Proceedings of the International MultiConference of Engineers and Computer Scientists 2008 Vol I
IMECS 2008, 19-21 March, 2008, Hong Kong

Fault Tolerance Through Automated Diversity in the Management of Distributed Systems

Jörg Preißinger *

Abstract—**Nowadays the reliability of software is often the main goal in the software development process. Despite more and more improvements in fault preventing techniques, it is a fact that faults remain in every complex software system. In contrast to hardware-faults, no concepts or mechanisms for fault tolerance of general software-faults became widely accepted. In this paper we present a new concept for the design of system management, that enables the tolerance of software-faults of the executed applications. We explain how the underlying resource management of a distributed system affects the occurrence of errors of a faulty application and how the re-execution of the application with altered resource management decisions prevents the occurrence of errors with certain probability. Our contribution is to motivate a new approach for fault tolerance in distributed systems and to give a general concept for system designers which states what decision spaces can be used to tolerate faults and how different alternatives in one decision space can be evaluated and generated. The concept is applicable for a large class of system models.**

Keywords: fault tolerance, software-faults, distributed systems, system management, diversity

1 Introduction

An improvement of the reliability and survivability of distributed systems is necessary due to the increasing use of these systems in environments, in which a system failure implies a financial disaster or even the harm of human beings. This improvement can only be achieved by combining fault preventing and fault tolerating techniques. On the one hand, fault preventing techniques during the software engineering process reduce the number of faults in software drastically. But on the other hand, despite exhaustive testing and model-checking techniques, bugs remain in complex software and thus especially in distributed applications due to the dependencies in concurrent problem solutions (cp. [1] and [2]). These remaining software-faults as well as hardware-faults need to be tolerated during execution to improve the reliability of systems.

*Institut für Informatik, Technische Universität München, 85748 Garching bei München, Email: preissin@in.tum.de

1.1 Fault Tolerance

Approaches to tolerate hardware-faults are well explored and established. The goal of these approaches is to tolerate transient or permanent faults of one or several instances of a hardware entity under the assumption, that some other instances of that kind of hardware work correctly. This implies that the hardware has no faults in its design, but faults appear in some of its produced instances by aging or environmental influence. Through structural redundancy faults of some components can be compensated by the correctly working ones and thus hardware-faults can be tolerated (cp. [3]).

1.1.1 Software-fault Tolerance

In contradiction to hardware-faults, software-faults (i.e. *bugs*) are faults in the design of the software; a software-fault is permanent and exists in *every* instance (e.g. every copy) of the software. If the techniques of fault tolerance for hardware-faults were used to tolerate software-faults, either every redundantly executed part of a software (e.g. module, function) computed the same correct result or produced the same error, for a certain input. For this reason the tolerance of software-faults is more complex, less understood and less established than the tolerance of hardware-faults (cp. [3]).

Today software-fault tolerance[1] can be differentiated into two classes: The first class covers exception mechanisms, which handle predictable problems such as communication errors or insufficient resources by explicitly defined or standardized functions. The second class covers approaches that try to cope with any kind of unpredictable, general design fault in software. The first class is very useful and established, but the fault tolerance in this class can be viewed more as part of the problem solution than as an approach to tolerate software-faults. The approach of *diversity programming* (N-Version Programming) is the only approach in the second class of software-fault tolerance, attempting to tolerate any software-fault: Starting from the same specification, several separated programming teams implement several versions of a soft-

[1] In this paper the term *software-fault tolerance* means the tolerance of *software*-faults, not in software implemented tolerance of faults.

ISBN: 978-988-98671-8-8

ware with equal functionality. The different versions can then be executed redundantly and faults can be tolerated, if they exist only in some of the versions (cp. [4]).

1.1.2 Problems of Diversity Programming

Several Studies and Experiments show that the errors made in the different versions are correlated due to several reasons ([5] p. 165): Common specifications, intrinsic difficulty of the problem, common algorithms, cultural factors, common platforms. The most important reason for the correlation of errors is the fact, that the complexity of software or generally of problem solutions differs. There nearly always exist some cases of input or some parts of the problem solution, where additional dependencies drastically increase the complexity of programming. In these cases programmers tend to make more errors, independent of their education, the used programming language or their cultural background (cp. [6]). The correlation of errors in the different versions of the software reduces the theoretically achievable reliability increase of diversity programming. Nevertheless diversity programming is a very expensive technique, because the programming and testing of not only one but several versions of a software needs to be done. The high costs combined with the moderate increase of reliability due to correlated errors makes diversity programming unprofitable in most cases in practice (cp. [7]).

1.2 Challenge

Unpredictable software-faults remain in complex software systems in spite of fault preventing techniques as testing and model-checking. Techniques to tolerate software-faults are needed to meet high reliability and survivability requirements of computer systems, particularly in the case of distributed systems due to the additional software complexity. These techniques must avoid the shortcoming of diversity programming in the sense that the development costs of software must remain profitable for practice.

1.3 Contribution

We present a new concept of software-fault tolerance based on checkpointing and re-execution with automated diversity in system management. We illustrate why certain changes in the execution environment of an application, e.g. decisions and strategies of resource management in the underlying operating system or middleware, avoid the re-occurrence of errors with high probability. We propose strategies to vary management decisions and to generate execution alternatives based on approximations of execution costs and on the approximated probability of tolerating faults.

1.4 Organisation

The remainder of this paper is structured as follows: section 2 provides some terminology and discusses what kinds of faults often remain in software. The concept of fault tolerance through automated diversity is described in section 3. In section 4 two examples are briefly explained to illustrate the concept. Related work is described in section 5. Section 6 concludes the paper and emphasizes resulting challenges for the future.

2 Software-faults

This section provides a short introduction to the used terminology and discusses what kind of software-faults are particularly hard to detect and remove by fault preventing techniques and therefore need to be tolerated.

2.1 Terminology

In the area of fault tolerance and fault prevention several kinds of faults and different aspects can be distinguished. For a comprehensive description of terms and definitions we refer to P. Jalote [3]; for the paper at hand we distinguish between fault, error and failure. A failure of a system occurs, if the system (viewed as black box) does not behave as specified. An error is an internal state of the system (viewed as glass box) which may lead to a failure. Whether the failure actually occurs depends on the execution that takes place after the error occurred. An error is the result of a fault, but the error occurs at runtime whereas the fault exists statically. This paper addresses software-faults in particular; these are permanent faults in the source code of software that may or may not cause an error during execution. In this case fault and resulting error both are commonly called *bug*.

Fault *prevention* techniques aim at preventing the software from faults during the engineering process. Testing, model-checking, bug-finding tools and reviews are used to reduce the number of faults in software. The aim of fault *tolerance* techniques is to avoid a failure in the system even if faults are present. Therefore errors must be detected and the causal chain to system failure interrupted. Fault tolerance with forward recovery changes an erroneous result to the correct result by using redundant information, e.g. crc-codes or raid systems. Backward recovery is based on former saved states that can be recovered. Usually the same execution is tried again from that saved states, thus errors caused by non-permanent faults can be tolerated.

2.2 Remaining Software-faults

Fault prevention techniques, in particular software tests, are valuable and very efficient to decrease the number of software-faults drastically. As stated in the introduction some faults remain in software even though excessive software tests and other fault prevention techniques were

done (cp. [8]). Some kinds of faults are more easy to detect and remove than others and particularly the kinds of faults that frequently remain in software need to be tolerated at runtime.

Apparently the difficulty to detect a software-fault by testing depends on the determinism of it's caused error and failure. If a software-fault causes an error and leads to a failure deterministically for a given input, the existence of this fault can easily be detected during the software testing process. If a fault causes - given the same input - only in seldom cases an error, it's detection is much more difficult and the chance is high that the fault stays undetected during the testing process. The un-determinism of the occurrence of errors in these cases is due to additional dependencies concerning the execution environment, which is the system internal resource management, e.g. scheduling, as well as the system external environment, e.g. incoming messages. Typically, the occurrence of those errors depends on timing (race-conditions), or on the use of resources, e.g. accessing wrong memory regions (not allocated, buffer-overflow, etc.).

It's not possible to influence the external environment, e.g. what messages reach a system, at least not transparently. In contrast, all internal causes of the occurrence of such an error can be influenced by the execution environment. We call faults *concretion dependent*, if the sum of all management decisions is decisive for the fault causing an error or not. In other words, if a fault in an application causes an error only in some of all possible combinations of management decisions of the underlying system, it is called concretion dependent fault. We define the *management* of the system as all software components, that are necessary to execute the problem solution written in a high level programming language on the given hardware; this includes compiler, binder, loader, operating system, communication middleware, etc.

On the one hand concretion dependent faults are hard to detect by fault prevention techniques and thus often remain in software despite high effort for quality assurance during the software engineering process. On the other hand the cause of errors by concretion dependent faults can – by definition – be avoided and thus tolerated due to their dependence on management decisions. Furthermore, if a software-fault in a closed system is not concretion dependent, it causes errors deterministically and thus is easy to detect and can be removed. In the following section we describe how management decisions can be altered to tolerate concretion dependent faults in applications.

3 Automated Diversity in Distributed System Management

The concept to tolerate software-faults described in this paper is based on mechanisms to detect errors on the one hand, and to perform rollbacks to saved system states on the other hand. The mechanisms are generally independent from the automated diversity that occurs after having detected an error and rolled back to a former saved state. The demand on the error detection mechanism is to enable a detection of a large class of software-faults, whereas the checkpoint-rollback mechanism is independent of the fault tolerance concept. Due to the limits of this paper we will postulate the existence of mechanisms for error detection and checkpoint-rollback on system level and refer to former work in the same project on error detection [9] and checkpointing [10].

3.1 General Idea

Particularly those software-faults that only appear in special execution environments or specific temporal execution sequences are often not detected by fault preventing techniques. The management's decisions on execution and resource management of the high level problem solution affect the occurrence of these errors. Error detection mechanisms are used to detect software errors, without classification or localization. The management performs a rollback to a former saved state after an error was detected. The management's degrees of freedom are used when re-executing from that saved state to vary the decisions of resource management based on collected application information. If the fault was concretion dependent, there is a probability that a different combination of management decisions does not lead to the occurrence of the same error again.

Viewed from a high abstraction level, one can say that the problem to tolerate a concretion dependent fault is a classical search problem on the possible combinations of management decisions. If the fault is concretion dependent, then - by definition - some combinations of management decisions will lead to errors whereas other ones will lead to the correctly executed and terminated system. In the following sections we will describe, *what* management decisions should be manipulated, *how* new alternative decision combinations can be generated and *which* of them should be chosen. On the one hand, the goal for efficient system execution must be met, whereas on the other hand the probability for the tolerance of faults should be maximized.

3.2 Decision Spaces

For the execution of a software written in a high level language, the decision space for the management, including compilation, deployment and execution, is tremendous. For practicability, we need a small amount of management decisions that can easily be altered and former decisions logged, but have a high impact on the probability of the occurrence of typical concretion dependent faults. It is difficult to get statistic informations about the kind of faults that remain in software after extensive testing, and more important, the reasons of their occurrence.

One large number of bugs are either in the field of erroneous memory management, meaning to overwrite data due to wrong pointers, loop variables, array boundaries, etc. (cp. [8]). Another large remaining class of faults, or more reason for the occurrence of faults or unspecified behavior, are race conditions. We use two classes of management decisions that affect these two fault classes to demonstrate the general concept of fault tolerance through automated diversity on management level. The concept is not restricted to those two classes of decisions and can easily be applied for other decision spaces. It depends on the whole distributed system, including application programming language, compiler, communication services and operating system, which decisions promise a high probability to be the trigger of errors.

One decision space that has an important impact on resource availability and in particular on the timing between cooperating processes in a distributed execution is the process placement. In a general model for distributed systems we presume to have a finite set of nodes N and a dynamically changing set of processes P. Processes $p \in P$ are created dynamically, communicate via message passing and terminate after finishing their calculations. For every created process the system management must decide on which of the Nodes N the process should be executed. The mapping of processes to nodes affects the execution of the interacting processes: first, processes executed on one node can communicate about 10^5 times faster than processes on different nodes via standard network. Second, resources as cpu and memory are shared through virtualisation and thus the total number of processes on one node affects the real time behavior of processes and therefore the relative timing to processes on other nodes.

Assuming that a distributed application consisting of three processes p_1, p_2 and p_3 has a software fault in synchronization that leads only to an error if p_1 reaches a critical section before p_2 does. If this application is executed on a distributed system consisting of two nodes, the occurrence of this error may depend on the placement decision: if p_1 is executed on one node whereas p_2 and p_3 are executed on the other, the probability for the occurrence of the error is much higher than in the case of p_2 being executed on one node and p_1 and p_3 on the other. Obviously the probability for the occurrence of the error lies somewhere between if p_3 is on one node and p_1 and p_2 share the node. We will use the process placement in a cooperating distributed system as an example decision space to illustrate and explain the general concept, which can be applied to any decision space in system management.

3.3 Generating Alternatives

After an error is detected, the management needs to perform a rollback to a previously saved state and re-execute the application with changed management decisions. A change of management decisions requires the possibility to generate different alternatives and to evaluate them. First we will explain how alternatives can be generated given an evaluation function $f_e : A \times A \mapsto I\!R$ that maps the difference between two alternatives a_1, a_2 to a real number. Later we will explain how to design f_e. Suppose that, comparing two alternatives a_1, a_2, the higher the chance that an error that occurred when performing a_1 will not occur again when performing a_2, the higher the value of f_e will be for a_1, a_2. An alternative $a \in A$ is one of all possible outcomes in the decision space. In the example of the process placement decision, the alternatives are all possible mappings of actually existent processes to the available nodes. More precisely A is the set of all possible partitionings of the set P of actually existing processes into $1 \ldots N$ partitions, where the processes are distinguished but the nodes are presumed to be equal.

The complexity of generating a new alternative for a given decision and given evaluation function f_e depends on the decision space. If the set A of alternatives is large, we propose the usage of search algorithms to generate a new alternative starting from the one that lead to the error. For example hill-climbing (cp. [11]), simulated annealing (cp. [12]) or genetic search algorithms (cp. [13]) can be used to generate a new alternative. Which search algorithm should be chosen depends on the properties of the decision space. For the process placement a genetic search algorithm can be used, because the genetic operations *crossover* and *mutation* can be applied for the partitioning easily. Given one partitioning, mutation is to move one random process from one to another set in the partitioning. For the crossover operation, we suggest to divide the set P of processes into two disjunct subsets P_1, P_2. Then the partitioning of the subset P_1 from one alternative is combined with the partitioning of P_2 from another alternative. With these operations the chance to get a new alternative with good evaluation in few populations is high.

If the decision space is small, the new alternative can be generated algorithmically from the old one regarding the evaluation function f_e. When designing a system and identifying decision spaces, one way for the generation method of new alternatives must be chosen and implemented.

3.4 Evaluating Alternatives

The requirement to generate a new alternative for a management decision that lead to an error is the evaluation function f_e, that expresses the difference in achieving the system goals between the last used alternative and a new one as numerical value. This evaluation function represents the system strategy for fault tolerance, which is in most systems a tradeoff between system performance and the probability to tolerate faults. We propose to construct the evaluation function f_e as a combination of two

functions respecting those two important system criteria.

3.4.1 Cost

One important goal of every system is performance. Usually the strategies in management aim at system performance, this implies that the decisions, after which an error occurred, usually preferred the alternative with most performance. The loss of performance when trying another alternative is expressed by the *cost function* $f_c : A \times A \mapsto I\!R$. The cost function $f_c(a_1, a_2)$ expresses the relative performance loss as a real number, when using alternative a_2 instead of a_1. The cost function is one part of the evaluation function f_e. The problem is how to construct the cost function, or in other words an adequate approximation of the relative loss of system performance for two given alternatives in a decision space. Again, the cost function depends on the decision space as well as the system details and must be developed during system design.

Constructing a practically relevant approximation for the effect of different process placements on system performance is a difficult task. We based our cost function on the work of C. Rehn who used compiler analysis of communication dependencies and execution times to approximate the real parallel execution times for an optimization of task placement in distributed systems [14]. The important factor for a task placement that optimizes performance is not the cpu load in the moment of the placement decision, but the communication dependencies that influence which processes can be executed in parallel if placed on different nodes and which process communication is fast on the same node or slow via network. Based on the compiler analysis we get an approximation of the system overall execution time dependent on process placement. Additionally we take the migration costs of processes into account, if already placed processes need to migrate to another node to achieve an alternative process placement starting from a saved state. Let *runtime()* be the algorithm that approximates the overall system runtime for a given process placement $a \in A$ from a saved state including migration costs, then the cost function is $f_c(a_1, a_2) = runtime(a_2) - runtime(a_1); a_1, a_2 \in A$. We will not give details on the approximation algorithm, because this paper focuses on the general concept rather than the details for one implementation, but the example should illustrate the concept. Details on the approximation can be found in [14].

3.4.2 Variance

The second part of the evaluation function f_e, that should affect the management decision after an error occurred, is the probability to tolerate the fault in the sense that the same error will not occur again at that point of execution. It is not possible to approximate this probability, because it depends on every single fault, which changes in decisions and which alternatives affect the occurrence of the implied error. As we argued in section 2.2, concretion dependent faults only lead to an error if certain decisions in management are taken, which often affect timing or resource management. This leads to an important assumption: statistically the *distance* in the dimensions *time* and *resource consumption* between two alternatives is directly proportional to the probability, that an error which occurred in one alternative caused by a concretion dependent fault will not occur in the other alternative. This assumption is based on the locality principle of code and data. If for example the mutual exclusion to access a critical region is faulty and we suppose that in one alternative a_1 the timing lead to the erroneous case that two processes entered the critical region at the same time, then the more the time interleaving of those two processes gets changed, the higher is the chance that the error will not occur again. Suppose a process to overwrite data that is allocated directly after an array with erroneous boundaries. Again the greater the distance in resource management, in this case the changes of memory layout, the higher the chance that there is no valuable data following the array.

We call the *distance* between two alternatives in time or resource consumption the *variance* of two alternatives. The variance function $f_v : A \times A \mapsto I\!R$ expresses this relative distance between two alternatives of one decision space as real number. The construction of a variance function is to a certain extent intuitive and not entirely measurable. However, at system design time a function must be constructed that approximates that distance efficiently. The complexity of this task is again dependent on the decision space, but we will illustrate the construction for the process placement example. Let two processes p_1 and p_2 cooperate via message passing. The execution timing of those processes is influenced by the decision if they are executed on the same node, or on two different ones. If they are on the same node, the communication via message passing is very fast, if executed on two different nodes the time messages need for transportation will have a large impact on the concurrent execution. We express this in a boolean auxiliary function $f_a : A \times A \times P \times P \mapsto \{0, 1\}$. f_a compares the placement of two processes $p_1, p_2 \in P$ in two placement alternatives $a_1, a_2 \in A$ and results to 0, if the two processes are either on one node or on different nodes in *both* alternatives. If they are placed on the same node in one alternative and on different ones in the other, f_a results to 1. The variance function can then be defined as the sum of all changes of combined process-pairs:

$$f_v(a_1, a_2) = \sum_{\forall p_i, p_j \in P} f_a(a_1, a_2, p_i, p_j)$$

3.4.3 Strategy

The cost function f_c and variance function f_v need to be combined to the evaluation function f_e, which is used to generate new alternatives via search algorithms. The weighted combination of cost function and variance function expresses the tradeoff between system performance and probability to tolerate faults. This tradeoff has an effect only after an error is detected an a rollback performed. A strategy that lays more stress on a low value of the cost function will execute the system similarly efficient from the saved state as the time before, but the chance that the same error occurs again will be relatively high. If a high value of the variance function is focused in the strategy, the re-execution of the system part will be less efficient, but the chance to tolerate the fault will be higher. If the same error occurs again, the rollback can be performed another time, but the decision should be chosen differently. So we propose to use a dynamically changing strategy for the combination of f_v and f_c such that first, f_c is weighted heavier to keep the system efficient, but the more often the same error occurs again, the more f_v is weighted more to increase the probability to tolerate the fault. Due to the dynamically evolving evaluation function, the management tries other alternatives for a given decision every time the same error occurs again. Thus the chance to find a combination of management decisions in which the error does not occur rises.

4 Examples

In the following we briefly describe two examples for decision spaces with evaluation functions to illustrate the general concept.

4.1 Process Placement

As already mentioned above, the process placement of cooperating processes in a distributed system has an impact on execution timing and resource availability. Cost and variance functions were already described above. A genetic search algorithm can be used to generate the new alternatives based on the evaluation function, which is dynamically changed as follows: in the first retry, we use lowest cost

$$f_e(a_1, a_2) = \frac{1}{f_c(a_1, a_2)}, with f_c \neq 0$$

the second retry uses the best ratio between variance and cost:

$$f_e(a_1, a_2) = \frac{f_v(a_1, a_2)}{f_c(a_1, a_2)}, with f_c \neq 0.$$

If the error occurs another time, the variance function is maximized without respecting the costs:

$$f_e(a_1, a_2) = f_v(a_1, a_2).$$

Due to the genetic search, even if the evaluation function is unchanged, different alternatives will be generated. Race conditions, i.e. faults that cause their errors dependent on the execution timing of concurrent processes, can be tolerated with high probability by altering the process placement. Due to the changing number of processes on one processor as well as the different communication delay, the interdigitation of process execution can be altered essentially and thus the fault tolerated.

4.2 Memory Layout

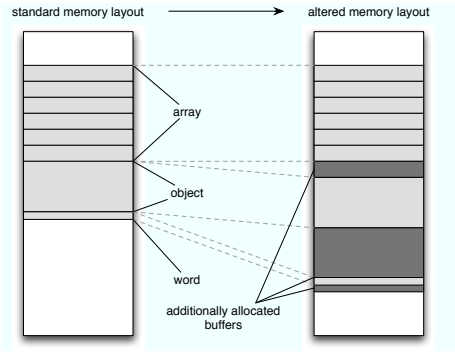

Figure 1: decision space *memory layout*

Faults in memory management are a large class of typical software bugs. The proposed concept can also be used to alter the memory layout of processes to tolerate some of these faults. We propose a very simple way to change the memory layout here to demonstrate a decision space, where cost function and variance function are trivial in contrast to those of the process placement. In many memory related faults, the memory region that is located directly after the allocated region gets overwritten. One solution for that problem is to allocate an additional buffer in the size of the stored data at the next address of allocated memory. Our decision space has only two alternatives, usual memory layout and that with additional buffers, as illustrated in fig. 1. In a decision space with only two alternatives, the evaluation function simply changes the memory layout from normal to buffered mode. This is an example of a very simple use of the proposed concept, to illustrate that even minor changes of decisions with acceptable expenses in development can raise the fault tolerance of distributed systems. Of course the altering of memory layout can be designed far more complex and thus lead to a higher probability to tolerate faults.

5 Related Work

The introduced concept must be generally distinguished from diversity programming, where different versions of the application need to be developed. In the proposed concept the diversity is implemented via altering management decisions such that a faulty application executed on the system will be executed correctly with a certain probability. The software development effort must be invested once at system design time, not for every application.

Wang et al. describe an approach called *"progressive retry"* to tolerate faults in message passing systems via reproducing incoming messages with altered order in case of an error [15]. Wangs work is one example for the use of diversity on management level to tolerate faults. In that case the decision space is the order of incoming messages. Wangs approach encourages the general concept presented in the paper at hand.

Some interesting work in the field of security is done in recent years using randomized diversity in system management. In contrast to the work at hand, the goal of that work is not to tolerate software-faults, but to change the uniformity of systems which is the main reason for the vulnerability to attacks [16]. E.g. Xu et al. prohibit attacks based on *unauthorized control information tampering (ucit)* as buffer overflows etc. via dynamically randomized memory layout in their project *Transparent Runtime Randomization (TRR)* [17].

6 Conclusion

In the paper at hand we presented a general concept to tolerate software-faults. The main contribution of our work is to present a new approach of fault tolerance in system management without dependencies to special system models or concrete management implementations. We argued that most faults that remain in complex software after exhaustive testing are concretion dependent faults: the combination of management decisions implies if the fault causes an error or not. Thus one way to tolerate those faults and thus improve the reliability of the software is automated diversity in the management: after an error gets detected, the system is re-executed starting from a saved state with an altered combination of management decisions. We presented possibilities to identify decision spaces, generate alternatives and construct evaluation functions. Further we illustrated the general concept on the basis of the process placement decision.

We want to challenge system designers to implement fault tolerance based on this concept and gain practical experiences with different decision spaces, evaluation functions and methods to generate alternatives. Practical experiences are necessary to identify the effectivity of the proposed concept as well as the efficiency of the concept for different decision spaces to improve the reliability of software systems.

Acknowledgment

The author would like to thank Prof. Dr. P. P. Spies for his valuable comments in discussions on the proposed concept. Further thank goes to Prof. Dr. U. Baumgarten for his support of this work.

References

[1] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[2] Gerard J. Holzmann. The logic of bugs. *SIGSOFT Softw. Eng. Notes*, 27(6):81–87, 2002.

[3] Pankaj Jalote. *Fault tolerance in distributed systems*. Prentice-Hall, 1998.

[4] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. roc. the First IEEE-CS International Computer Software and Applications Conference (COMPSAC 77), Nov 1977. Chicago.

[5] Israel Koren and C.Mani Krishna. *Fault-Tolerant Systems*. Morgan-Kaufman Publishers, San Franciso, CA, 2007.

[6] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Softw. Eng.*, 12(1):96–109, 1986.

[7] R. Feldt. Generating diverse software versions with genetic programming: an experimental study. *IEEE Proceedings - Software*, 145(6):228–236, 1998.

[8] M. Sullivan and R. Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pages 2–9, 25-27 June 1991.

[9] Jörg Preißinger and Alexander Mayer. Integrating design by contract in insel. In *Proceedings of the International MultiConference of Engineers and Computer Scientists, IMECS'2007*, pages 1037–1043, Hong Kong, March 2007.

[10] Jörg Preißinger and Mark Pflüger. Compiler supported interval optimisation for communication induced checkpointing. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'07*, volume II, pages 550–556, Las Vegas, NV, June 2007. CSREA Press.

[11] Winston P. Henry and Patrick Henry Winston. *Artificial Intelligence*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[12] S. Kirkpatrick, Jr. Gelatt, C. D., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[13] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[14] Christian Rehn. Dynamic mapping of cooperating tasks to nodes in a distributed system. *Future Generation Comput. Syst.*, 22(1):35–45, 2006.

[15] Yi-Min Wang, Yennun Huang, W. Kent Fuchs, and Chandra Kintala. Progressive retry for software failure recovery in message-passing applications. *IEEE Trans. Comput.*, 46(10):1137–1141, 1997.

[16] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanovi. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, 2005.

[17] Jun Xu, Z. Kalbarczyk, and R.K. Iyer. Transparent runtime randomization for security. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 260–269, 6-18 Oct. 2003.