Shubinsky I.B.

METHODS OF SOFTWARE FUNCTIONAL DEPENDABILITY ASSURANCE

The paper considers the basic aspects of the construction of reliable software. It discusses methods and ways of error prevention based on protective and multiversion programming, tools of passive and active detection of errors, principles and methods of error correction based on dynamic redundancy and restart. Some emphasis is made on solving the tasks of construction of error-tolerant programs with the help of digression methods, techniques of error isolation and construction of redundant algorithms not critical to various types of information process violations.

Keywords: *a program, an error, software dependability, error prevention, error detection, error correction, error-tolerant software, antibugging, multiversion programming, restart, dynamic redundancy, algorithmic redundancy, isolation of errors.*

1. Introduction

The functional dependability of information systems substantially depends on software dependability [1]. All principles and methods of software functional dependability assurance in accordance with their purpose can be divided into four groups: ***error prevention, error detection, error correction and error tolerance assurance*** (fig. 1). Principles and methods allowing to minimize or completely exclude errors belong to the first group. Methods of the second group concentrate on the functions of software itself that help to reveal errors. The third group includes the functions of software intended for correction of errors or their consequences. Error tolerance (the fourth group) is a degree of software system ability to continue functioning in case of errors.

2. Error prevention

This group includes principles and methods, whose purpose is to prevent from occurrence of errors in the ready-made software. It should be obvious that the error prevention is an optimal way to achieve software dependability. The best way to provide dependability is in the first place to prevent from occurrence of errors. To that end, developers widely apply methods of the so-called ***antibugging (safe programming)*** designed to reduce the probability of errors in programs, and also ***multiversion programming***.

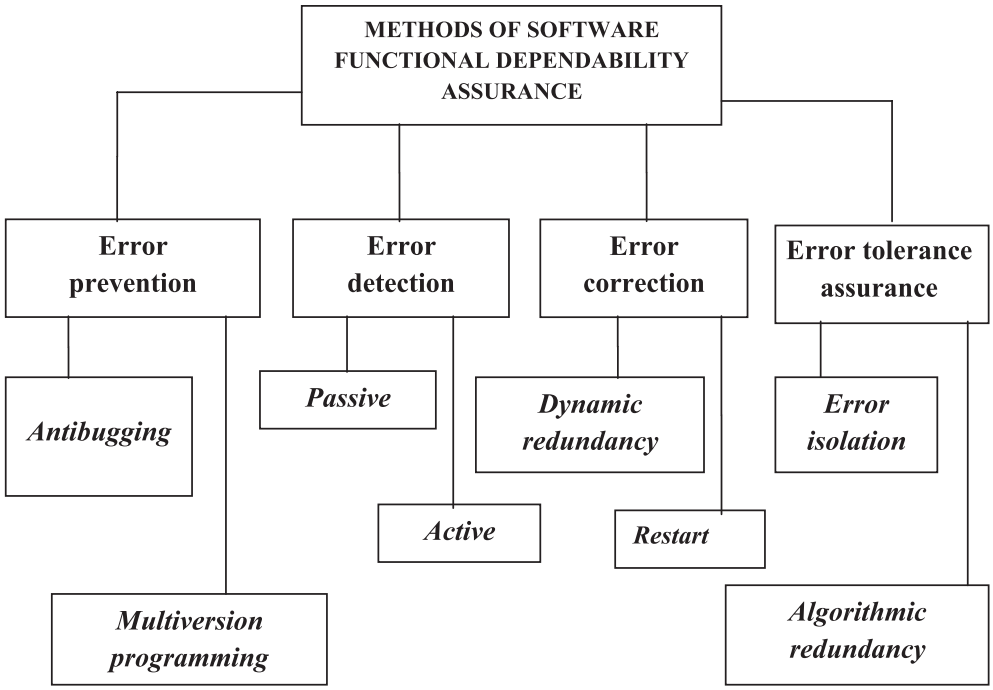

Fig. 1. The basic aspects of construction of the architecture of reliable software

Antibugging is understood as a restriction of misuse of program objects. In other words, the requirement is put forward to designing and programming in such a way that would insure the expected use of the program in strict conformity with specifications, as well as to make its misuse impossible. For example, when designing a system in which many modules interact, we can demand that some interactions between them should be allowed only in certain situations. Thus, module A can request module B always, in certain situations or never in spite of the fact that modules A and B are in such relations that the request is possible.

Antibugging is a technology of potential errors' prevention by checking a set of allowable conditions in each module. During programming many techniques can be used to check up anomalies in control or in data. Some of antibugging methods are listed below:

- boundary values of variables are analyzed;
- the sizes, type and range of parameters of data input procedure are checked;
- parameters "access only to read" and "read-write" should be divided, and access to them should be checked up;
- symbolic constants should not be accessible for writing;
- input variables and intermediate (auxiliary) variables with physical value should be checked up for authenticity;
- influence of output variables should be checked up, preferably by direct observation of changes of a system state connected to them, etc.

The elementary method of antibugging consists in use of special traps for errors intended for errors such as misuse of modules. The developer of the program does not concern about what the user will do after receiving the message on misuse of the module, but he is obliged to design the module so that user errors will not cause irreversible changes in the module. Thus, the user caught on misuse of the module makes adjusting actions and again activates the module, not leaving any traces of erroneous calls. In other words, it implies such programming when it is not so easy or impossible to use the software product outside the scope of its specification.

Multiversion programming (N-version programming). The purpose of this method is to find out and mask residual errors of software design during execution of programs, to prevent critically hazardous

system failures, and to continue operation with high dependability. In N-version programming the given specification of the program is realized differently for N time. The same values of input data are given to N-versions, and then the results made by N-versions are compared. If the result is considered credible, it is transferred to computer as output data. N-versions can be carried out in parallel on separate computers or all versions can alternatively be executed on one computer. Output results are exposed to internal voting. Various strategies of voting can be used on N-versions depending on requirements of application. For critical information systems, the full consent of all N-versions is necessary. For other information systems, the strategy of voting by a simple majority can be used. For cases where there is no collective consent, probabilistic approaches can be used to maximize the chance of choosing a correct value, for example, by taking an average value, temporarily freezing output data before the consent returns, etc.

For dual programming (if two versions of the program are developed), in case of detection of divergence in results, it is necessary to define by additional criteria what result is correct and to reject any other result. For N-version programming, the correct result is defined according to a majority criterion, i.e. we select the result which is observed in the majority of program versions.

The considered ways of redundancy demand twice or N as much time for calculations and increase in labor volume of programmers by as much time. In this reference, of interest is modified dual programming where a sufficiently exact, but complex basic program is accompanied by a less exact, but simple backup program. If for identical initial data the results of programs' execution differ by the value which is higher than the allowable error, the assumption is taken that the basic program has failed as a less dependable one, and the result of backup program execution is accepted as the correct result.

However, it is never possible to guarantee the absence of errors. Other three groups of methods rely on the assumption that errors will nevertheless take place.

3. Error detection

If we assume that any mistakes will nevertheless take place in software, the best strategy (after error prevention) is to include tools for error detection in software itself.

The majority of methods are designed whenever possible for immediate detection of failures. The immediate detection has two advantages: it is possible to minimize error influence and the subsequent difficulties for the person who should extract information about it, to find and to correct it.

Measures related to error detection can be divided in two subgroups: *passive* attempts to find out error symptoms in the process of "usual" execution of software and *active* attempts of program system to periodically survey its own condition in search for an error flag.

Passive detection

Measures related to error detection can be accepted at several structural levels of a program system. Here we shall consider a level of subsystems, or components, i.e. we shall be interested in measures on error symptoms' detection undertaken at transition from one symptom to another, and also symptoms inside a component. Certainly, all this applies also to individual modules inside a component.

Developing these measures, we shall rely on the following.

1. *Mutual suspicion.* Each of the components should assume that all others contain errors. When it obtains any data from another component or from a source outside of the system, it should assume that the data can be wrong, and try to find errors in them.

2. *Immediate detection.* It is necessary to detect errors as soon as possible. It limits damage inflicted by them as well as considerably simplifies the problem of debugging.

3. *Redundancy.* All means of error detection are based on some form of redundancy (explicit or implicit).

When measures on error detection are developed, it is necessary to accept the coordinated strategy for the whole system. The actions undertaken after error detection in software should be uniform for all system components. It brings an attention to the question as to what actions should be taken, when an error is not detected. The best decision is immediately to end the program execution or (in case of an operating system) to put a central processor in wait state. From the point of view of granting the most favorable conditions in error diagnostics for a person debugging the program, for example, a system programmer, the immediate end offers the best strategy. Certainly, in many systems a similar strategy appears unreasonable (for example, it can appear that it is impossible to suspend system operation). In this case a method of *error registration* is used. The description of error symptoms and "snapshot" of system state is saved in an external file, and then the system can continue operation. This file will be investigated by maintenance staff later.

Always, when it is possible, it is better to suspend program execution than to register errors (or to provide system operation in any of these modes as an additional opportunity). The difference between these methods will be illustrated by ways of revealing causes for a grinding sound sometimes arising in your car. If the car mechanic is on a back seat, he can survey a car's state at that moment when a grinding sound arises. If you choose a method of error registration, the problem of diagnostics becomes more complex.

Active detection of errors

Not all errors can be revealed by passive detection methods as these methods detect an error only when its symptoms are exposed to appropriate checkup. It is possible to make even additional checkup if to design special software for active search for error attributes in a system. Such means refer to as *tools for active detection of errors*.

Tools for active detection of errors are usually combined in *the diagnostic monitor*: parallel process, which periodically analyzes a system's state with the purpose to detect an error. The large-scale program systems managing resources frequently contain errors that lead to resource loss for long time. For example, running of an operating system's memory hands over blocks of memory "in rent" to programs of users and other parts of the operating system. An error in these very "other parts" of the system can sometimes lead to erroneous operation of the memory control block as the memory engaged in return of memory handed over earlier in rent, which causes slow degeneration of the system.

The diagnostic monitor can be realized as a periodically executed task (for example, it is planned for each hour) or as a task with a low priority, which is planned for execution when the system switches over to wait state. As before, concrete checks carried out by the monitor depend on system specificity, but some ideas will be understandable from examples. The monitor can examine the basic memory to detect memory blocks, not dedicated to any of tasks carried out and not included in a system's list of free memory. It can check also unusual situations: for example, a process was not planned for performance during some reasonable interval of time. The monitor can carry out search for "mislaid" messages inside the system or operations of input-output which remain uncompleted for unusually long time. The monitor can also search for sites of memory on a disk which are not marked as allocated and are not included in the list of free memory, and also a various sort of strangeness in data files.

Sometimes it is desirable that under emergency the monitor should execute trouble-shooting testing of a system. It can activate certain system functions, comparing their results with beforehand determined ones and checking them as far as the execution time is reasonable. The monitor can also periodically submit "empty" or "easy" tasks to the system to make sure that the system functions at least in some primitive mode.

4. Error correction

The following step consists in methods of error correction; when the error is detected, either it or its consequences should be corrected by the software. Error correction by the system is a fruitful method of dependable hardware. Some devices are capable to detect faulty components and switch over to use identical backup components. Similar methods are inapplicable to software due to deep internal distinctions between equipment failures and errors in programs. If some program module contains an error, identical "backup" modules will also contain the same error.

Another approach to error correction is connected to attempts to recover the destructions caused by errors, for example, distortions of records in databases or control tables of a system. The advantages of methods used for struggle against distortions are limited, as it is supposed that the developer beforehand will foresee some possible types of distortions and will provide software functions for their elimination. It is similar to paradox as, if to foreknow, what errors will arise, it would be possible to take additional measures for their prevention. If methods of failure consequence elimination cannot be generalized to work with many types of distortions, it will be the best way to direct efforts and means for error prevention. Instead of equipping an operating system under development with its tools of detection and restoration of a chain of distorted tables or control blocks, when developing an operating system, it is better to design a system in such a way that only one module will have access to this chain, and then persistently try to receive evidence of this module correctness.

5. Error tolerance

The basic assumption of ***software error-tolerant*** programming consists in the fact that it does not matter how good the program has been designed and realized, it will still contain some residual errors. And if that's the case, then modules of the program which can fail, should have "a buffer stock". With this purpose the module is designed in the form of *recovery blocks*. Each recovery block contains a pass-through test and one or several alternatives of realization. The basic alternative is initiated by activation of the recovery block and when its execution comes to the end, the checkup of a pass-through test value is carried out. If it makes "true", it is considered that the execution of a recovery block is successfully completed. If the test makes "false", then another alternative is initiated and after that the definition of a pass-through test value is carried out again, etc. and so on till the successful execution of a recovery block. If any alternative has not passed the pass-through test, the recovery block is considered as erroneous and the execution of another alternative of the activated module begins. Other technique is applied as well: various segments of the program are written, frequently independently, each of which is intended for execution of one function. The program is made up of these segments. The first segment named primary is executed first. It is followed by a pass-through test of the calculation result of the first segment. If the test has passed successfully, then the result is accepted and transferred to the subsequent parts of the system. If the test has been unsuccessful, any side effects of the first segment are reset, and the second segment named as the first alternative is executed. It is also followed by a pass-through test whose results are considered as in the first case. If it is necessary, other alternative techniques can be realized.

The introduced method of parrying software residual errors by designing the module in the form of a recovery block sometimes requires the unjustified efforts connected to designing of several recovery blocks, each of which contains the pass-through test and one or several alternatives of realization. For the purpose of practical support of program system functioning at presence of errors, the group of methods which is divided into four subgroups has been developed: *dynamic redundancy, back-off methods, methods*

of error isolation and construction of algorithms immune (or not critical) to various sorts of information process violations (use of algorithmic redundancy).

1. The ***dynamic redundancy*** concept roots in designing of hardware. One of the approaches to dynamic redundancy is the *majority backup* (a method of voting). The data are processed independently by several identical devices, and obtained results are compared. If the majority of devices have produced an identical result, then this result is considered as correct. And again, as a result of special nature of errors in software, the error presented in a copy of the program module will be present also in all its other copies; therefore, in this case the idea of voting probably is unacceptable. The approach sometimes offered for the solution of this problem consists in having some not identical copies of the module. It means that all copies execute the same function, but either they realize various algorithms or are developed by different authors. This approach is unpromising for the following reasons. It is frequently difficult to receive essentially different versions of the module which executes identical functions. Besides, there is a necessity in additional software for the execution of these module versions in parallel or serially and comparisons of obtained results. This additional software improves the level of system complexity that certainly contradicts to the basic idea of error prevention – first of all to aspire to minimize software complexity.

The second approach to dynamic redundancy consists in execution of these backup copies only when the results received with the help of the basic copy are recognized wrong. If it occurs, the system automatically activates a backup copy. If its results are also wrong, another backup copy is activated etc.

2. The second subgroup of error tolerance assurance methods refers to as ***back-off methods*** or methods of reduced service. These methods are acceptable usually only when the most important thing for software system is to end an operation correctly. For example, if an error turns out in the system controlling technological processes and as a result this system fails, then a special fragment of the program designed to secure the system and to provide accident-free end of all processes controlled by the system can be loaded and executed. Similar tools are frequently necessary in operating systems. If an operating system detects that it is just about to fail, it can load the emergency fragment responsible for notification of users at terminals about a forthcoming failure and for saving of all system critical data.

3. The third subgroup consists of error isolation methods. Their basic idea is to prevent error consequences from going outside the boundaries of the smallest possible system software part, so that if an error arises then not the whole system will fail; some system functions or some of its users are shut down. For example, in many operational systems errors of some individual users are isolated, so failure influences only some subset of users, and the system as a whole continues to function. In telephone switching systems for recovery after an error in order not to risk the whole system failure, telephone connection is simply cut. Other methods of error isolation are related to protection of each of system programs from errors of other programs. An error in the applied program which is executed under operating system control, should affect only this program. It should not affect an operating system or other programs functioning in this system.

Isolation of programs in an information system is the key factor ensuring that errors in a single user program will not lead to errors in programs of other users or to a complete system failure. Key rules of error isolation in programs consist in the following:

1. The applied program should not have any opportunity to refer directly to another applied program or data in other programs and to change them.

2. The applied program should not have any opportunity to refer directly to programs or a operating system and to change them. Communication between two programs (or the program and the operating system) can be allowed only under the condition of usage of precisely defined interfaces and only in case when both programs agree to this communication.

3. Applied programs and their data should be protected from the operating system to a point that errors in the operating system could not lead to casual change of applied programs or their data.

4. The operating system should protect all applied programs and data from their casual change by system operators or maintenance staff.

5. Applied programs should not have any opportunity either to stop a system, or to compel it to change another applied program or its data.

6. When the applied program addresses the operating system, the acceptability of all parameters should be checked, the applied program should not have any opportunity to change these parameters between the moments of check and their real use by the operating system.

7. None of system data directly accessible to applied programs should influence the functioning of an operating system. An error in the applied program owing to which the contents of this memory can accidentally be changed, leads eventually to system failure.

8. Applied programs should not have any opportunity to bypass an operating system to directly use hardware resources controlled by it. Applied programs should not call directly the components of an operating system intended for use only by its subsystems.

9. Components of an operating system should be isolated from each other so that an error in one of them should not lead to changes of other components or their data.

10. If an operating system detects an error in itself, it should try to limit influence of this error by one applied program and as a last resort to stop execution of only this program.

11. An operating system should give applied programs an opportunity to correct errors detected in them on demand, instead of unconditionally stopping the execution of applied programs.

Realization of many of these principles influences the architecture of system underlying hardware. Though we use the term "operating system" when defining many of them, they are applicable to any program (whether it is an operating system, teleprocessing monitor or file management subsystem) which is engaged with service of other programs.

4. The fourth subgroup is ***introduction of algorithmic redundancy***.

6. The conclusion

The paper considers effective methods and ways for construction of the architecture of functionally dependable software. The European experience in development of programs for safety related systems [2] is used, and recommendations of standard [3] are also applied. Ensuring software dependability is not limited by development stages of their specification and qualitative architecture. To solve the key task of reliable information system construction, it is necessary to design the dependable software, to implement, to integrate it with hardware, to provide dependability of software during certification, operation and maintenance. These stages of construction of dependable program are the subject of further discussion.

References

1. **Shubinsky I.B.** Functional dependability of information systems (Methods of analysis). M.: Dependability Journal Ltd., 2012, 296p.

2. BS EN 50128:2011. The software for control and protection systems on railways.

3. GOST R/IEC 61508-3-2012. Functional safety of electrical/electronic/programmable electronic safety-related systems. Requirements for software.