Regular Paper

The Applicability of Formal Specification to Maintenance of Large-Scale Software

SEIICHI KAWANO*†, KOUICHI ONO*, YOSHIAKI FUKAZAWA* and TOSHIO KADOKURA*

For over a decade, major reserach efforts have been devoted to formal specification techniques for system development. As a result, many specification languages and their support tools have been developed. But not much research has been done on the application of formal specifications to software maintenance.

We developed a formal specification language called Waseda Specification Notation (WSN), and have been attempting to apply it to some existing large-scale software systems. One of these is the scheduler of the VM/SP operating system, which was completely specified in WSN.

We tuned the scheduler on the basis of the formal specification described, and according to the current characteristics of our computer environment.

This paper gives a detailed introduction to the benefits of applying formal specification methods during the maintenance of large systems. Experiences obtained during the study are also described.

1. Introduction

The application of formal specification techniques to software development has been researched for over a decade [1, 2, 3]. The specification of an existing software system brings many benefits. Long-term benefits result when specification is used as:

1. A basis for discussing and developing specifications for changes or additions to the system
2. A basis for selecting materials for reimplementation of the system in cases such as when the machine or operating system is replaced
3. A model of the system's functional behaviour for use in educating new staff.

When changes or additions to the system are made, a new specification can be developed with reference to the previous specification. This process will give insight into the effects of the changes and their interactions with the existing parts of the system.

We are attempting to scale up formal specification methods, so far used in a research environment, to large-scale software in an industrial environment. For this purpose, we defined the specification language Waseda Specification Notation (WSN) and wrote many specifications in the language [4, 5]. WSN shows how a mathematically based notation can be used to capture important aspects of the behaviour of a system that is clearly not just a toy. In the design of WSN, we paid special attention to the writability and readability of the specification.

*Department of Electrical Engineering, School of Science and Engineering, Waseda University.
†Now with Yamato Laboratory, IBM Japan, Ltd.

This paper describes our experiences in applying formal specification techniques to the maintenance of the IBM Virtual Machine/System Product (VM/SP).

There have been some attempts to apply formal specification techniques to existing real software [6, 7], but the applicability of specification to the maintenance phase has not been studied well enough.

We selected the Control Program (CP) of the VM/SP operating system (Release 4.0) as our first maintenance target. VM/SP [8] is an interactive and multiple-access operating system. It has two basic components: Control Program (CP) and Conversational Monitor System (CMS). Each provides services of a specific type. CP manages system resources and delivers an individual working environment to each user. Of CP's many modules, the dispatcher and the scheduler play major roles. Since the dispatcher's activities depend on the actual hardware and because of the scheduler's central position in CP, we first attempt a formal specification of the scheduler.

2. The Specification Process

The starting point for our specification work was the VM/SP System Logic Manual [9]. Very soon, we had many questions that were not satisfactorily answered by the manual, because of its incompleteness and vagueness. Most of the questions that arose in the specification process were subtle and required us to refer to the source code of the module to find satisfactory answers.

Moreover, the comments in the source code written in the assembler language were also incomplete and vague.

2.1 Notation

The specification language WSN developed by the Study Group of Specification and Verification at the Centre for Informatics, Waseda University, is the primary notation that has been used in the specification work.

The formal basis of WSN is elementary set theory. WSN is a many sorted, type-free, and weak second-order language. Those familiar with set theory should have little trouble in reading and writing specifications in WSN.

Specifications in WSN are written at two levels. One is called a "high-level specification (HLS)" and the other a "descriptive specification (DS)."

An HLS describes the abstract function of a module. This is achieved by providing a formal specification notation and, where appropriate, natural language specifications, figures, and diagrams. The main purpose of an HLS is to assist the interpretation of a DS. We do not intend to provide a mathematical verification of HLS.

A DS formally specifies the structure and other considerations crucial to the implementation. To improve the readability, comments in a natural language, figures, and other explanatory items can be added where needed. A DS is used for implementing and verifying the redesigned software. A short summary of the notation used within this paper is given in Appendix 1.

The object of an HLS is to help the reader of the specification in WSN to understand the DS. It is not possible to prove the correctness of an HLS, but formal and informal specification can complement each other well. This makes it quicker and easier for the readers of the specification to understand it.

WSN does not confine the specifier to a particular style of specification method. A software specifier may choose a style suited to a given problem. The specification may be written in a predicate logic style, a functional style, a precondition-postcondition style, or any other style suited to the problem. This is analogous to choosing a programming language for a particular problem, such as Cobol for data processing instead of Lisp.

WSN was designed not for performance specification, but for functional specification of required software. Therefore the necessary execution time, the capacity of main storage, and so on cannot be specified in WSN. We asume that such items should be specified as comments, and these are taken into consideration in the implementation phase.

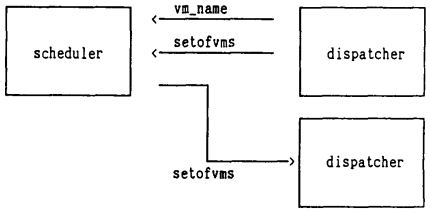
2.2 Sample Specification

As a sample of a specification, we will look in detail at the specification of the CP scheduler. Note that the specification is detailed enough to capture the behaviour of the scheduler, yet can generalize from issues of data representation and detail algorithms, which might be used to implement the system.

```
vm_name = natural number

setofvms = set of vm


schedule : vm_name X setofvms -> setofvms
```




```
The function of the scheduler is formally specified as follows:

  scheduler : vm_name X setofvms -> vmlist

              where  vm_name = natural number

                     setofvms = set of vms

                     vm = vm_name X status

                     status = P1 X P2 X P3 ..... X Pn
```

Fig. 1 High-Level Specification of Scheduler.

```
define predicate virtualmachine( vm ) <==>
  [ ( Exist! name in N )
    ( Exist! runnable in { "Runnable", "notRunnable" } )
    ( Exist! queue in { "Q1", "Q2", "Q3" } )
    ( Exist! type in { "interactive", "compute" } )
    ( Exist! inQ in { "InQueue", "notInQueue" } )
    ( Exist! wasrun in { "WasRunnable", "notWasRunnable" } )
                          :
    [ vm = < name, runnable, queue, type, inQ, wasrun, Elige,
             priority, TE, QE, SS, AE, LW, CF, PGW, DBW, cnt,
             page, time, UHS, FS, WE, RP, AEXP, DSP, logon,
             logoff, idle, FVRF, noQ3, elg, dd >
    ] ]
```

Fig. 2 Definition of Virtual Machine in WSN.

2.2.1 High-Level Specification

CP manages the services required by each vm as in Fig. 1. Each vm can be thought of as a list of its name and other control and status information.

The scheduler is called when the dispatcher requires it to update the status of a vm. This situation, for example, occurs when a vm is waiting for an I/O or runs out of its time slice. The dispatcher passes the vm_name (a natural number) and the list of vms (set of vms) to the scheduler. The activated scheduler analyzes the status of the vm in question, updates the vm status for the next execution, and repositions the vm within the list of vms. This activity is shown in Fig. 1. The dispatcher may then choose a suitable vm for execution from the list of vms and dispatch it.

The scheduler expressed in the function "scheduler" accepts a natural number signifying a vm_name and a set of vms, and returns a set of vms. Each vm is represent-

```
define predicate runnable( vm ) <==>
      [ element( vm, 2 ) = 'Runnable' ]
```

Fig. 3 Example of Predicate Definition.

```
define function vmMakeRunnable( vm ) =
       changeElement( vm, 2, 'Runnable' )
```

Fig. 4 Example of Function Definition.

ed by a list with its name and other information such as its status.

2.2.2 Descriptive Specification

In the description of a DS, first of all, a virtual machine (vm) that is an object of the scheduler must be defined. In CP, a vm is viewed as a data block called 'VMBLOCK,' in which many dynamic properties of the vm are recorded. Examples of these properties are whether the vm is runnable and how much memory it has used. The scheduler does not use all the data of each vm. Some of them are used only by the dispatcher, or other CP modules such as the memory control module. We selected 32 properties of each vm necessary for effective scheduling.

We define a vm as a list structure of these 32 properties, which is the predicate 'virtualmachine'.

The scheduler must manage these vm properties and may change them dynamically. Some functions and predicates are defined to access the properties or change their values. For example, the definition of the predicate 'runnable' is shown in Fig. 3.

The predicate 'runnable' is true if the second element of a vm (list structure) is "Runnable," which means that the vm is runnable. When the predicate is false, the vm is not runnable; in other words, the vm is in an idle state or an I/O blocked state.

The built-in function 'element' returns an element from a list. Element (L, n) yields the n-th component of the list L. For example, element ([a, b, c], 2)=b.

The function 'vmMakeRunnable' (in Fig. 4) sets the vm's second element to "Runnable." This means that the function makes a vm runnable. The function changeElement (L, n, x) returns a new list, whose n-th element is x. For example, changeElement ([a, b, c], 2, d) =[a, d, c].

Up to now, we have only described the predicate and function definitions for checking or modifying the second element of a vm, which indicates whether the vm is runnable. We now define some other functions and predicates for checking or modifying other elements (properties) of a vm.

The set of vms is now defined. vms are roughly classified into the following thre types. Each type of vm belongs to one of three lists: the runlist, the eligiblelist, or the nonActivelist.

vms in the runlist: runnable and candidate for dispatch

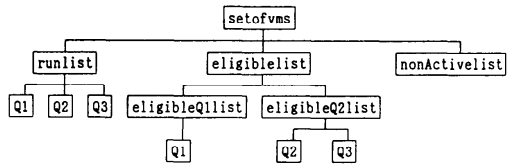


Fig. 5 Structure of VM List.

```
/* Specification of VM/SP Scheduler */
    specification of scheduler
        ( Creation! vm )[ virtualmachine( vm ) ]
        ( Creation! VM )[ setofvms( VM ) ]
        ( Creation! systemData )[ dataOfSystem( systemData ) ]

        scheduler( < vm, VM, systemData > )
        = < vm', VM', systemData' >
    end of scheduler
```

Fig. 6 Specification of scheduler.

vms in the eligiblelist: runnable but not candidate for dispatch

vms in the nonActivelist: not runnable

According to their characteristics, all runnable vms (in the runlist or the eligiblelist) are divided into three classes, regardless of the list they belong to. The types are interactive (or I/O bound), compute bound, and heavy compute bound. The classes are called Q1-type, Q2-type, and Q3-type correspondingly.

The eligiblelist is further separated into two parts, the 'eligbleQ1list' and the 'eligibleQ2list.' The first contains 'Q1'-type vms and the other 'Q2' and 'Q3'-type vms. These classifications are illustrated in Fig. 5.

In all, it is necessary to define four lists (the runlist, the eligibleQ1list, the eligibleQ2list, and the nonActivelist). The list that consists of the above four lists is called 'setofvms.'

We must also define other information crucial for effective scheduling. These data are defined as a list named systemData. The components of this list include memory usage, CPU usage, and the number of active vms.

After the necessary auxiliary function and predicate definition, the main part of the scheduler is specified in a top-down manner.

In the specification of the scheduler, the objects of specification must first be 'created.' These objects are one vm, a set of vms, and system information.

The scheduling function is represented by the function 'scheduler.' A single quotation mark means that the quoted object is modified by the function. For example, vm' is a modified vm. As described in the HLS, this function receives a vm, the set of vms, and some information on the system, and updates them (Fig. 6).

The definition of the function 'scheduler' is shown in Fig. 7. This means that if the vm is runnable, then the function scheduler is to call the function 'runstate'; otherwise, it is to call the function 'waitstate.'

```
define function scheduler( < vm, VM, systemData > )
  = [ runnable( vm ) --> runstate( < vm, VM, systemData > ) :
                         waitstate( < vm, VM, systemData > ) ]
```

Fig. 7 Definition of Function scheduler.

```
define function makeDrop( < vm, VM, systemData > ) =
   promote · reEntry · dropQueue ·
        dropFromRunList( < vm, VM, systemData > )
```

Fig. 8 Definition of Function makeDrop.

```
define function reEntry( < vm, VM, systemData > ) =
  [ assuredExecution( vm ) -->
       addToRunList · addQueue( < vm, VM, systemData > ) :
       addToElgList( < vm, VM, systemData > ) ]
```

Fig. 9 Definition of Function reEntry.

```
define function checkPage( < vm, VM, systemData > )
  = [ ( vmCkPage( vm ) > vmPageRead( vm ) ) -->
      ( < vm, VM, systemData > ) :
      [ delayDrop( vm ) -->
           checkTotalPage( < vm, VM, systemData > ) :
           checkStealPage( < vm, VM, systemData > ) ] ]
```

Fig. 10 Definition of Function checkPage.

The main work of the scheduler is to manage three lists: the runlist, the eligibleQ1list, and the eligibleQ2list. A vm is usually first entered into the eligiblelist (the eligibleQ1list or the eligibleQ2list), then added to the runlist, and held until it can run for a certain term (time slice). These vms are sorted by 'Dead-Line Priority,' and usually the first vm in the runlist will be able to run. If the status of the running vm changes, the scheduler is called and reorganizes the three lists. One of the status changes is to make a vm not-runnable. This may occur as a result of an I/O request or for some other reason.

If the vm requests a long term I/O or runs out of time that it can stay in the runlist, the scheduler drops the vm from the runlist. This is defined by the function 'makeDrop' (Fig. 8). The symbol '·' is the operator for function composition. The function 'dropFromRunlist' describes the operation of dropping the vm from the runlist, and the function 'dropQueue' performs the operation of changing the condition of the vm, for example, to priority or projected working-set. The function 'reEntry' defines the operation of adding the vm to a list (the runlist or the eligiblelist). The function 'promote' defines the operation of searching the eligiblelist for the vm to be added to the runlist.

The runnable vms are added to one of three lists (Fig. 9). If the vm has the option of 'assuredExecution,' which means that it can always stay in the runlist, then the function 'addQueue' updates the conditions of the vm and the 'addToRunList' adds the vm to the runlist. Ordinary vms are added to the eligiblelist by the function 'addToElgList.'

As described, the vms in each of three lists are sorted according to priority, which is calculated on the basis of the user priority, page usage, CPU usage, and so on. The function 'checkPage' in Fig. 10 checks these properties. The definition of 'checkPage' contains the following description.

If the vm does not use some predicted number of pages, then the function 'checkPage' does nothing. This condition is checked by 'vmCkPage(vm)> vmReadPage(vm)'. If this condition does not hold, and the vm is marked as 'delayDrop' by the function 'delayDrop,' then its page usage must be estimated by the function 'checkTotalPage.' If neither of the above conditions holds, the page usage of the vm should be estimated by the function 'checkStealPage.'

3. The Maintenance process

3.1 Decision on the Modification of the Scheduler

We attempted to tune the VM/SP system for our environment. For this purpose, we investigated the current utilization of our VM/SP by using the system monitoring tool VM/RTM (SMART [10]). VM/RTM was designed as a real-time monitor and diagnostic tool for short-time monitoring, analysis, and problem solving. It is used for installation of hardware or software to help in validating the system components and establishing requirements for additional hardware or software. Some of the results are shown in [11].

After considering the results, we decided to modify the scheduler in the following two respects:

1) The eligiblelist should be removed.
2) vms that consume very long periods should be given an initial time-slice of 200 ms.

The eligiblelist plays two roles. One is to prevent the system performance from dropping because of paging. As described, a vm in the eligiblelist is runnable but not a candidate for dispatching. This is because the parts of these vms' memory-space do not exist in real main-memory. When a vm is executed, some parts of its virtual space must exist in real main-memory. But if all virtual space parts of runnable vm exist in main-memory, the paging ratio will increase. In order to keep the system performance high, it is necessary to restrict the number of vms.

Another role of the eligiblelist is to make the response-time (turn-around time) shorter. The eligiblelist is separated into two lists, the eligibleQ1list and the eligibleQ2list. The first contains Q1-type vms and the other contains Q2- and Q3-type vms. When there is room in the runlist, the vms in the eligibleQ1list are first chosen to be added to the runlist. The vm's type is Q1 when it becomes runnable, so this method makes the response-time shorter.

In our computer system, the memory utilization ratio

```
define function makeDrop( < vm, VM, systemData > ) =
   reEntry · dropQueue ·
        dropFromRunList( < vm, VM, systemData > )
```

Fig. 11 Modified Specification of Function makeDrop.

```
define function reEntry( < vm, VM, systemData > ) =
    addToRunList · addQueue( < vm, VM, systemData > ) :
```

Fig. 12 Modified Specification of Function reEntry.

```
define function checkPage( < vm, VM, systemData > )
   = [ ( vmCkPage( vm ) > vmPageRead( vm ) ) -->
       ( < vm, VM, systemData > ) :
         checkStealPage( < vm, VM, systemData > ) ]
```

Fig. 13 Modified Specification of Function checkPage.

is very low. Even if the main memory is used most heavily, its utilization ratio is less than 30% of total memory. In such an environment, the scheduling method based on the eligiblelist is not effective. It seems better to remove the eligiblelist.

However, some of our computer system users often execute very compute bound tasks, such as structural analysis of buildings and simulation of turbulence. It is an attractive idea to give such users (vms) time-slices that are as long as possible. This is the reason for the second proposed modification.

In the current VM/SP system, when vms are added to the runlist from the eligiblelist, all types are given time-slice of about 50 ms. When a vm has spent its first time-slice and can still stay in the runlist, it will be given about 200 ms the next time.

As Q3-type vms tend to be in compute bound state, especially in our computer system environment, the second method seems to be more effective. As a result, the CPU time that the CP uses can be reduced, and the CPU time that the users (vms) use will consequently increase.

3.2 Modification of the Scheduler

Of the two kinds of modification, only the removal of the eligiblelist is discussed in this paper. In what follows, a portion of the modification is described, and the effects of the modification on other descriptions are also explained.

If the eligiblelist is removed, it becomes unnecessary to state that some vm in the eligiblelist is removed from the list and added to the runlist. This is defined in the function 'promote' and its sub-functions.

Consequently, these functions and the 'promote' function calls, including the function 'makeDrop' in Fig. 11, become unnecessary. The 'promote' function call is thus deleted.

Now there exists only one list, namely, the runlist. When a vm becomes runnable, it is added to the runlist immediately. The function 'reEntry,' which described the addition of runnable vms to the runlist or the eligiblelist, is modified as follows. The function is changed to append directly to the runlist without checking the attributes of the vm.

Similarly, all descriptions of adding vms to the eligiblelist have been changed to adding vms directly to the runlist, and the 'promote' function calls have been deleted.

These modifications are rather easy, but it is necessary to check the side-efects caused by the removal of the eligiblelist.

In one of the deleted function definitions, there is a description that changes the vm 'delayDrop' property, which means that this vm should be dropped from the runlist. This 'delayDrop' property is used so that the highest-priority vm in the eligiblelist can be executed sooner. This is easily understood from the specification.

As the eligiblelist is not used, deleting this 'delayDrop' property presents no problem. The property is referred to the function 'checkPage,' which was modified as follows.

3.3 Code Generation

The system for translating from the specification in WSN to Prolog code has already been implemented [12]. This system has generated the Prolog code from the modified specification discussed in the previous section.

The generated predicates (Prolog codes) are classified into two types. One has only one arity that is bound when the predicate is called. These type predicates are referred to here as Type-1. Calling a Type-1 predicate results in success or failure.

The other type has two arities. One of them is bound and the other unbound variable returns or passes the value. These type predicates are referred to here as Type-2. Calling this type of predicate always results in success.

For example, the following predicate prd2 is Type-1, while prd1 and prd3 are Type-2.

```
prd1(X, Y): –
   prd2(X),
   prd3(X, Y).
```

The generated Prolog code is manually translated to assembler code (IBM System/370 Assembler Language). Most of the Prolog code is translated into pseudo-Macro code, and the rest of the code is translated into assembler code with reference to the interface between the scheduler and other modules in the CP of VM/SP. The strategy of the translation is as follows:

1. A type-1 predicate is translated into a conditional instruction or a subroutine call instruction, which sets the truth value in its condition flag.
2. A Type-2 predicate is translated into a subroutine call instruction. As all variables are global, formal parameters are not used.

For example, the following codes (Fig. 14(b)) are

```
prd1( X, Y ) :-
        prd2( X ),
        prd3( X, Y ).

prd1( X, Y ) :-
        prd4( X, Y ).
```

(a) Generated Prolog Code

```
PRD1    DS      0H
        CALL    PRD2
        CONDF   PRD2.PRD1_2
        CALL    PRD3
        RETURN
PRD1_2  DS      0H
        CALL    PRD4
        RETURN
```

(b) Manually Generated Assembly Code

Fig. 14 Manual Translation of Prolog Code.

translated in Fig. 14(a). In Fig. 14, CALL, CONDF, and RETURN are pseudo-Macro-operations. CALL is a subroutine call, and RETURN means returning to the caller. CONDF is a kind of IF-statement. Calling a subroutine PRD2 sets a true or false value in its condition flag. The above CONDF means that if a false value is set in the condition flag of PRD2, then goto label PRD1_2. Similarly, the pseudo-Macro-operation CONDT means that if a true value is set in the condition flag, then goto label.

4. Evaluation and Discussion

During this case study of tuning an OS, we found three advantages in using formal specifications.

One is readability. This allows the user to find the parts to be modified. In the present case study, the modifications of the scheduler are very simple. This is because we are interested not in the tuning of the OS itself, but in the usefulness of formal specification in the maintenance phase. Consequently, the parts to be modified were found very easily.

A second advantage is that WSN is based on predicate logic. Therefore the user can take account of the influences of the modification only by drowing up a specification.

In maintenance activities, it is important to ascertain the influences of changes on other modules. For example, the changes in the scheduler described above consist of the removal of functions relevant to the eligiblelist or the addition of a new function to give the Q3-attributed vm a constant time slice. The former influenced the other functions and predicates because of the deletion of some functions and predicates. For the latter, however, it is sufficient to modify an existing function.

Actually, the specification of the existing scheduler consists of 272 function definitions and 55 predicate definitions. The changes in the scheduler influence 47 function definitions and 9 predicate definitions. Of the 47 influenced function definitions, 6 were modified and 41 were deleted. Of the 9 predicate definitions, 2 were modified and 7 were deleted.

The performances of the original and tuned schedulers were measured [11]. The system throughputs, that is the CPU usage rates of users for the working ratio of the CPU, were 0.50 for the original scheduler and 0.60 for the tuned scheduler. Therefore the tuning improves the performance.

In traditional software development processes, which do not adopt formal specification techniques, modification points must be detected by careful checking of the source code, with a little guidance provided by informal specification. It is very difficult to detect influences on other modules in this approach.

In our experiences based on a formal specification method, it is easy to remove a function relevant to the eligiblelist and to effect a relevant modification of the delayDrop attribute. Moreover, the modification of the corresponding source code is facilitated by referring to the modified formal specification written at a higher abstraction level than the source code in assembler language. The abstraction level of the specification can be sufficient to effect the above changes.

However, we also found the problem that software maintenance based on the formal specification approach cannot be applied to changes that are not written in the specification. In order to cope with this difficulty, it is necessary to write a specification near the source code level.

The third advantage of formal specification is that this specification can be used as a communication method.

In the maintenance of large-scale software, the implementer and maintainer are different people. Person-to-person communication is therefore very important, and formal specification is thus required to be understandable and rigorous. To meet this demand, it is necessary to raise the abstraction level and improve the understandability.

These are very difficult trade-offs. But in our experience, the larger the scale of the software, the more important the person-to-person communication. One of our conclusions is that in software maintenance, a high abstraction level and an understandable formal specification are very useful. If necessary, efficient maintenance can be realized by providing a more detailed formal specification.

The actual workload of modifying the existing system was as follows. It took about 6 person-months to draw up a specification of the existing scheduler, 4 person-months to investigate the university computer environment and decide the points that required modification, and about 1 person-month to translate from Prolog to assembler code and introduce the modified system.

We consider that there are no general methods for finding the correspondence between a user's original requirement and a given specification, or between the

specification and its program. Our principles are as follows: to draw up a readable and understandable specification, and to offer support tools for understanding a specification and program. We are studying support tools for understand specifications and programs [13, 14]. This work supports the verification of a given specification by transformation, and the understanding of a program by both specification and cliche assignment.

It may seem a waste of time to draw up a formal specification, because it involves such a lot of work. But once the formal specification has been made, it seems to make it very easy to maintain an existing system. The fact that the abstraction level of the specification coped with the modification points proves that a maintenance process using formal specification is more useful than a traditional maintenance process without formal specification.

WSN is intended for the reader who can decide on the implementation from the specification. For other readers, a system for generating a program from a specification in the DS is provided [15]. However, the generated program leaves some room for performance improvement.

WSN makes no assumptions on the relation-ship of an HLS and DS. Therefore, both the HLS and the DS are modified individually when the modification of the specification affects the HLS. During our case study, such modification of an HLS was comparatively straightforward because no restrictions were imposed on the representation form of an HLS.

5. Conclusion

We have described the formal specification approach to maintaining a large system and our experiences in adopting this approach. The usefulness of formal specification was confirmed, but some problems remain. One is the abstraction level of the specification. If the required modification is at a lower level than the abstraction level of the specification, a detailed auxiliary description is needed. This problem must be further researched.

Our experience has led us to recognize that some maintenance tools are very useful. In the future, a maintenance environment corresponding to these techniques must be realized.

Acknowledgement

The contents of this paper are some of the research results of the Specifications Study Group at Waseda University. The authors are greatly indebted to the other members of the group and to many others who gave us helpful suggestions.

References

1. GEHANI, N. and MCGETTRICK, A. D. Software Specification Techniques, Addison-Wesley Publishing Company (1986).
2. CHI, U. H. Formal Specification of User Interface: A Comparison and Evaluation of Four Axiomatic Approaches, *IEEE Trans. on SE*, SE-**10**, 2 (1985).
3. HENDERSON, P. Functional Programming, Formal Specification, and Rapid Prototyping, *IEEE Trans. on SE*, SE-**12**, 2 (1986).
4. FUKAZAWA, Y. et al. The Reconstruction of an Operating System Using a Formal Approach, Proc. of the Third International Workshop on Software Specification and Design, IEEE Computer Society (1985).
5. NASU, H., HOSOKAWA, K. and YAMADA, S. WSM and Its Application: Design of an Elevator Control System in WSN, Bulletin of Centre for Informatics, Waseda Univ., **2** (1985).
6. MELLIAR-SMITH, P. M. and SCHWARTZ, S. Z. Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System, *IEEE Trans. Comput.* C-**31**, 7 (1982).
7. HAYES, I. J. Applying Formal Specification of Software Development in Industry, *IEEE Trans. on SE*, SE-**11**, 2 (1985).
8. IBM "Virtual Machine/System Product: General Information," GC20-1838-4 (1984).
9. IBM "Virtual Machine/System Product: System Logic and Problem Determination Guide," LY20-0892-2 (1983).
10. IBM "VM Real Time Monitor Program: Description/Operation Manual," SH20-2777−5 (1986).
11. KAWANO, S., FUKAZAWA, Y. and KADOKURA, T. Tuning of an OS for a University Environment and Related Experiences, Bulletin of Centre for Informatics, Waseda Univ., **6** (1987).
12. YOON, S. et al. Translator Description Language (TDL) for Specification Languages, *Proc. 37th. Conf. IPS Japan* (1988).
13. ONO, K., YAMAMOTO, T. et al. Specification Debugging by Transforming Representation, *Report of SE, IPS Japan*, 90-SE-73 (1990).
14. FUKUDA, T., FUKAZAWA, Y. and KADOKURA, T. Extraction of Design Decisions from Programs and Specifications, *Proc. 41st. Conf., IPS Japan* (1990).
15. YOON, S. A Translator Description Language (TDL) for Specification Languages and Automatic Generation of Their Translators, *Journal of Information Processing*, **13**, 3 (1990).

(Received February 9, 1990; revised November 9, 1990)

Appendix 1. Descriptive Specification of WSN

Definitions

Let s be a specification name; p a predicate name; x, xi identifiers; S, Si sets; Logic_Formula(x) and Logic_Formula($x1, \cdots, xn$) logic formulas that include x and $x1, \cdots, xn$ as their arguments; and Expression(x) and Expression ($x1, \cdots, xn$) expressions (functional terms) that include x and $x1, \cdots, xn$ as their arguments.

specification of $s \cdots$ end of s
  Definition of specification s
define predicate $p(x) \langle\Longrightarrow$ [Logic_Formula(x)]
  Definition of predicate $p(x)$ as equivalent to Logic_Formula(x)
define predicate $p(x1, \cdots, xn) \langle\Longrightarrow$ [Logic_Formula($x1, \cdots, xn$)]

Definition of n-ary predicate $p(x1, \cdots, xn)$ as equivalent to Logic_Formula$(x1, \cdots, xn)$

define function $f(x)$=[Expression(x)]

Definition of function $f(x)$ as equivalent to Expression(x)

define function $f(x1, \cdots, xn)$=[Expression$(x1, \cdots, xn)$]

Definition of n-ary function $f(x1, \cdots, xn)$ as equivalent to Expression$(x1, \cdots, xn)$

Creations

Let p be a predicate name; x an identifier; and S a set.

(Creation! x in S) [$p(x)$]

Creation of a unique object named x:

"create a unique x in S, such that $p(x)$ holds."

Logic

Let P, Q be predicates or logic formulas; p predicate name; x, xi identifiers; S, Si sets.

$\sim P$

Negation: "not P"

P & Q

Conjunction: "P and Q"

$P|Q$

Disjunction: "P or G"

$Q<-P$

$P->Q$

Implication: "P implies Q"

$P<- ->Q$

Equivalence: "P is logically equivalent to Q"

(x in S) [$p(x)$]

Universal quantification:

"for all x in S, $p(x)$ holds."

($\equiv \forall x \in S \cdot p(x)$)

(Exist x in S) [$p(x)$]

Existential quantification:

"there exists an x in S, such that $p(x)$ holds."

($\equiv \exists x \in S \cdot p(x)$)

(Exist! x in S) ($p(x)$]

Unique existence:

"there exists a unique x in S, such that $p(x)$ holds."

($\equiv \exists! x \in S \cdot p(x)$)

($\equiv \exists x \in S \cdot (p(x) \wedge \sim \exists y \in S \cdot (y \neq x \wedge p(y))))$

($x1$ in $S1$)$\cdots$(xn in Sn) [$p(x1, \cdots, xn)$]

Universal quantification:

"for all $x1$ in $S1$, $\cdots$ and xn in Sn, $p(x1, \cdots, xn)$ holds."

($\equiv \forall x1 \in S1; \cdots xn \in Sn \cdot p(x1, \cdots, xn)$]

(Exist $x1$ in $S1$) $\cdots$ (Exist xn in Sn) [$p(x1, \cdots, xn)$]

Existential quantification:

"there exists and $x1$ in $S1$, $\cdots$ and xn in Sn, such that $p(x1, \cdots, xn)$ holds."

($\equiv \exists x1 \in S1; \cdots xn \in Sn \cdot p(x1, \cdots, xn)$)

(Exist! $x1$ in $S1$) $\cdots$ (Exist! xn in Sn) [$p(x1, \cdots, xn)$]

Unique existence:

"there exists a unique $x1$ in $S1$, $\cdots$ and xn in Sn, such that $p(x1, \cdots, xn)$ holds."

($\equiv \exists!$ $x1 \in S1; \cdots xn \in Sn \cdot p(x1, \cdots, xn)$)

$x1=x2$

Equality between terms $x1$ and $x2$ (after evaluation)

($x1$ equals $x2$)

$x1=^*x2$

Identity between terms $x1$ and $x2$

Sets

Let S, T, and Si be sets; x terms.

nul, { }, [ ]

The empty set

x in S
Set membership: "x is an element of S"
($\equiv x \in S$)
S sub T
Set inclusion: "S is a subset of T"
($\equiv S \subset T$)
($\equiv \forall x \in S \cdot x \in T$)
S cap T
Set intersection:
($\equiv S \cap T$)
S kup T
Set union:
($\equiv S \cup T$)
S-T
Set difference
$S1$ X $S2$ X $\cdots$ X Sn
Cartesian product:
the set of all n-tuples such that the i-th component is in set Si
min(S)
Minimum of set S
max(S)
Maximum of set S

Lists
Let xi be terms.
$\langle x1, x2, \cdots, xn \rangle$
Ordered list of $x1, x2, \cdots, xn$

Numbers
N
The set of natural numbers:
(non-negative integers)
Z
The set of integers

Functions
Let x, E, Ei, F and Fi be expressions (functional terms); f function.
E — $>F1$: $F2$
If term:
"If E holds then the value is eval($F1$) else eval($F2$)"
$E1$ — $>F1$ && $E2$ —$>F2$ && $\cdots$ && En —$>Fn$
Case term:
"If $E1$ holds then the value is eval ($F1$),
else if $E2$ holds then the value is eval($F2$),
$\cdots$,
else if En holds then the value is eval(Fn)"
$E \cdot F$
Functional composition:
given E: $S->T$; F: $R->S$; x: R,
$E \cdot F(x) \equiv E(F(x))$ ($\in T$)
$f(x)$
The function f applied to x

Comment
/* $\cdots$ */
Comment