Dependable System Development Model and N-Version Software Paradigm

Developing dependable, critical applications is not an easy task. The trend toward increasing complexity and size, distribution on heterogeneous platforms, diverse accidental and malicious origins of system failures, the consequences of failures, and the severity of those consequences combine to thwart the best human efforts at developing these applications. In this section, we describe methods to assist in the design and development of these critical, fault-tolerant software systems. In doing so, the following topics are covered: design considerations, a development model for dependable systems, and a design paradigm specific to NVP.

1- Design Considerations

The issues are primarily related to design diverse software fault tolerance techniques, but are also useful to consider for other types of software fault tolerance techniques:

1.1 Component Selection

One of the first major decisions to make is to determine which software functions or components to make fault tolerant. Specifications, simulation and modeling, cost-effectiveness analysis, and expert opinion from those familiar with the application and with software fault tolerance can be helpful in this determination.

1.2 Level of Application of Fault Tolerance

One of the most important early decisions impacting the system architecture is the level of fault tolerance application. There are two major questions determining the level of application of fault tolerance:

(1) At what level of detail should one perform the decomposition of the system into components that will be diversified?

(2) Which layers (application software, executive, hardware) must be diversified?

- **Level of Decomposition/Granularity**

While discussing checkpointing, we touched on this question. There is a trade-off between large and small components, that is, the granularity of fault tolerance application. **<u>Component size can be a determining factor in the effectiveness and cost of a fault tolerance strategy</u>**. <u>Small pieces of code enable lower error latency and make decision algorithms simpler and more precise</u>. However, small components will <u>increase the number of decision points, which limits design diversity</u>, since an increase in the frequency or number of decisions requires the agreement of variants at a higher level of detail. <u>A smaller modular decomposition</u> (i.e., smaller code segments between decision or checkpoints) <u>increases execution overhead in decision making and fault tolerance control</u>. Larger components favor diversity and higher variant independence. In addition, larger components result in lower execution overheads because of the lower frequency of executing the decision algorithm. However, larger component size increases the error latency because of increased synchronization delays (e.g., for NVP or NSCP) or rollback distance (e.g., for RcB). Having larger code segments between checkpoints and decisions may result in a cursory decision mechanism that is incapable of localizing the errors that occur. Related to this decomposition/granularity decision is a lower-level design issue regarding the placement of decision and recovery points within a fault-tolerant section of code and the choice of the data upon which to perform decision making.

- **What Layer(s) to Diversify/Extent of Diversity**

Diversity can be applied at several layers of the system (e.g., the application software, executive software or operating system, and hardware) and throughout the development effort (e.g., languages, development teams and tools, and so on). Additional diversity is likely to increase reliability, but must be balanced against cost and management of the resulting diversity

1.3 Technique Selection

Selection of which technique(s) to use is an important design consideration. This decision can be helped by input from performance analysis, simulation and modeling, cost-effectiveness analysis, design tools, and expert opinion (again, from those familiar with the application (domain) and with software fault tolerance).

1.4 Number of Variants

Not considering any economic impact, the number of variants to be produced for a given software fault tolerance method is directly related to the number of faults to be tolerated. There are, of course, both cost and performance effects to consider. A larger number of variants should increase reliability (if the number of related faults does not also increase). However, a larger number of variants in a recovery block scheme will also increase the execution time and cost. Similarly, in multiversion software (such as NVP), an increase in the number of variants will result in higher development and support costs.

1.5 Design Methodology

Using a design methodology that effectively considers dependable, faulttolerant software needs will assist in managing the complexities, realizing and handling the design and development issues particular to fault-tolerant software, and developing a dependable system. The guidance in the methodologies can provide valuable assistance.

1.6 Decision Mechanism Algorithm

Selection of which DM to use is another important design consideration. This selection can be helped by input from fault-handling simulation, design tools, and expert opinion.

2- Dependable System Development Model

The dependability-explicit development model provides lists of key activities related to system development phases. The requirements phase begins with a detailed description of the systems intended functions and definition of the systems dependability objectives. The following list summarizes the key activities in the fault tolerance process for this phase.

a- Description of system behavior in the presence of failures:

- Identification of relevant dependability attributes and necessary trade-offs;
- Failure modes and acceptable degraded operation modes;
- Maximum tolerable duration of service interruption for each degraded operation mode;
- Number of consecutive and simultaneous failures to be tolerated for each degraded operation mode.

The main objective of the design phase is to define an architecture that will allow the system requirements to be met. The following list summarizes the key fault tolerance activities and issues for this phase.

a- Description of system behavior in presence of faults:

• Fault assumptions (faults considered, faults discarded);

b- System partitioning:

• Fault tolerance structuring: fault-containment regions, errorcontainment regions;

• Fault tolerance application layers;

c- Fault tolerance strategies:

• Redundancy, functional diversity, defensive programming, protection techniques and others;

d- Error-handling mechanisms:

• Error detection, error diagnosis, error recovery;

e- Fault-handling mechanisms:

• Fault diagnosis, fault passivation, reconfiguration;

• Identification of single points of failure.

The realization phase consists of implementing the system components based on the design specification. Below is a summary of the key fault tolerance process activities for the implementation or realization phase

a- Collect the number of faults discovered during this stage:

• Use as indicator of component dependability;

• Use to identify system components requiring reengineering.

The integration phase consists of assembling the system components and integrating the system into its environment to make sure that the final product meets its requirements. Following is a summary of the key fault tolerance activities for the integration phase.

a- Verification of integration of fault and error processing mechanisms:

• Use analysis and experimentation to ensure validated faulttolerant subsystems satisfy dependability requirements when integrated;

• Use fault injection (multiple and near-coincident faults);

• Evaluate fault tolerance mechanisms efficiency;

• Estimate fault tolerance mechanism coverage fault injection experiments.

The dependability-explicit development model is provided to ensure that dependability related issues are considered at each stage of the development process. The model is generic enough to be applied to a wide range of systems and application domains and can be customized as needed. Since the key activities and guidelines of the model focus on the nature of activities to be performed and the objectives to be met, they can be applied regardless of which development methods are used.

3- Design Paradigm for N-Version Programming

A design paradigm for NVP because it contains guidelines and rules that can be useful in the design of many software fault tolerance techniques. It is generally agreed that a high degree of variant independence and a low probability of failure correlation are vital to successful operation of N-version software (NVS). This requires attaining the lowest possible probability that the effects of similar errors in the variants will coincide at the DM. Hence, the objectives of the design paradigm are:

• To reduce the possibility of oversights, mistakes, and inconsistencies in the process of software development and testing;

• To eliminate most perceivable causes of related design faults in the independently generated versions of a program, and to identify causes of those that slip through the design process;

• To minimize the probability that two or more versions will produce similar erroneous results that coincide in time for a decision (consensus) action of the N-version executive (NVX).

The design paradigm for NVP is illustrated in Figure1.As shown; it consists of two groups of activities. On the left side of the figure are the standard software development activities. To the right are the activities specifying the concurrent implementation of NVS. Table1 summarizes the NVS design paradigms activities and guidelines incorporated into the software development life cycle. (The table was developed by combining information found in (the table structure and initial entries) and (updated information on the refined paradigm).

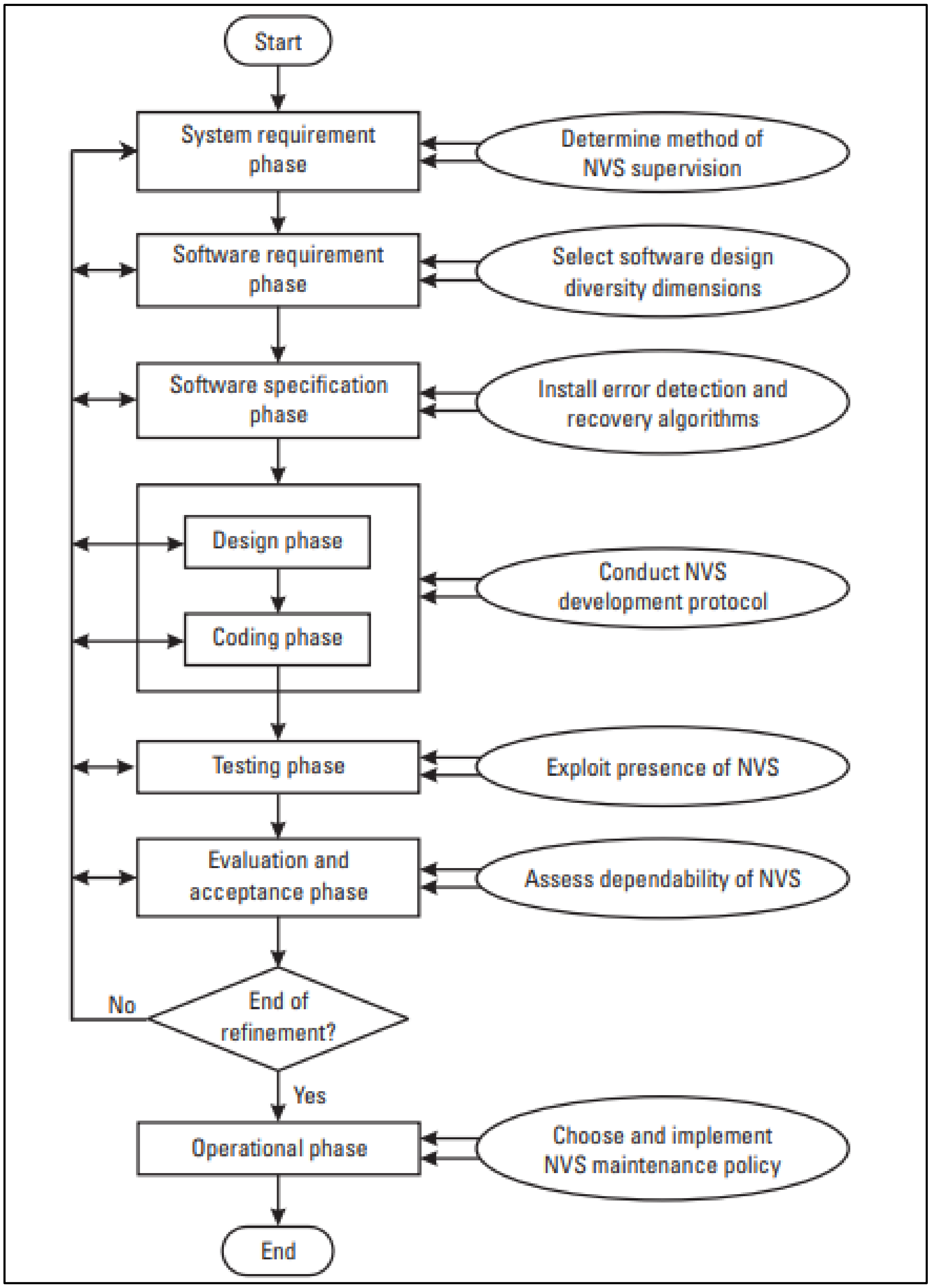
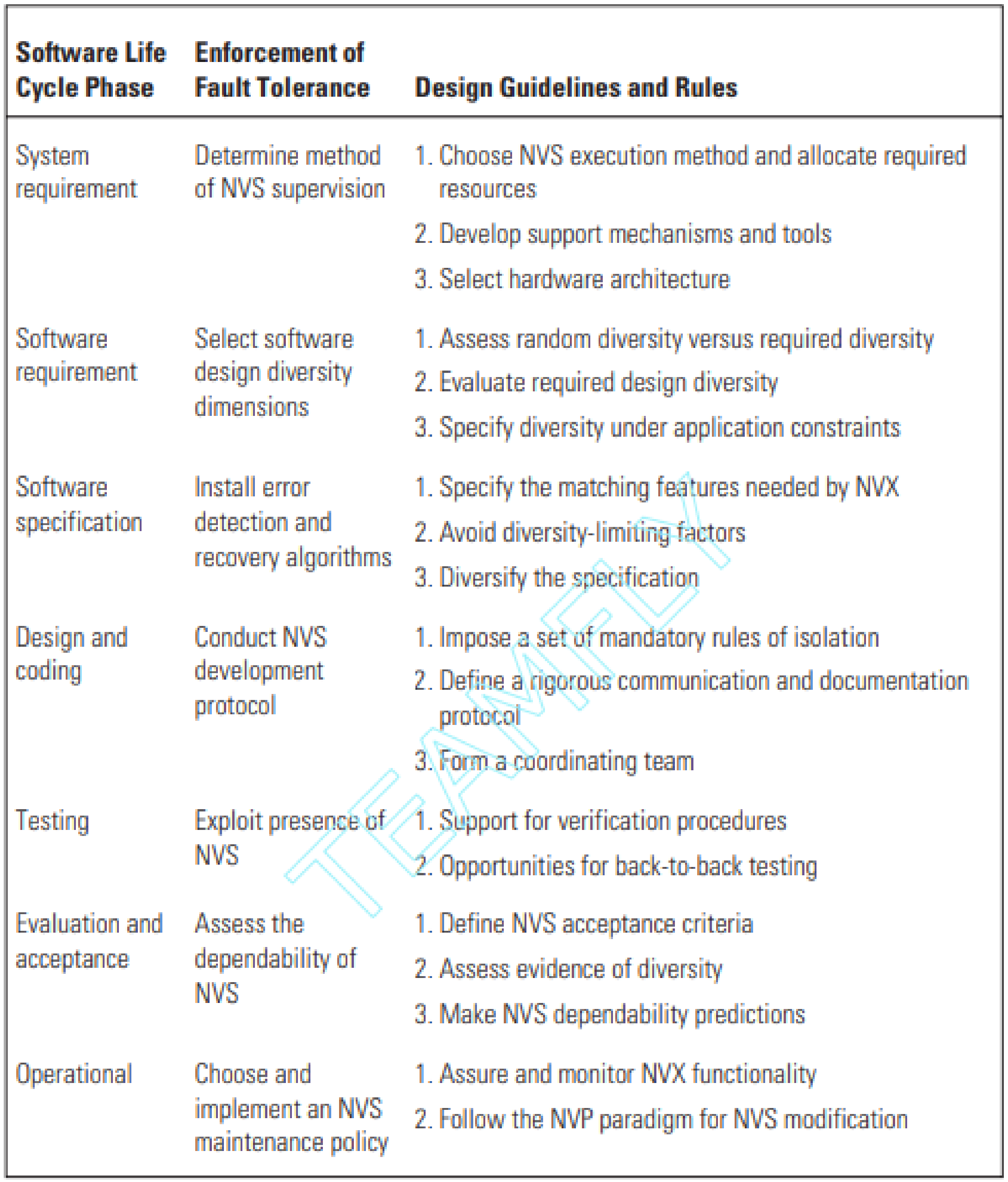

Start
System requirement phase
Determine method of NVS supervision
Software requirement phase
Select software design diversity dimensions
Software specification phase
Install error detection and recovery algorithms
Design phase
Coding phase
Conduct NVS development protocol
Testing phase
Exploit presence of NVS
Evaluation and acceptance phase
Assess dependability of NVS
End of refinement?
No
Yes
Operational phase
Choose and implement NVS maintenance policy
End

Figure 1 Design paradigm for N-version programming.

Table 1 *N*-Version Programming Design Paradigm Activities and Guidelines



Software Life Cycle Phase	**Enforcement of Fault Tolerance**	**Design Guidelines and Rules**
System requirement	Determine method of NVS supervision	1. Choose NVS execution method and allocate required resources 2. Develop support mechanisms and tools 3. Select hardware architecture
Software requirement	Select software design diversity dimensions	1. Assess random diversity versus required diversity 2. Evaluate required design diversity 3. Specify diversity under application constraints
Software specification	Install error detection and recovery algorithms	1. Specify the matching features needed by NVX 2. Avoid diversity-limiting factors 3. Diversify the specification
Design and coding	Conduct NVS development protocol	1. Impose a set of mandatory rules of isolation 2. Define a rigorous communication and documentation protocol 3. Form a coordinating team
Testing	Exploit presence of NVS	1. Support for verification procedures 2. Opportunities for back-to-back testing
Evaluation and acceptance	Assess the dependability of NVS	1. Define NVS acceptance criteria 2. Assess evidence of diversity 3. Make NVS dependability predictions
Operational	Choose and implement an NVS maintenance policy	1. Assure and monitor NVX functionality 2. Follow the NVP paradigm for NVS modification