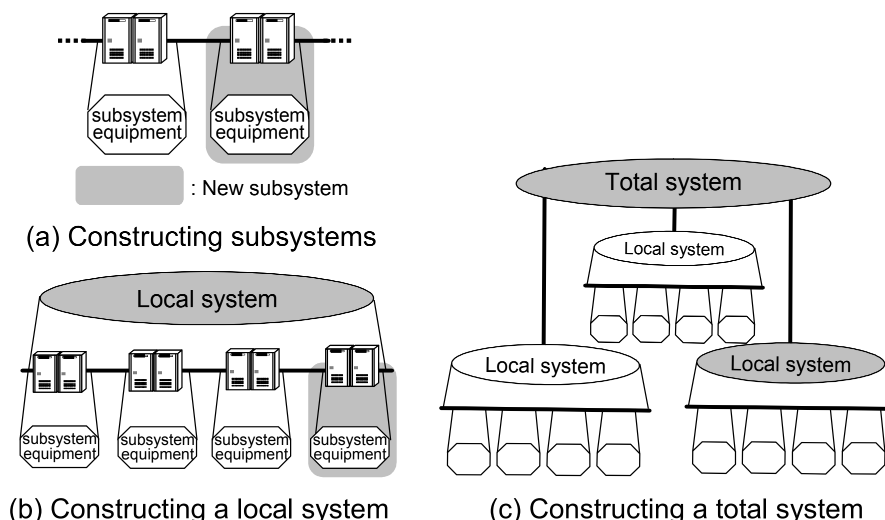

Figure 1: Construction Process

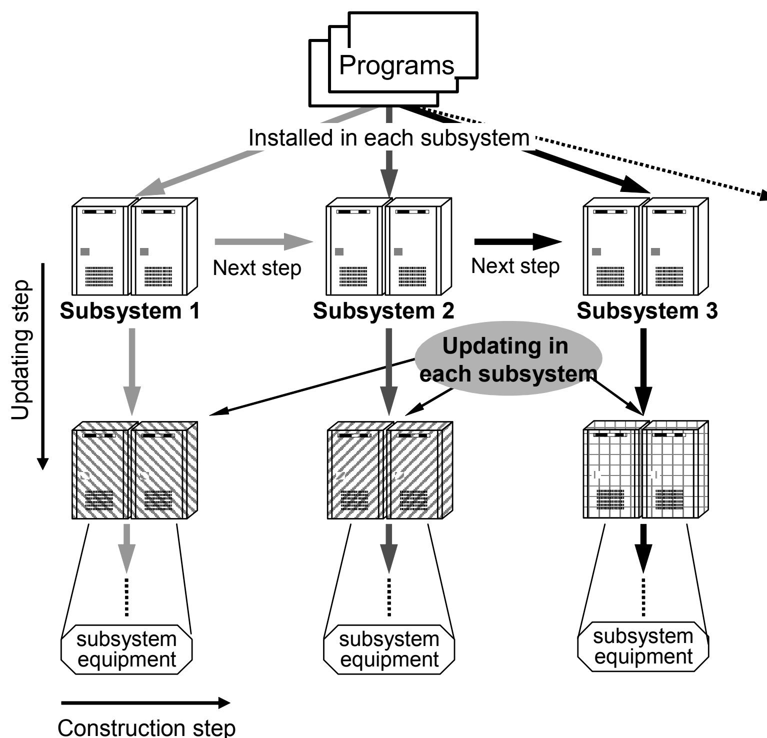

Figure 2: Maintenance Process

2.3. Software Maintenance Process

As mentioned in section 2.1, although the basic functions of the subsystems are identical, the functions of subsystems in large-scale distributed industrial control systems may vary slightly. These differences should be incorporated into one system, and common programs should be generated for software maintenance (if only these programs are installed, every subsystem can start its execution). There are, however, some problems:

(1) Checking the features of many subsystems and generating the programs common to them is very difficult and time-consuming work under the step-by-step construction in Fig. 1.

(2) Since the function of each subsystem gradually changes, the generation of common programs will be an endless task.

(3) The function in a control system cannot be fixed until the actual operation is executed, so the function of a subsystem cannot be completely understood in advance.

Therefore, the following guidelines for software development are defined:

(1) The software structures and program names in all subsystems are unique so that maintenance is easy.

(2) Basic functions are generated as standard programs that cover the standard types of subsystems.

(3) Common functions such as communication and schedule management, which do not depend on the operation conditions, are generated as common programs.

(4) Detail data about devices and their performance are generated as individual programs in each subsystem.

These programs are installed and maintenance is carried out in each subsystem according to two procedures shown in Fig. 2: installing the programs in subsystems sequentially, and updating the installed programs to fit the operational conditions of each subsystem.

3 Requirements for Software Maintenance

3.1. In Program Generation

As described in section 2.3, programs are installed in each subsystem and are updated in each subsystem. These programs are of three kinds:

(1) Common to all subsystems. For example, programs for communication with other subsystems or for schedule management.

(2) Specific to its subsystem. For example, individual programs for defining the local control in its own subsystem.

(3) Not clear whether or not it is related to other subsystems. For example, standard programs of main control functions (Figure 3).

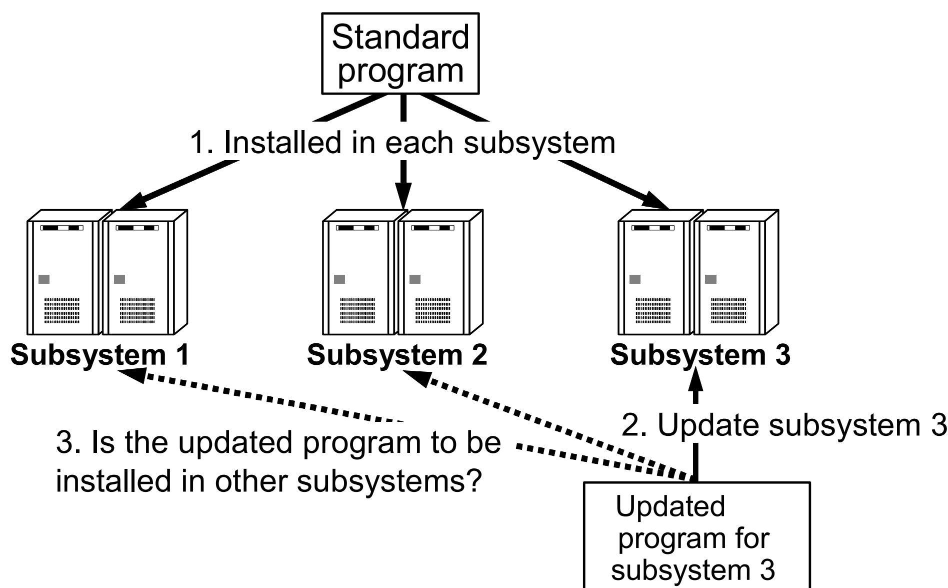

Figure 3: Updating a Program

An updated version of the first kind of program remains the common or standard program and must of course be installed in all the subsystems. An updated version of a program specific to its subsystem is managed as an individual program.

And it is necessary to check whether an updated version of a program that may or may not be related to other subsystems can be installed in other subsystems. If the revision is due to a logical flaw (that is unrelated to operational conditions), the updated program should be installed in all the subsystems. But if the revision is related to the operational conditions, installing the updated program in other subsystems may cause them to malfunction. To maintain such a program as a standard, it must be tested in all the subsystems. This is time-consuming work. Furthermore, whether a program remains a standard one will not be known until the total construction is completed. This kind of program therefore has to be managed as an individual program and must be kept separate from known standard programs in order to keep it from being installed in other subsystems. The standard programs, the individual programs, and the common programs thus coexist in subsystems and their configurations differ between subsystems.

Figure 4 shows changes in the program configurations in subsystems. In this figure, each subsystem I, II, is composed of Program A, B, C. In first step (05.01.01), standard programs were installed in each subsystem. In next step (05.02.03), Program B was revised. This program was updated as a standard version (V1.2) and installed in two subsystems again. In the third step (05.03.10), Program C was updated as a standard version (V1.2) in two subsystems. On the other hand, when Program A was updated, this program was divided into two versions. In subsystem I, an individual version (V1.2_I) was generated and installed. In subsystem II, a standard version (V1.2) was installed. It is therefore necessary to manage the program configurations in each subsystem.

This means that software maintenance requires version management not only for the common and standard programs but also for the individual programs and program configurations in each subsystem. Since only a limited amount of time is available for software maintenance in industrial control systems, it is necessary to minimize the volume of programs downloaded in the target subsystem.

Furthermore, the programs used in real-time control systems are not necessarily independent of each other. A main program, for example, is related to its subroutines. All newly updated programs depending on each other must be installed at the same time. If not, the interface between programs will be illegal and the subsystems will not be able to execute their operations properly even if the programs executed correctly in the test system [6].

Since subsystems will not work properly until all the necessary programs are completed, before standard or individual versions are installed the management must check whether the subsystem will have all the necessary programs.

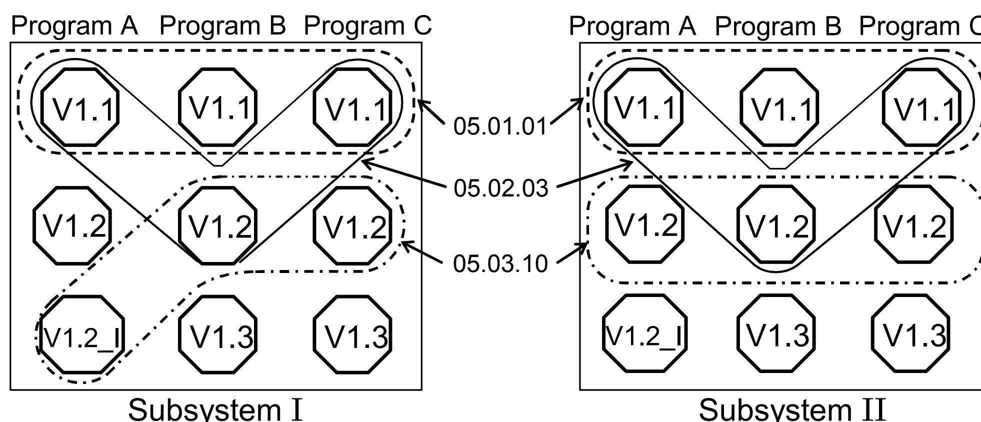

Figure 4: Changes in Program Configurations

3.2. In program Execution

Some kinds of programs used in real-time control systems are statically loaded in memory, and the subsystems cannot be executed as a complete system until these kinds of programs are fully loaded in the execution environment [7]. In software maintenance, these programs have to be checked in each loading module (such as

task program, subroutine program, etc.) to see if they are correctly loaded the execution environment. The configurations of programs loaded in the execution environment also have to be checked. And it must be possible to restore the previous version of a subsystem that malfunctions after being updated.

4. Software Maintenance Management Method

4.1. Generation phase

4.1.1. Programs

As mentioned in section 3.1, when the standard programs in one subsystem are updated, some of them cannot be transferred to other subsystems. Since these programs have to be managed as individual versions, two kinds of versions—standard and individual—coexist as shown in Fig. 5. Software maintenance management must distinguish between these two versions.

Table 1 shows which version numbers are assigned to the standard and individual programs. The standard version has only a version/revision number, whereas the individual version has a subsystem code number as well as the version/revision number. The individual version and its history of changes are managed independently of the standard version. The individual versions are tentatively assigned to the individual programs. Since the individual version programs are not combined with the standard programs until the local system is completed, the program versions can be managed in ways appropriate to the development phase.

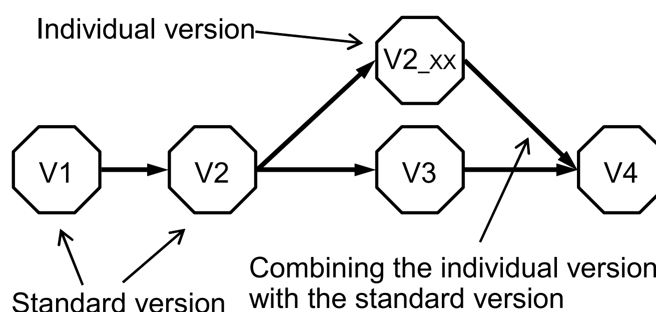

Figure 5: Changes in Program Versions

Table 1: Numbering of program Versions

Program	Version
Standard	V1.5
Individual	V1.5_XX XX: Subsystem number

4.1.2. Subsystems

To satisfy other requirements described in section 3.1, the software configuration in each subsystem is determined ahead of time and programs are installed accordingly. The management method thus has to check the integrity of programs in each subsystem in advance. The system definition file describing the software configuration of the subsystem is shown in Fig. 6. It can be seen in Fig. 6 (a) that the subsystem version is set under the subsystem code number and that there is a system definition file for each subsystem version so that changes between the new and old versions can be checked easily. As shown in Fig. 6 (b), the definition file contains each program name, program group name, version number, and kind of program. Programs are installed according to this definition file.

Thanks to this management method which determines the complete subsystem in advance, the software configuration can be managed in each subsystem. Thus the integrity of the subsystem can be guaranteed and the lack of programs can be prevented as long as the loading of programs does not fail.

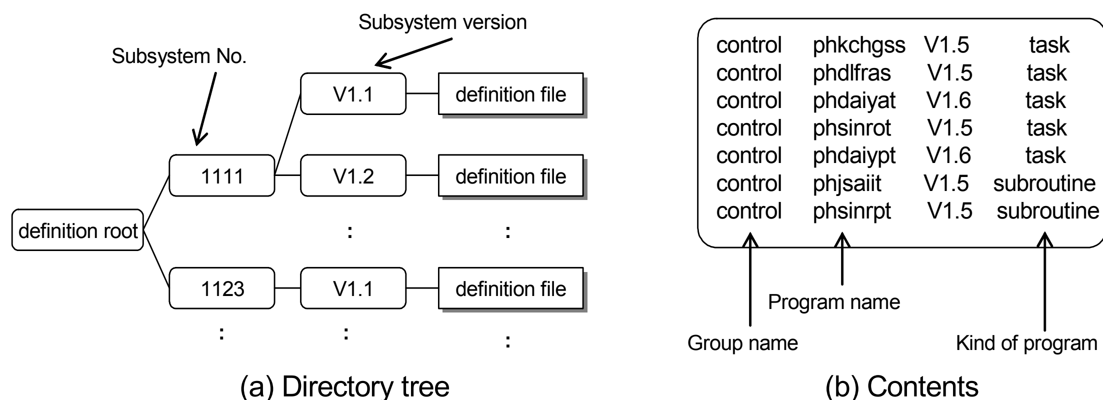

Figure 6: System Definition File

4.2. Execution phase

The programs defined in the definition file are not necessarily loaded into the execution environment of the subsystem because loading may fail. The loaded programs must therefore be checked. This section describes the loading procedure, the procedure used to check the consistency of programs loaded into the execution environment, and the method used to restore the previous version if a subsystem fails.

4.2.1. Loading

Program resources are saved in the program management computer, from which they are installed in each subsystem (Figure 7). According to the system definition file, programs are downloaded from the program management computer to the subsystem on-line or off-line and are temporarily saved in the disk area. During on-line downloading the program management computer compares the contents of the selected definition file with the registration data of the present version recorded in the target subsystem, so only different version programs are downloaded.

The loading command defined for each program loads its program and is executed when the subsystem reboots. The command generates the execution program from its object files and loads the program into the execution environment. It also records the loading result to the registration data file if successful. It gets source file versions from their object files and records the version information about the subsystem, its programs, and their source files in the registration file. The loading can thus be completed before applications start their executions.

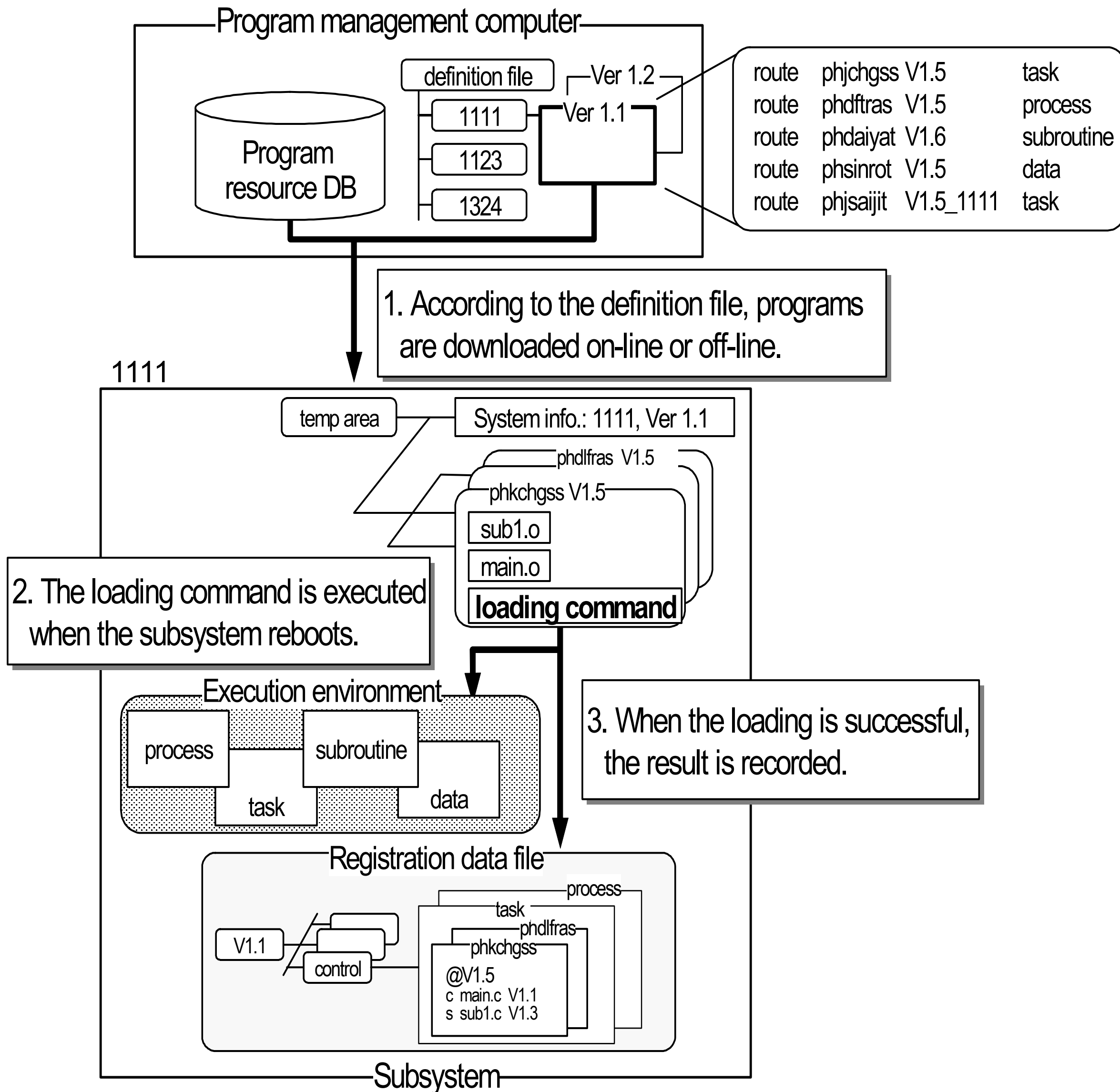

Figure 7: Loading Process

4.2.2. Checking

The loading command does not record the loading result in the registration data file until it determines that the program loading was successful. Since it is guaranteed that the execution environment and the registration data file are identical, the execution environment can therefore be checked simply by referring to the registration data file. Consistency between them can thus be guaranteed.

Figure 8 shows the structure of the registration data file in a subsystem. The loading results of the present subsystem version are registered for each program group. Each program group has four files according to the kinds of programs (task program, process program, subroutine, and data). For each program in each file are written the program name, the program version, the name of the source file, and the source file version. Thus the version information in the execution environment can be easily checked by referring to the registration data file.

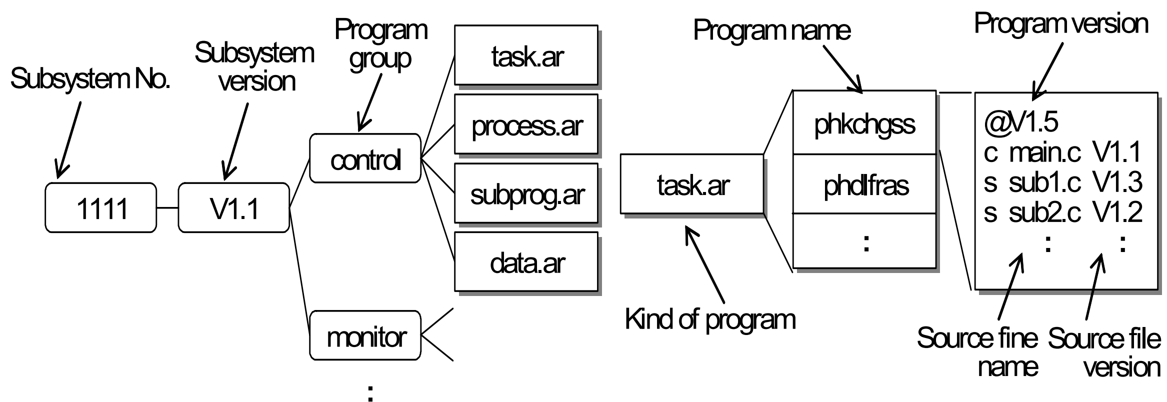

Figure 8: Registration Data File

4.2.3. Restoring a Previous State

If the subsystem malfunctions after updating, it is necessary to restore the subsystem to its previous state. This section discusses a restoration management method that can restore the subsystem to its previous state when the loading of a new version fails

Figure 9 shows the restoration procedures:

(1) The new version programs for updating the subsystem and the previous version programs for restoring the subsystem are downloaded into separate disk areas (temp and temp2).

(2) The compare command in each program restoration is defined as shown in Fig. 9. From its object files it generates the program to be restored, compares it with the execution program loaded into the execution environment, and checks that these two programs are identical. Even if there is only one inconsistency between these two programs, the management stops updating the subsystem to the new version.

(3) If these two programs are identical, the new version programs are loaded.

(4) Even if only one of the new version programs should fail to load, the management stops updating the subsystem. Then all the program versions to be restored are copied to the temporary disk area (temp) and sequentially reloaded.

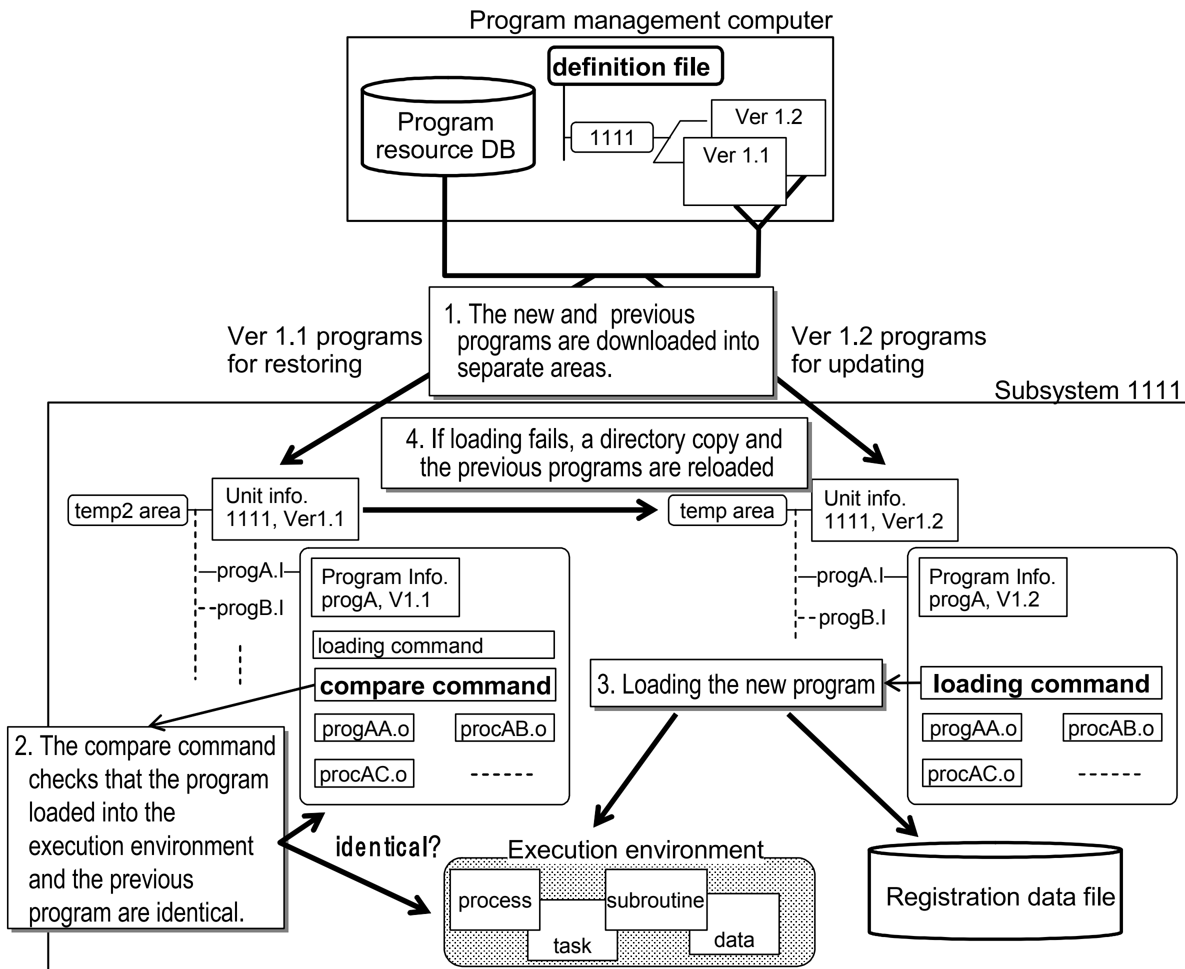

Figure 9: Restoring to the previous State

5. Evaluation

5.1. Application

A large-scale distributed train traffic control system consists of three hierarchical layers, and some train-line systems have more than 30 station-level subsystems. The proposed method in on-line installation was applied to a prototype system for a train-line system. No error or omission occurred when program installation was executed more than 1000 times. And the source files of an executed program could be identified in less than 1 minute, whereas this previously took more than an hour to do by hand.

The proposed off-line installation method using a memory media was applied to the actual operation system. The time available to update and restore a station-level subsystem is less than 20 minutes, and the execution results obtained in this experimental evaluation in which the volume of the installed programs was about 245k bytes are listed in Table 2. The restoration operations were completed in less than 20 minutes, and the updating time was almost same as the restoration time.

Table 2: Restoration results

Items	Time spent
(1) Installation of maintenance tool programs	0.7 minutes
(2) Installation of restored programs	1.1 minutes
(3) Rebooting of the subsystem	10.6 minutes
Total time	12.4 minutes
(4) Saving the loading results to a memory media	1.3 minutes

5.2. The Comparison with Other Existing Maintenance Tools

Differences between the proposed management method and existing tools [3, 8, 10] are listed in Table 3. As mentioned before, the existing tools install package software products into personal computers and workstations. These tools cannot update the subsystems to fit their individual conditions because they only install common programs uniformly to subsystems and perform software version management only for software products. The proposed method, on the other hand, performs version management for each subsystem, each of that subsystem's programs, and each of their source files. The proposed method can also manage individual version programs for specific subsystems.

The proposed method can confirm the integrity of the subsystem according to the system definition. The existing distribution tools do not have such function because they install only package software products.

As both the hardware and software in large-scale distributed industrial control systems have to be constructed by step-by-step, software management suitable for this kind of construction—such as the proposed method—is necessary. The existing maintenance tools do not provide the strict management needed for high reliability.

Table 3: Differences Between the Proposed Method and the Existing Tools

Items	Proposed method	Existing tools
Management level	Source file Program Subsystem	Software product
Individual management for each subsystem	Possible	Not supported
Integrity management for each subsystem	Possible	Not supported

6. Conclusion

The software maintenance management proposed in this paper assures the integrity and consistency of the software in large-scale distributed industrial control systems. The two–phase method controls program generation in a way that makes maintenance easy and controls program execution in a way that makes maintenance more reliable. In the generation phase the individual programs and the standard ones are managed separately and the subsystem is constructed according to the system definition file. In the execution phase software programs in the execution environment are recorded in the registration data file and, in the event of failure, the previous state of the subsystem is restored by installing the previous versions of updated programs. The proposed management method has been used in a distributed train traffic-control system, where it made software maintenance more effective and more reliable.

REFERENCES

1. BARRIE, D., HILL, D. S., and YUEN, A., **Computer Configuration for Ontario Hydro's New Energy Management System**, IEEE Trans. on Power Systems, 4, 3, 1989.

2. DEEN, S. M., **Agent-Based Manufacturing, Advances in the Holonic Approach (Advanced Information Processing)**, Springer Verlag, 2003.

3. KACZMAREK, S. D., **Microsoft Systems Management Server 2003 Administrator's Companion (Administrators Companion)**, Microsoft Press, 2003.

4. KAKUMOTO, Y., NISHIJIMA, E., SUIZU, H., and KOMODA, N., **Development of Middleware Applicable to Various Types of System Structure in Train-Traffic-Control Systems**, Proc. 9th Conf. on Computer Aided Design, Manufacture and Operation in the Railway and Other Mass Transit Systems, 2004.

5. KERA, K., BEKKI, K., FUJIWARA, F., KITAHARA, F., and KAMIJO, K., **Assurance System Technologies Based on Autonomous Decentralized System for Large Scale Transport Operation Control System**, IEICE Transactions on Communications, E83-B, 5, 2000.

6. KOBAYASHI, H., NAKANISHI, H., and HAYASHI, K., **On-line Software Expansion in Large-Scale Widely Distributed Systems**, Proc. 1st Symposium on Autonomous Decentralized System, 1993.

7. KOPTETS, H., **Real-Time Systems -Design Principles for Distributed Embedded Applications**-, Kluwer Academic, 1997.

8. MULLER, H. A., NORMAN, R. J., SLONIM, J., and MULLER, H., **Computer Aided Software Engineering**, Kluwer Academic, 1996.

9. SINGH, N. and RAJAMANI, D**., Cellular Manufacturing Systems: Design, Planning and Control**, Chapman & Hall, 1996.

10. VESPERMAN, J., **Essential CVS**, O'reilly & Associates, 2003.