N-version software module requirements to grant the software execution fault-tolerance

Denis V. Gruzenkin, Alexey S. Chernigovskiy, Roman Yu.Tsarev
Siberian Federal University, Department of Informatics

Annotation

N-version programming is one of the approach ensuring high reliability and fault-tolerance of software on the basis of program redundancy and diversity. This approach ensures that faults of one of the versions of an N-version software module will not result in malfunction of the module operation process. N-version software realization, as a rule, depends upon capacities and preferences of the teams of designers and developers. This work is an attempt to denote basic requirements, which should be met at design of N-version software to minimize occurrence of possible program faults and influence of the modules versions on one another. The requirements to versions (program modules) of N-version software allow to ensure high-level reliability and fault-tolerance due to elimination of possible influence of separate versions on each other. A special attention has been paid to their interaction, which should not have any impact on operation of the other components. For realization and research of N-version software developed taking into account the defined requirements an N-version software execution environment has been developed. Testing of the N-version software execution environment has demonstrated expediency of a component architecture application and high efficiency of N-version programming as a method of fault-tolerant software development.

Keywords: N-version software, requirements, execution environment, software reliability

Introduction

Software plays a crucial role in modern information systems. In this regard, reliability of the software as a part of information systems is one of the most important criteria at the stage of software design and development. There is a number of approaches to provision of high-level reliability and fault-tolerance of software which are based on introduction of time, informational or program redundancy [1], [2].

One of the approach ensuring high reliability of software on the basis of program redundancy is N-version programming (NVP) [3]. NVP in combination with modern methods of program testing and validation provides high level of reliability for software component execution. This approach ensures that faults of one of the versions will not result in malfunction of the module operation process, and consequently, of the application software and the whole information and control system [4], [5]. Such fault-tolerance is important for safety-critical information systems [6].

At present, there is a wide range of information processing hardware where software of an information controlling system can be deployed [7]. The Concept of N-version software implies parallel run of its modules versions [8]. Optimal operation of parallel program versions requires a choice of an appropriate computation system with parallel architecture.

The most suitable, but at the same time, more expensive one is an architecture with distributed memory, where each computational node has its own processor and memory [9]. However, using such computing system is not always possible. Other kind of a computer system which can be used for execution of versions of information and control system N-version software is a system with shared memory, where there is a number of processors (or cores of a multi-core processor) and single shared memory [10]. On the other hand, a shared memory can become a big problem at N-version software run. That is why there is a need to bring some requirements for design and deployment of N-version software for computer systems of that class.

A general approach to development of highly reliable software based on the N-version principle does not have clearly defined requirements, and N-version software realization, as a rule, depends upon capacities and preferences of the team (teams) of designers and developers. This work is an attempt to denote basic requirements, which should be met at design of N-version software to minimize occurrence of possible program faults and influence of the modules versions on one another. Besides this, realization the stated requirements ensures conveniences of further work with the software and its support.

The NVP concept presupposes that in the components realizing identical functions, a potential fault can happen in different points. This allows detecting a fault and avoiding failure of the software [11]. It is supposed that independence of potential faults of different module versions of N-version software can be achieved by means of diversity at development of N-version software [12]. As it was stated above, in this approach, potential faults occurs in different points of a program code, but such independence of faults lies at the level of source codes, which is a key problem. At the stage of N-version modules execution fault independence is lost due to the possible interactions of the modules versions have been neglected in the framework of the whole software exactly at the runtime stage.

In practice, systems with shared memory are most frequently used: computers with multicore processors, SMP computers [13]. For the shared memory computing systems, the modules versions can work in a unified address space of memory, share the same resources of the operating system, and so owing to this, additional dependencies between the modules of the N-version software emerge. Hence, a fault of one module version can result in a fault of other versions or even in a failure of the whole software system. This discredits reliability of N-version software and the whole method in general. Solution of this problem lies in preserving the diversity at the modules design stage and also in transferring it to the N-version software run stage. The requirements which should be met by the developed N-version modules of the software used in the safety-crucial areas were formulated to solve this problem.

Requirements for design and development of N-version software

N-version model of software design is an outstanding example of modular design. Each of the N-version software versions represents a program module. Further, the article will consider program modules as versions of a software module, which, in its turn, features an abstraction realized via concrete program modules, (i.e. versions).

Module architecture of N-version software system will be efficient only in case if the set of requirements below has been satisfied.

Dynamic connection of the program modules

A dynamic connection of the program modules allows to replace the program modules of the system during its operation. Program module replacement support during execution demands a dynamic connection. The replacement of the system program modules should be done with no additional re-linking or recompilation of the program.

However the need to replace the program modules during the program system operation arises rarely, nonetheless, in a crucial application, a supporting mechanism for it may be very desirable. Support of a program module replacement during the program system run stipulates a necessity for a dynamic connection.

The importance of the modules dynamic connection can be estimated through the example of an application composed of the modules which can not be joined during runtime. If a replacement of one of the modules of such system is required, it will require to relink or recompile the program statically and to deploy it anew. An application composed of a set of modules that need to be statically relink every time one of them is replaced is equivalent to a monolithic application.

This requirement is equivalent to hardware methods of ensuring reliability by means of hot swap [14]. In hardware when a failure occurs, a defective component is replaced by a new one from a set of spare parts. Employing such approach for software will allow increasing reliability of the program system. However, there is an important difference between software and hardware. In most cases, software presents a completed and compiled full code of a program; this code is loaded to the computer memory for execution. That is why there is no need to compile the whole program system when separate dynamically connected modules are available.

Besides, the given requirement implicitly indicates that every program module should operate with its own memory data. This means that two program modules can not share memory as the memory allocated for the module under replacement has to be cleared and by no means should influence the operation of other modules and the whole program system.

The specified possibility of accessing the same memory block by independent program modules can cause correlated errors in the program modules of the N-version software system, which in its turn will cause troubles in assessment of the program modules operation results and will lead to lowering the reliability factor of the program system. That is why access of several program

modules to single memory block is worth excluding at all to avoid negative consequences.

Encapsulation requirement

This requirement results from the program module dynamic connection requirement. To form a software system, the program modules connect to each other through interfaces. If one of the program modules needs a replacement, this requires to disconnect the old one from the system and connect a new one. The new program module shall be connected the same way as the old one was, otherwise the components will have to be recoded, recompiled and relinked. Thus, the program modules and clients have to remain their interfaces unchanged. They must be encapsulated.

Let us define some terms connected to encapsulation. A program or a module using the other program module is called a client. A client connects to a program module via an interface. If a program module changes without a change of an interface, no changes in the client will be required. Likewise, if a client changes without any changes of the interface, there is no need to change a program module. However, if changing either a client or a program module causes changing of the interface, the other side of the interface also needs to be changed.

So, to use the advantages of dynamic connection, the program modules and clients should not change their interfaces. They should be encapsulated. The details of the client and program module realization should not be reflected in the interface. The more reliably an interface is isolated from the realization, the less possibly it will change after modification of a client or a program module. If an interface does not change, changing of a program module has a minor effect for the software system as a whole.

A need for isolation of the client from the realization details imposes a number of important requirements on the components. Such requirements are listed below:

a) A program module must hide the used programming language. Any client must have a possibility to use a program module irrespective of the programming languages used. Reveling the language of realization makes new dependencies between the client and the program module.

b) The program modules must be distributed in binary form. Indeed, as they should hide the realization language, they need to be delivered being already compiled, linked and ready for usage.

c) There should be a possibility to upgrade the program modules without affecting already existing clients. New versions of the program module should be capable to work with both the new and the old clients.

d) There should be a possibility to transparently move the program modules in the network. It is required that both the program module and the system using it could be executed within one process, in different processes or on different machines. A client should take a remote module the same as a local one. Should one work with a remote module otherwise than with a local module, this would

require recompilation of the client every time when the local module moves to other location of the network.

Inter-module access (protection)

This requirement provides safe interaction of the program modules in the framework of the program system. Execution of incorrect or erroneous code accidentally or deliberately inserted to the structure of one of the program modules should not have any unforeseen impact on the state of other modules and even more so, on the system in general. Inter-module interaction needs to be organized so that it will eliminate the module operation troubles caused externally, namely, unallowed access of the other modules. For this purpose, it is enough to ensure that all the accesses to any program module are performed only by means of the instruments prescribed by the developers of this very program module, i. e. via interface.

The aim of inter-module protection is to ensure safe interaction of the program modules within the system. It is impossible to provide faultless operation of a program module picked at random, in other words, to detect and remedy all the errors made during the module development [15]. Nevertheless, inter-module interaction can be organized in such a way that is sufficient to eliminate disfunctions in the program module operation caused by external reasons, namely, unallowed access by the other program modules. It is enough to have all accesses to this program module carried out only via the interface. Any interaction that by-passes the interface means a potential violation of internal logic of the program module and its unpredictable behavior later on.

In the high-level programming languages, the correct interaction of the program objects, including program modules, is ensured by a type system, by mechanisms of scopes and access modes. Yet all these mechanisms are non-operational at the binary level after compilation of the program. A robust protection, for example, in case of using operations of direct memory access or transfer of control on dynamically calculated addresses is not possible without proper checks during runtime of the program [16].

Let us consider a set of all exported program objects of the module and a set of the module internal objects, links to which are someway transferred to the other program modules. This set shall be called an interface of the module. The purpose of inter-module protection is to guarantee that the objects which are not included into the module interface are impossible to be read or modified from the other modules. In order to prevent a module that has access to the interface objects of the other module from getting access to its internal objects, control over observing of the following conditions should be ensured.

a) Control of data bounds. As variables are often located in the continuous memory block, the ability to overrun the block taken by one variable means an ability to access the neighboring variables, which is an access violation.

b) Control of code bounds. Distinguishing data from code is significant for protection. Usage of the some other module code instead of data means a possibility of dynamic modification for this code. Usage of the data of other

program module instead of a code means a possibility to interpret random data as machine's commands, which results in unpredictable behaviors of the program module.

c) Control of memory free. In case of reuse of a memory area, it is necessary to provide that the pointers stored during the first usage can not be read during further use of this page.

d) Control of pointers to the destructed objects. A pointer to the object which lifetime has already ended can result in an unpredictable behavior, and in case of reuse of the memory – to the access violation.

Absence of inter-module protection leads to emerging of correlated errors. As it is described in [17], correlated errors between the versions can increase the total probability of an error occurrence by several orders of magnitude more.

Let us consider the program that consists of three versions (program modules) and is tolerant to failures of one version for any input data. Let us suppose that probability that the version will give an incorrect result equals $q = 10^{-4}$, i. e. an average incorrect output happens one time to 10 000 runs. If the versions are stochastically independent, then the error occurrence probability for the program with three versions is

$$q^3 + 3\,q^2\,(1-q) \approx 3 \cdot 10^{-8}.$$

Now let us suppose that a stochastic independence can not be applied, and that there is one deficient mode typical for two out of these three versions (program modules) and emerges in average once for a million runs (i. e. approximately one out of 100 bugs of the version occurs due to a common mistake). Every time when this bug happens, the program fails. The error probability of the three-version system now increases to over 10^{-6}, which is more than 30 times the error probability of the uncorrelated system [17].

Independence of the program modules from the programming language

This requirement partially results from the encapsulation requirement stated in the above. Another reason for such requirement is economic expediency: the broader the number of supported programming languages, the greater the number of external developers who can write the necessary program modules. Besides, different versions of compilers have different mechanisms of error detection in the code at the stage of compiling and linking.

Everyone who has experience in programming knows, that a programming language can drastically influence the quality of the software being developed. So, it is supposed that a program written in Assembler tends to be more bug-prone, than that in the high-level language [18]. The nature of bugs can differ. For instance, for programs written in C there is a possibility of the buffer overflow. Such bugs would be impossible in language that strictly manages memory. Bugs resulted from incorrect use of the pointers, which is not a rare case for the programs written in C, will not happen in Fortran which has no pointers [17], [19].

Various programming languages can have various libraries and compilers, which, as a user expects, will have uncorrelated (or better negatively correlated) bugs.

Certain languages can be more adaptable to this aspect than other ones. For example, capacities of Lisp for expression construction are more suitable for some tasks of artificial intelligence than those ones of C or Fortran [20].

In this case, if all the versions are written in one most appropriate language, then different versions can encounter with correlated errors. If the versions are written in different languages, then the error rate of the N-version software system can be lower, as possible errors will not be correlated. Similar observation is also applicable to using the diversity in other aspects, such as development environments or instruments.

It is obvious than independence of the modules from the programming language can not be unimportant for development of fault-tolerant N-version software. The principle of diversity is ensured by development of different versions of the modules implementing different languages. All this allows to decrease a possibility of simultaneous occurrence of faults in different module versions and to increase general fault-tolerance of the N-version software system.

Results and Discussion

For realization and research of N-version software developed taking into account the above defined requirements an N-version software execution environment (NVX) has been developed on the grounds of the component approach [21]. N-version software execution environment consists of several components and a set of interfaces (figure 1).

The main specific feature of the developed NVX is runtime support for the program modules realized as independent components which can be executed in the processes separate from the NVX. Moreover, the program components can be located on different computers and interact with NVX by means of a network. Firstly, this solves the performance problem of a computer; secondly, provides the protection of the program modules from each other and from errors of runtime environment.

Other functional modules of NVX are also realized as separate components, which enables upgrading and replacement of NVX separate parts with minimal efforts. Particularly, depending on the applied task that can be solved by N-version software, the subsystem of making a decision about correctness of version operation results can be replaced with more suitable variant taking into consideration characteristics of specific task.

The most significant task is protection of the NVX from destructive external influence, so a user interacts with the runtime environment only via a special interface component. A component of the user interface interacts with the environment component only via special interfaces provided by the environment; that allows protection of the environment from incorrect actions of the user.

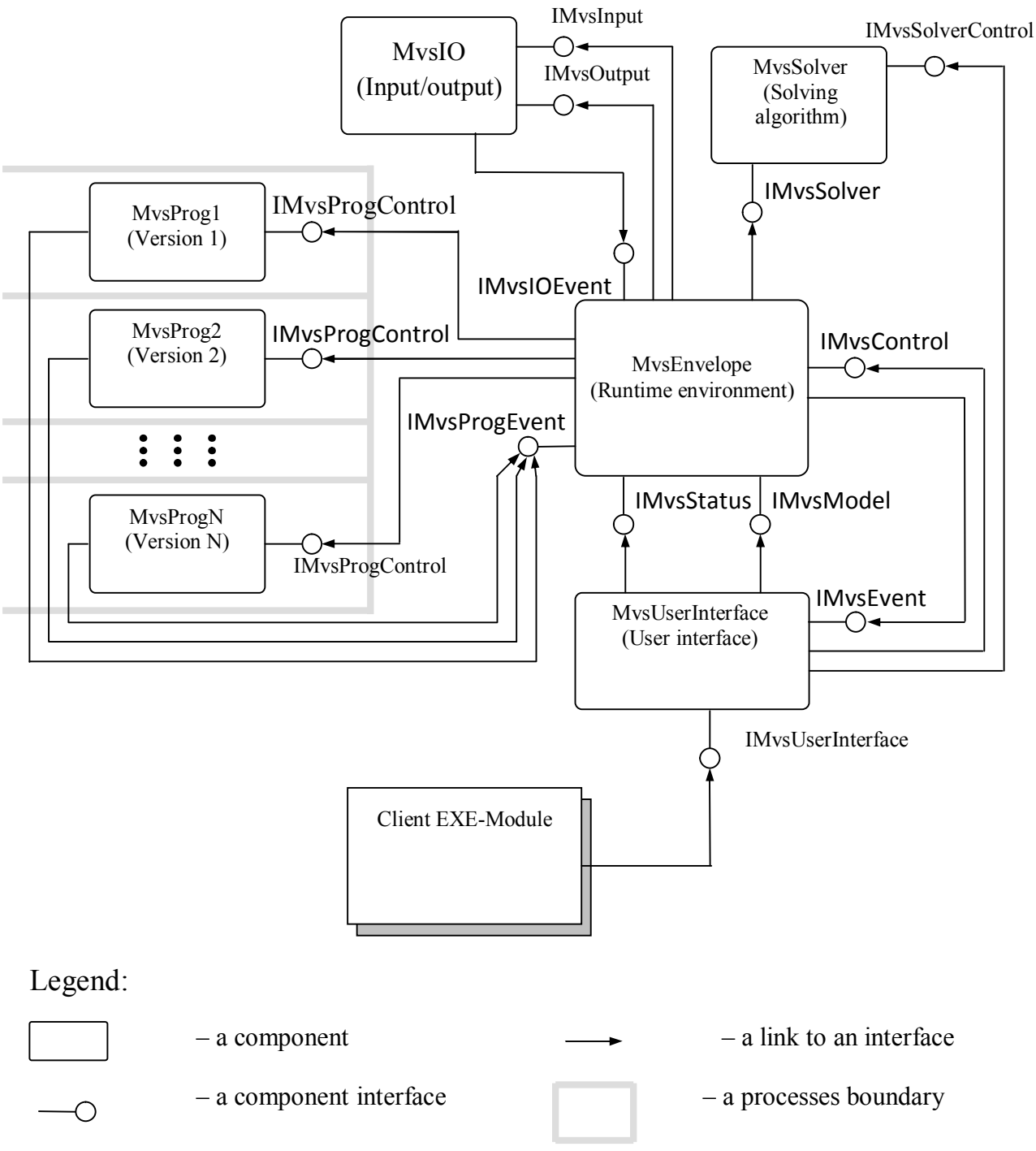

Fig. 1. A model of N-version software based on the COM

Development of this runtime environment for N-version software strived the research purposes; that is why:

- N-version software program modules have been developed taking into account the proposed requirements;
- The NVX input/output components do not interact with any external objects and have been designed to emulate such interaction;

- Data processed by N-version software have been modelled , for the purposes of testing and definition of the NVX properties;
- Errors occurring in the program modules have been injected at the development stage and designed for testing and analysis of the consequences for the NVX and N-version software [22], [23].

The results of testing of the N-version software developed subject to the described requirements in the suggested runtime environment are specified below. Testing was performed with changing of various parameters, and allowed to find out strengths and weaknesses of N-version approach to realization of fault-tolerant software. Table 1 specifies the parameters of errors injected to the realized modules.

Table 1. Errors injected into the program modules

N	Occurrence probability	Error	Consequences of the error
1	0.045	Arithmetic computation error	fault
2	0.038	Arithmetic computation error	fault
3	0.015	Arithmetic computation error	fault
4	0.18	Memory access error	failure
5	0.072	Logic error, circularity	failure
6	0.031	Logic error, infinite recursion	failure

Analyzing the results of testing carried out with the N-version software runtime environment; one can draw the two main conclusions:

1) The developed runtime environment worked reliable during testing even if errors occurred in individual program modules;
2) Rising the number of program modules by one allows to lower probability of the whole system fault by order of magnitude greater (see table 2). Fault probability assessment for the runtime environment was performed with the use of the software reliability assessment model developed by the authors earlier [24].

Table 2. Correlation of the whole program system fault and a number of program modules

Parameters	Amount of program components			
	2	3	4	5
Probability of fault of the program system	0.04783	0.00186	0.00029	0.00003

Testing results of N-version software in the framework of the developed environment of N-version software run support the theoretical conclusions about the provision of fault tolerance with three of more modules.

The requirements described above should be performed at development of any N-version software system.

Meeting the suggested requirements at development of N-version software enables achieving the following results:

- elimination of mutual influence of the program modules on each other's operation;
- localization of the program bugs and quick feedback;
- convenient deployment and enlargement the N-version program system;
- enlarging the circle of possible developers for the program modules of the N-version program system.

Conclusion

N-version software is used in a number of critical areas and successfully ensures the required tolerance to program and, sometimes, to hardware failures. Three are a lot of works dedicated to the software reliability issue and particularly to NVP. The authors distinguish more new aspects that, if taken into consideration, can help to avoid faults in the software systems.

This article is devoted to a topic which has not drawn much attention of researchers and practical professionals before – the requirements to design and development of N-version software program modules. The article formulates the main requirements to versions (program modules) of N-version software which ensure a high-level reliability and fault-tolerance due to elimination of possible influence of separate versions on each other. A special attention has been paid to their interaction, which should not have any impact on operation of the other components.

The proposed requirements are especially important at the stage of design and development of the modules versions of N-version software for machines with shared memory.

Creation of the environment for the program module N-version run and a set of program modules became a result of practical realization of the N-version approach to development of a fault-tolerant software with component design. Design and realization of the program modules took place with the consideration of the stated requirements. Testing of the N-version software runtime environment showed expediency of a component architecture application and high efficiency of NVP as a method of fault-tolerant software development.

References

1. Carzaniga, A., Gorla, A., Pezzè, M.: Handling software faults with redundancy. In: Architecting Dependable Systems VI. LNCS, vol. 5835, pp. 148–171. Springer Berlin, Heidelberg (2009)

2. Aidemark, J., Vinter, J., Folkesson, P., Karlsson, J.: Experimental evaluation of time-redundant execution for a brake-by-wire application. In: 2002 International Conference on Dependable Systems and Networks, pp. 210–215. IEEE, Washington DC (2002)

3. Avizienis, A., Chen, L.: On the Implementation of N-Version Programming for Software Fault Tolerance During Execution, In: IEEE COMPSAC'77, Chicago, pp. 149–155 (1977)

4. Avizienis, A.: The Methodology of N-Version Programming in M. Liu (ed.), Software Fault Tolerance. Wiley, Chichester (1995)

5. Chernigovskiy, A.S., Tsarev, R.Y., Knyazkov, A.N.: Hu's algorithm application for task scheduling in N-version software for satellite communications control systems. In: 2015 International Siberian Conference on Control and Communications, pp. 1–4, IEEE (2015)

6. Sommerville, I.: Software engineering 9 edition. In: Addison Wesley Pearson, New York (2011)

7. Westphal, L.C.: Handbook of Control Systems Engineering. Springer Science & Business Media, New York (2012)

8. Hosek, P., Cadar, C.: VARAN the unbelievable: An efficient N-version execution framework. In: International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 339–353. ACM, New York (2015)

9. Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., Zainlinger, R.: Distributed fault-tolerant real-time systems: The Mars approach. IEEE Micro. 9, 25–40 (1989)

10. Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: TreadMarks: shared memory computing on networks of workstations. Computer. 29, 18–28, (1996)

11. Gruzenkin, D.V., Tsarev, R.Yu., Pupkov, A.N.: Technique of Selecting Multiversion Software System Structure with Minimum Simultaneous Module Version Usage. Adv. Intell. Syst. Comput. 465, 375–386 (2016)

12. Baudry, B., Monperrus, M.: The multiple facets of software diversity: Recent developments in year 2000 and beyond. ACM Comput. Surv. 48, 16 (2015)

13. Creeger, M.: Multicore CPUs for the Masses. ACM Queue. 3, 64-ff (2005)

14. Chen, Y., Cheng, D.K.W., Lee, Y.S.: A hot-swap solution for paralleled power modules by using current-sharing interface circuits. IEEE T. Power Electr. 21, 1564–1571 (2006)

15. Buxton, J.N., Randell, B.: Software Engineering Techniques Report. In: Conference sponsored by the NATO Science Committee, pp. 27–31. Scientific Affairs Division, Brussels (1969)

16. Hastings, R., Joyce, B.: Purify: Fast detection of memory leaks and access errors. In: Proc. of the Winter 1992 USENIX Conference, pp. 125-138. USENIX Association, Berkeley (1991)

17. Koren, I., Krishna, C. M.: Fault-tolerant systems. Morgan Kaufmann, San Francisco (2007)

18. Burns, A., Wellings, A.: Real-Time Systems and Programming Languages, Addison-Wesley Longman, Harlow (1997)

19. Adams, J.C.: Fortran 95 handbook: complete ISO/ANSI reference. MIT press, Cambridge, Massachusetts (1997)

20. Norvig, P.: Paradigms of artificial intelligence programming: case studies in Common LISP. Morgan Kaufmann, San Francisco (1992)

21. Box, D.: Essential COM. Addison-Wesley, Menlo Park (1998)

22. Natella, R., Cotroneo, D., Duraes, J.A., Madeira, H.S.: On fault representativeness of software fault injection. IEEE T. Software Eng. 39, 80–96 (2013)

23. Winter, S., Tretter, M., Sattler, B., Suri, N.: SimFI: From single to simultaneous software fault injections. In: International Conference on Dependable Systems and Networks, pp. 1–12. IEEE Computer Society, Washington (2013)

24. Tsarev, R.Yu., Chernigovskiy, A.S., Shtarik, E.N., Shtarik, A.V., Durmuş, M.S., Üstoglu, I.: Modular Integrated Probabilistic Model of Software Reliability Estimation. Informatica 40, 125–132 (2016)