SOFTWARE FAULT TOLERANCE USING HIERARCHICAL N-VERSION PROGRAMMING

Jie Wu

Department of Computer Engineering
Florida Atlantic University
Boca Raton, FL 33431

ABSTRACT

Techniques for software fault tolerance are based on design diversity and employ different software versions for the same problem. One of the most commonly used techniques is N-version programming. In this paper, an extension to N-version programming is proposed, where a problem is viewed as a set of objects which can be hierarchically organized into several levels. *N-version programming* is then applied to objects at different levels. The concept of recovery metaprogram (RMP) is used to provide the needed support and the Ada language is used to make this scheme more concrete.

INTRODUCTION

Fault tolerance is a feature of computing systems that serves to assure the continued delivery of required services in the presence of faults which cause errors within the system. Techniques for software fault tolerance are based on design diversity and employ different software versions for the same program [1]. N-version programming is one of the techniques used to provide fault tolerance in software. This approach requires the independent preparation of several versions of a program. These versions receive identical inputs and each version produces its separate result. These outputs are collected by a voter and the results of the majority (assuming there is one) are assumed to be the correct output.

Software fault tolerance by N-version programming was proposed by Avizienis [2], who defined replication in three domains:

- time replication
- space replication
- information replication

The notation xT/yH/zS means x-fold execution time, y-fold hardware and z-fold software. Three commonly used N-version programming structures are: 1T/nH/nS, 1T/nH/1S, and 1T/1H/nS. Since the efficiency of N-version programming is dependent on the independent failures of different versions, several approaches have been adopted to achieve this objective such as:

- Independent programming teams [3]
- Different languages [4].

In general, N-version programming can be applied to a part of the problem or to the whole problem. The size of the piece selected is important, and might affect the reliability improvement as well as the costs of developing N different versions. A simple extension to N-version programming is to partition a problem into a sequence of subproblems, and N-version programming is then applied to each subproblem. This method is suitable for a restricted set of problems which are sequential in nature.

In this paper we propose another extension to N-version programming which we call *hierarchical N-version programming*. In this method a problem is viewed as a set of objects which can be hierarchically organized into several levels. N-version programming is then applied to objects at different levels. This approach is motivated by the fact that the reliability of the whole system depends on the reliability of the subsystems at each level. Reliable subsystems can enhance the reliability of the whole system. For concreteness, the Ada language is used to describe the proposed scheme. Four levels could be considered in the design:

level 1	the whole program
level 2	module (Ada package)
level 3	procedure (Ada procedure)
level 4	data structure

The problem of implementing a support system for N-version execution is also addressed in this paper and is based on the concept of *recovery metaprogram* (RMP) [5], which consists of a set of primitives supported by the kernel. The RMP is a recovery software which provides syntax and run time support to implement fault tolerant mechanisms. The use of the RMP is to hide the fault tolerance related implementation details from the programmer. Specific support mechanisms can be classified as either *application specific* or *generic*. Generic mechanisms such as assigning each different version to a different processor are supported by the RMP. The application specific mechanism are defined by the user, e.g., voting method.

This paper is organized as follows: in Section 2 the hierartic N-version programming scheme is proposed and some implementation issues are addressed. Section 3 focuses on N-version programming at the subprogram level. Section 4 discusses N-version programming applied to the data structure level, and we provide conclusions in Section 5.

HIERARCHICAL N-VERSION PROGRAMMING

Our method is to apply N-version programming scheme to four different design levels:

level 1 - complete system
level 2 - module
level 3 - procedure
level 4 - data structure

Ada [6] is a suitable language to represent this scheme where the concepts of generic package, package, and task can be used to support objects at different levels. However Ada does not provide the user the necessary features to distribute objects in to processing elements (distribution semantics [7]) and when special fault-tolerant techniques (recovery block [8], [9], N-version programming [2], and resilient procedures [10], [11]) are used Ada lacks direct support in the embedded system. Two possible approaches can be adopted:

CH2998-3/91/0000-0243$01.00©1991IEEE

RMP											
package RMP											
proc. 1T/1H/nS				**proc.** 1T/nH/1S				**proc.** 1T/nH/nS			
T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4

where

T1: **task** user-interface
Receives messages from the user. No entries.

T2: **task type** proc-caller
Calls the critical procedure and sends data to task voter. No entries.

T3: **task** voter
Votes the data from N different procedures and sends an error message to task reconfiguration. One entry data-to-be-voted.

T4: **task** reconfiguration
Isolates the procedures where errors have occurred based on messages sent from voter. One entry: error message.

Of these four tasks T2 and T3 are essential while T1 and T4 are optional. The above structure is just a skeleton, implementation details need further study.

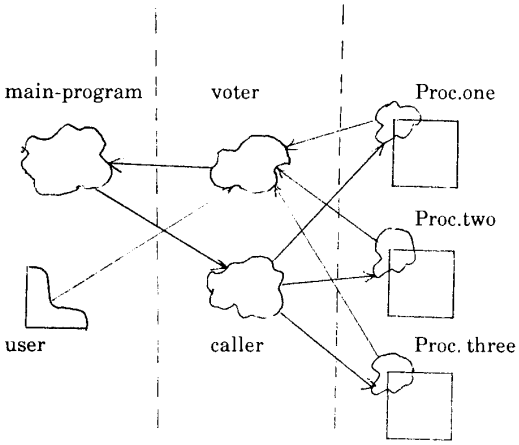


Figure 2. A detailed RMP implementation.

The the Ada program structure for N-version programming applied at the procedure level using 1T/nH/1S is as follows:

AT MAIN-PROGRAM LEVEL

```
proc. main-program
      .
   package RMP_inst_one is new
        RMP_one (vote-method-one, A_i);
   package RMP_inst_two is new
        RMP_one (vote-method-two, A_k);
      .
   begin
      .
   RMP_inst_one. 1T/nH/1S
      .
      .
   B_j
      .
   RMP_inst_two. 1T/nH/1S
      .
      .
   end main-program
```

AT PROCEDURE LEVEL

Since the method 1T/nH/1S is used, nothing needs to be changed.

```
RMP
   generic
        version    :    integer;
        vote-method:    (first, majority, ...);
        with procedure A
   package RMP
        procedure 1T/nH/nS is separate;
        procedure 1T/nH/1S is separate;
        procedure 1T/1H/nS is separate;
   end RMP.
   separate (RMP);
   procedure 1T/nH/1S is
        task user-interface;
        task body user-interface is separate;
        task reconfiguration;
             entry error-message
        end reconfiguration;
        task body reconfiguration is separate;
        task voter
             entry data_to_be_voted
        end voter;
        task body voter is separate;
        task type proc-caller;
        task body proc-caller;
             A;
             vote.data
        end proc-caller;
        proc: array (1..version) of proc-caller;
   begin
        null;
   end
```

N-VERSION PROGRAMMING AT THE DATA STRUCTURE LEVEL

As discussed earlier the reliability of a software system depends on the design of the system at different levels. In this section, we discuss N-version programming at the data structure level. Several papers have discussed redundancy in data structures [14], but they concentrate on robust data structures. Here we use diversity of data structures, that is, several different data structures are used to implement the same algorithm.

Note that Ada allows great flexibility in defining different types of data structures. It is sometimes difficult to visualize and use safely these structures in a program. For example Ada allows the user to do the declarations shown in Example 1:

```
type item;
type pointer is access item;
type item is
   record
        component     :    component-type
        pointer-one   :    pointer:
        pointer-two   :    pointer;
   end record;
```

This structure has two different meanings. It can be a binary tree as shown in Figure 3 (a) or it can be a double linked list as in Figure 3 (b).

(1) Extend Ada [12] using special commands are used for specific fault tolerance methods (such as N-version programming). For example, special commands embedded in the program can specify different voting methods in N-version programming. General commands can be used to support distribution semantics and failure semantics. One possible form to provide task distribution can be [7]:

pragma (distribution),

which distributes tasks to different processing elements.

Two approaches are usually adopted for task distribution. One approach requires the application programmer to consider the hardware configuration very early in the development process. The other approach requires an Ada program to be written using the normal features of the Ada language. After its design, the program is partitioned for execution on different processing elements. The problem with this approach is its cost - both the Ada syntax and the run-time system have to be extended.

(2) Make full use of Ada facilities.

The former method suffers from the cost to expand Ada. A better method does not require to change Ada and the special commands can be implemented in a special layer (the RMP layer). Normally software fault-tolerance techniques do not need to know details of hardware aspects. For the N-version programming approach only one general commands will be used which assigns each version to different processing elements. It is safe to assume that the run-time system or the operating system will assign each different version (implemented as tasks) to different processing elements.

The latter approach will be used in the paper. One more issue that needs to be addressed is to what degree should application programmers know the details of implementation of the N-version programming. Ancona et al. [12] discuss two methodologies: *application-embedded* and *application-transparent*. In application-embedded implementation, the whole application is designed and developed so as to include special commands for N-version programming. In application transparent implementation system behavior in case of faults is specified without modifying actual application software. Strict application-embedded methods suffer from overhead at the application level, and programmers need to know all the details of N-version programming, therefore greatly complicating the program implementation, readability, and maintenance. Strict application-transparent is also impractical, since the run-time system might not be so intelligent as to identify the critical parts of the program. In this paper, we adopt a method which falls between the application-embedded and application-transparent methods. The basic idea is to make recovery transparent to the programmer without considerably increasing the overhead on the RMP or the run-time supporting system.

The Recovery Metaprogram (RMP), acts like a programmer who monitors and modifies the execution of the application program. Figure 1 shows a general hierarchical N-version system using RMP. The implementation of hierarchical N-version programming using Ada at the procedure level and at the data structure level is discussed in the following two sections.

N-VERSION PROGRAMMING AT THE PROCEDURE LEVEL

N-version programming at the procedure level can be expressed as: when executing a main program (module, subprogram) one or more procedures are called by the main program. These procedures are classified into critical or non-critical. Non-critical procedures are conventional procedures (one version for each) while procedures in the critical sections require different versions.

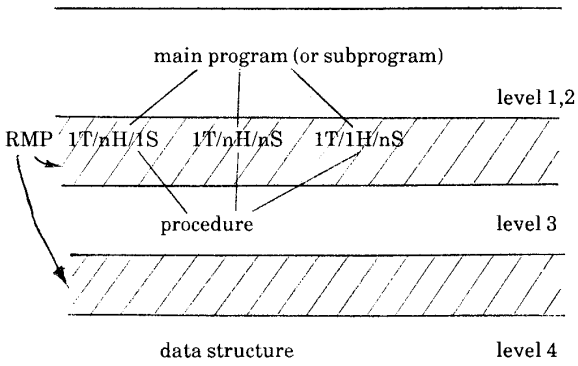


Figure 1: An implementation of hierarchical n-version programming using RMP

The general structure of an Ada program is then as follows:

```
proc. main_program     proc. A1      proc. B1
  begin                  begin         begin
    .                      body A1       body B1
    Ai                   end           end
    .                      .             .
    .                      .             .
    Bj                     .             .
    .                  proc. An      proc. Bm
    .                    begin         begin
    Ak                     body An       body Bm
    .                    end           end
  end
                       critical units    non-critical units

main-program level          procedure level
```

To apply N-version programming we try to modify the application program as little as possible, and all the fault tolerance related details are put into the RMP. A detailed system structure is shown in Figure 2.

At the main-program level only the calls to critical procedures need to be changed. Instead of calling these procedures directly, the main program calls them via the RMP. The necessary information to be sent from the main program to the RMP is listed below:

- The adopted method, such as 1T/nH/1S or 1T/1H/nS.
- The selected voting method to be used, such as by *first* or by *majority* [10].
- The name of the called procedure.

At the procedure level the programmer only needs to provide N different versions (procedures) of the modules which are critical. Note that when 1T/nH/1S is used, nothing needs to be changed at the procedure level.

The RMP is implemented as a generic package and includes three procedures which correspond to three different N-version programming structures. Each structure has four tasks, with each representing a primitive in the RMP.

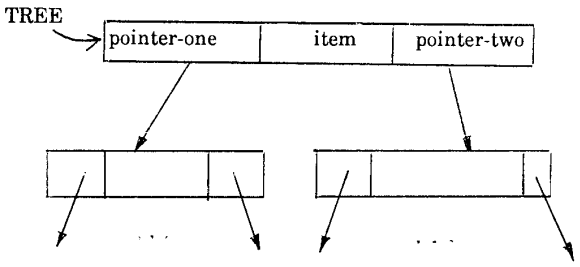


Figure 3 (a) A TREE data structure

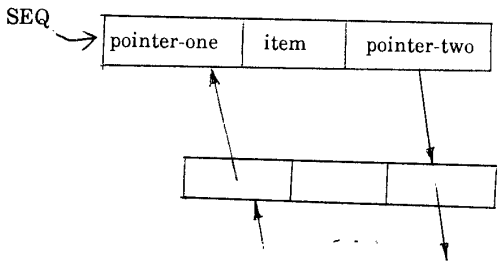


Figure 3 (b) A SEQ data structure

The access type defined here is also a low-level feature and it may easily be misused, although TREE and SEQ used here intend to be higher level structures.

We propose below a high level data representation that can express a large variety of data structures.

```
data          : : = {{<item>}, <item_number>}
item          : : = <homo_item> ¦ <heter_item>
homo_item     : : = <component> <component_number>
heter_item    : : = <component> ¦ <component>,
                    <heter_item>
component     : : = {<item>} ¦ <name> : <name_type>
item_number : : = integer ¦ unlimited
component_number : : = integer ¦ unlimited
```

<name> in the above representation could be an identifier and <name_type> could be different data types defined in the implementation language.

With this high level data representation, the structure in Example 2 can be expressed as:

{{component : component_type}, unlimited}

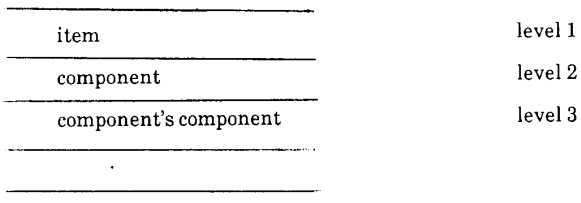
Example 2: A data structure in Ada is defined as follows:

```
type grades_on_course is array (1 . .M) of integer;
type personal_record is
     record
          my_name     :     string (1 . .80);
          my_grade    :     grades_on_course;
     end record;
type class_record is array (1 . .N) of personal_record;
```

With the above high level data representation, this data structure can be expressed as:

{my_name: string (1. .80), {single_grade: integer), M}}, N}

The high level data representation can be thought of as a hierarchical structure:



Between every two adjacent levels there can exist a set of different structure representations (array, list, tree, heap, . . ., etc). The number of levels can be derived directly from the high level structure by counting the number of pairs of { }.

There are two options that can be used to implement the high level data representation:

(1) In the application software (data structure level) M different data structures are defined. The selection of these data structures is transparent to the programmer, and the RMP is in charge of the selection.

(2) The programmer does not need to supply different kinds of data structures. Some general-purpose generic data structures are predefined in the RMP. The RMP is responsible to instantiate M different data structures based on the messages (in the high level data representation) sent to the procedure level.

Option one is similar to the approach in N-version programming at the procedure level; therefore we focus here on option two.

The generic structure in Ada can be used to send a high level data structure from the procedure level to the RMP level.

```
procedure user                  generic
     .                              .
     .                              .
package inst is new             list of messages needed to
     .                          instantiate a data structure
                                in the RMP
     RMP                            .
     (list_of_messages)             .
     .                          package RMP
                                    .
                                    .
end user.                       end RMP
```

The list of messages includes:

(1) item number
(2) number of levels
(3) components at each level

In the RMP there are a set of predefined data structures (Figure 5). The selection of data structure is based on the type of message passed to the RMP.

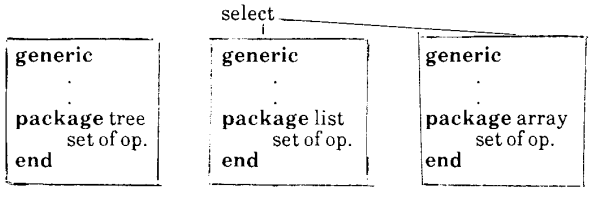


Figure 4. Selection of data structures.

The main difference between the method used here and the one in the previous section is that here the N different data structures are 'automatically' defined, while in the previous method N different versions are defined by the programmer.

CONCLUSION

The concept of hierarchical N-version programming has been proposed. Some implementation structures at the procedure level and at the data structure level are studied and compared. Ada examples are used for concreteness. The concept of RMP is used to implement the proposed scheme but implementation details need further study.

REFERENCES

[1] Hecht, H. and M. Hecht, "Fault-tolerant software," in *Fault-tolerant computing theory and techniques*, Vol. 2, D. J. Pradhan, Ed., NJ, Prentice-Hall, Englewood Cliffs, 1986, pp 658-696.

[2] Avizienis, A., "The N-version approach to fault-tolerant software," *IEEE Trans. on Software Engineering*, 11, 12, Dec. 1985, pp 1491-1501.

[3] Knight, J. C., and N. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," IEEE Trans. on *Software Engineering*, 12, 1, Jan. 1986, pp 96-109.

[4] Purtilo, J. M. and P. Jalote, "A system for supporting multi-language versions for software fault tolerance," *Proc. of FTCS 19*, 1989, pp 268-274.

[5] Ancona, M., G. Dodero, V. Gianuzzi, A. Clematis, and E. B. Fernandez, "A system architecture for fault tolerance in concurrent software," *IEEE Computer*, 23, 10, Oct. 1990, pp 23-32.

[6] U.S. Dept. of Defense, *Reference manual for the Ada programming language*, MIL-STD 1815A, Feb. 1983.

[7] Knight, J. C. and S. T. Gregory, "On the implementation and use of Ada on fault-tolerant distributed systems", *IEEE Trans on Software Engineering*, 13, 5, May 1987.

[8] Randell, B., "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, 1, 6, June 1975, pp 220-232.

[9] Kim, K. H., "Approaches to mechanization of the conversation scheme based on monitors," *IEEE Trans. on Software Engineering*, 8, 5, May 1982, 1pp 89-197.

[10] Lin, K. J., "Resilient procedures -- an approach to highly available systems", *Proc. Intl. Conf. on Computer Languages*, 1986.

[11] Svobodova, L., "Resilient distributed computing," *IEEE Trans. on Software Engineering*, 10, 5, May 1984.

[12] Jha, R., J. M. Kamrad II, and D. T. Cornhill, "Ada program partitioning language: a notation for distributed Ada programmers," *IEEE Trans. on Software Engineer*, 15, 3, March 1989.

[13] Ancona, M., A. Clematis, G. Dodero, E. B. Fernandez, and V. Gianuzzi, "Using different Language levels for implementing fault tolerant programs," *Micro processing and Microprogramming*, 20, 1987, pp 33-38.

[14] Taylor, D. J., D. E. Morgan, and J. P. Black, "Redundancy in data structures: improving software fault tolerance, *IEEE Trans. on Software. Engineering*, 6, 6, Nov. 1980, pp 585-594.