České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering

Ing. Martin Kropík, CSc.

Software pro systémy důležité z hlediska jaderné bezpečnosti

Software for Systems Important to Nuclear Safety

Summary

The lecture deals with software for systems important to nuclear safety. The computer based systems are utilized in such cases at present. The quality and reliability of computer based systems is given not only by the quality of the hardware, but also of the software. The lecture provides recommended methodology of software development for systems important to nuclear safety. To prepare this lecture, the author utilized his experience of proper standards, lectures at the university, and activities during the upgrade of the VR-1 training reactor control and safety system.

The three fundamental principles of high quality software assurance – fault avoidance, fault detection and fault tolerance are mentioned and explained in the first part of the lecture. The next part deals with the software life cycle. The software life cycle consists of the following phases - requirements, design, coding, verification, hardware/software integration, validation, operation and maintenance. The requirements represent a basis for software development. The requirements have to be correct, complete, consistent and unambiguous. During the software design, the software requirements are analyzed, the software specification is prepared, and algorithms and data structures are designed. Coding converts the software design into a real program in a programming language. Coding methods and techniques are shortly discussed in the lecture. Verification checks that all activities and demands of each software life cycle phase were successfully accomplished. Typical verification aims, methods and tools are addressed in the lecture. The next software life cycle phase is hardware/software integration, where the hardware and software modules are assembled, the software is loaded into the hardware and the integrated system is tested. The purpose of validation is to check that the integrated system satisfies the given requirements. The validating system is checked through the simulation of input signals to represent normal operation and accident conditions. The operation phase then includes all activities connected with the commissioning and the operation of the system. During the maintenance phase, software modifications are carried out.

The following chapter describes the configuration management, which is important part of the software development and production. It defines the organization of the project, identification of items, secure storage of developing software, tools and documents, change control procedures and revision control.

The final chapter studies the defense against software common cause failure and diversity. It discusses different methods to decrease danger of common cause failure, further benefits and drawbacks of diversity as a protection against common cause failures.

Souhrn

Přednáška se zabývá programovým vybavením software pro systémy důležité z hlediska jaderné bezpečnosti. Počítačové systémy jsou v současnosti stále častěji používány v takových systémech. Jakost a spolehlivost počítačových systémů je určována nejen jakostí a spolehlivostí technického vybavení (hardware), ale i programového vybavení (software). Přednáška seznamuje s doporučenou metodikou pro vývoj software systémů důležitých z hlediska jaderné bezpečnosti. K její přípravě využil autor zkušenosti získané studiem příslušných norem, z přednášek na vysoké škole a z činností při inovaci bezpečnostního a řídicího systému školního jaderného reaktoru VR-1.

První část přednášky zmiňuje a vysvětluje tři základní principy pro zajištění vysoké jakosti software – vyloučení chyb, detekce chyb a tolerance k chybám. Další část se zabývá životním cyklem software. Životní cyklus se skládá z následujících fází – požadavků, návrhu, kódování, verifikace, integrace hardware/software , validace, provozu a údržby. Požadavky představují základ pro vývoj software. Požadavky musí být správné, úplné, konzistentní a jednoznačné. Během fáze návrhu software jsou analyzovány požadavky, je připravena specifikace software, jsou navrženy algoritmy a datové struktury. Fáze kódování převádí návrh software do skutečného programu v programovacím jazyce. Metody a techniky kódování jsou v přednášce stručně diskutovány. Verifikace pak ověřuje, zda všechny činnosti v každé fázi životního cyklu byly úspěšně vykonány a příslušné požadavky splněny. Typické cíle, metody a nástroje verifikace jsou uvedeny v přednášce. Další fází životního cyklu software je integrace hardware/software, během níž jsou sestaveny hardwarové a software moduly, software je nahrán do hardware a integrovaný systém je vyzkoušen. Účelem validace je ověřit, že integrovaný systém vyhovuje stanoveným požadavkům. Vylisovaný systém je testován simulacemi vstupních signálů za podmínek normálního provozu i havarijních situací. Fáze provozu pak zahrnuje všechny aktivity v souvislosti s uvedením do provozu a provozováním systému. Během fáze údržby jsou prováděny změny v software.

Další kapitola se zabývá správou konfigurace, která je důležitou součástí vývoje a produkce software. Definuje organizaci projektu, identifikace položek, bezpečné zálohování software, nástrojů a dokumentů, postupy pro správu změn a revizí.

Závěrečná kapitola se zabývá ochranou proti chybám ze společné příčiny a diverzitou. Poskytuje rozbor různých metod ke snížení nebezpečí vzniku chyb ze společné příčiny, dále pak výhody a nevýhody diverzity jako ochrany proti chybám ze společné příčiny.

Keywords: software quality and reliability, software life cycle, verification, validation, configuration management, common cause failure, diversity

Klíčová slova: jakost a spolehlivost software, životní cyklus software, verifikace, validace, správa konfigurace, chyba ze společné příčiny, diverzita

Contents

1. Introduction

Electronic parts of nuclear installation systems that are important to safety have recently been designed as hardwired electronics. However, computer based safety systems are already used at present. On the one hand, performance, flexibility, testability and economic factors are a great advantage of such systems. On the other hand, the quality and reliability of these systems is given not only by the quality and reliability of the hardware (HW), but also by the quality and reliability of the software (SW). Thus, software is playing an ever increasing role in the design, manufacture and operation of nuclear installations, and the potential of error and poor quality is real.

Because of problems with assessing the qualitative and quantitative reliability of the software, there is a lack of confidence on the side of licensing authorities. The aim of software engineering is therefore to optimize the quality of software products and to find generally acceptable methods to determine the quantitative reliability of the software.

2. Basic principles

Safety has to have priority over availability in the sense that if the two goals conflict then availability should be sacrificed in order to maintain safety. However, both safety and availability are closely correlated to the quality and reliability of the software. Three basic principles to assure high quality and reliability of software are further mentioned.

The first principle in this respect is fault avoidance through good software engineering and quality assurance throughout the complete lifecycle of the software. The second principle is fault detection through a thorough validation and verification activity. A third principle, which should also be considered, is fault tolerance, i.e. the target system should be designed so that a failure will not jeopardize safety. This could be made so that the failure has no effect, e.g. through redundancy, or by bringing the system into a safe state, or into a state of reduced risk, in case of a failure. Despite the effort to increase the reliability of safety critical software, the software itself and the environment should be continuously monitored to intercept hazards before they convert into accidents. This principle is based on the recognition of the fact that no presently available technology can guarantee absolute safety.

The software should be designed using well-known and widely accepted design techniques. These techniques could either be supported by computer-based solutions, e.g. CASE (Computer Aided Software Engineering) tools supporting different design techniques, or they could be just theoretical techniques with no computer support. Within the same project, the number of techniques should be limited to just a few. Preferably only one method should be used throughout the design. What could be convenient is to use two different methods for top level design and detailed design respectively. The top level design methods could be for example of graphical nature, while more detailed level methods could utilize the language sensitive method.

In order to meet the overall system reliability requirements, the computer system software shall continuously supervise both itself and the hardware. No single failure shall be able to block directly or indirectly any function dedicated to prevent the release of radioactivity. Those parts of the memory that contain code or invariable data shall be monitored to prevent any changes. The software for systems important to nuclear safety shall be designed so that essential functions of the whole system are testable during the operation of the nuclear installations. The system self-checking shall not adversely affect the intended system functions.

Before starting the software development, some necessary documents should be prepared - at least “Quality Assurance Plan“, “Verification and Validation Plan“ and “Configuration Management Plan”. The first document specifies the methodology of the software development with respect to high quality and reliability requirements for the safety critical software. The second one defines the activities of verification and validation during the whole software life cycle. The third one deals with the identification of product items, the control and implementation of changes, secure storage and revision control. All the activities described in these plans are approached later in this lecture.

3. Software life cycle

The software life cycle is a period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life cycle consists of the following phases, which are to some extent self-contained, but will depend on other phases as well. These phases are informally recognizable by specific activities pertinent to them.

The list of the software life phases is as follows:

- requirements,
- design,
- coding,
- verification (V&V),
- integration HW/SW,
- validation (FAT, SAT),
- operation,
- maintenance.

Concerning the software life cycle, it is necessary to mention that in the earlier documents the term *verification* was used for checking the activities during each software life cycle phase and *validation* was used for testing the integrated system usually by simulated input signals. In later documents, the earlier verification is called *verification and validation (V&V)*, and for integrated system testing the abbreviations *FAT (factory acceptance tests)* and *SAT (site acceptance tests)* are used. In the following text, the individual phases of the software life cycle will be discussed in more detail.

3.1. Requirements

Software requirements are a basis for further development. Software requirements are derived from the computer system specification. The computer system specification is a description of the combined hardware/software system and states the objectives and functions assigned to the computer system. The requirements present an expansion of the functions assigned to the computer system to be implemented. Functions dedicated to prevent the release of radioactivity in accident conditions shall be explicitly described. The software requirements describe the product, not the project. They should require what must be done, not how to do it.

The software requirements should have the following attributes; they should be:

- correct - the requirements shall correctly express the intention with the target system, it should also be based on a correct interpretation of the nuclear installation and environment,
- unambiguous - there should be only one possible semantic interpretation of each requirement; if the terminology of real world items has multiple meaning, a glossary with the precise use of terms is necessary; the use of a formal language can be used to reduce ambiguity,
- complete - no requirement or need of the customer is overlooked; for every possible set of inputs, a system response should be clearly defined; all referenced terms must be defined, all referenced tables, figures, sections, etc. must be present in the document,
- consistent - there should be no internal contradiction between individual requirements, and terms must not have different meanings at different places within the document,
- verifiable - it should be possible to verify the requirements versus the computer system specification, this attribute is facilitated by using formal specification methods,
- modifiable - the requirements should ease later modifications, redundant information may be necessary, even if it complicates the modification; a computerized support system would facilitate modifiability,
- traceable - the origin of each of the requirements should be clear.

Utilization of formal methods supports the fulfillment of the above mentioned points.

3.2. Design

In the design phase, the software requirements are analyzed, the software specification is prepared, and algorithms and data structures of the software are designed and tested. The software should be divided into modules. The design should be made in top-bottom manner, from more complex to specific units. The software needs to fulfill demands for modularity and testability. Minimal global variables, proper input arguments and local variables should be used.

Program modules should perform exactly one function, have exactly one exit and have a reasonable number of branches. Consistent naming conventions for constants, variables, functions and program modules should also be introduced. Structures of the data are an important aspect from a reliability point of view. Tricky data structures should be avoided, and also items like dynamic data allocation should be avoided if possible, since the verification of the correctness of such structures is very difficult.

When communication between programs or computers is required it is very important to use standardized communication facilities, home made solutions should be avoided if possible. Safety critical applications should, if possible, be isolated from external communication. It is important to consider both the possibility of unauthorized access into the safety critical applications and the possibility of performance degradation due to increased network traffic.

3.3. Coding

Coding converts the software design into a real program in a programming language. The coding process should be carried out in bottom-top manner, from specific low level modules to the more complex ones.

Suitable programming languages with reliable compilers should be used. High level programming languages are strongly recommended. The languages should be completely and unambiguously defined, otherwise the use of the languages shall be restricted to completely and unambiguously defined features. Within a single project, the number of programming languages used should be minimized, and low level languages (e.g. assembly language) should be used only in specific, typically time critical parts of the software project.

Relatively strict guidelines should be closely followed when producing code. One should have in mind that the code should be easy to read for an outsider, which means that the code must be fairly simple.

The following should be avoided:

- tricky programming,
- fancy programming,
- code compression, etc.

Encouragement should be given to provide:

- easily understandable code,

- use of strict naming conventions for variables,
- modularity,
- initiation of all variables, etc.

The code should be properly commented, making sure that one can easily understand it. A heading should be provided for all programs, functions or subroutines, containing, at least, the following information:

- module name,
- short description,
- reference,
- version number,
- language,
- file name, load file,
- target computer and operating system,
- compiler used,
- modification history.

The software modules should have satisfactory comments, their style and conventions are uniform throughout the whole project.

3.4. Verification

Each phase of the software life cycle should be finished by a verification (V&V) process to prove that all activities and requirements for this phase were successfully fulfilled. No method, even not formal proofs, can guarantee correctness with 100 % confidence. Verification can actually only show the presence of faults, it cannot prove the complete absence of faults. However, the more one search for faults, the higher the probability to find all residual faults.

A detailed "Verification and Validation Plan" of the V&V activities should be established at an early stage in software development. This plan should describe all the planned V&V activities with respect to attributes as:

- methods,
- purpose,
- importance for the safety assessment,
- criteria of fulfillment,
- documentation of activity results.

The V&V plan should be observed for all V&V activities, and result in a V&V document that forms one basis for the safety assessment.

To verify the correctness of the target system, it is necessary to have a basis to check it against. The essential basis for the verification is the specification (requirements). A correct, complete and unambiguous specification is therefore a necessary requirement for the verification. Notice that not only the functional requirement, but also the safety requirements must be fulfilled by the target system. A verification of the specification should also be made. A check on completeness and consistency should be possible if the specification requirements are properly written.

There are several complementary methods for V&V. They can be divided into some main classes:

- static analysis,
- testing,
- formal verification,
- real-time aspects.

Static analysis is defined as a process of evaluating a computer program without executing it. Static analysis can be performed at various stages in the development process and can therefore be applied during inspections and walkthroughs. It can, however, also be applied to the final program system. One objective of the static analysis is to compare the final program with the specification. Static analysis can be used to check that the specified coding is observed. The static analysis can also reveal faults in the program, as e.g. use of undefined variables, defined variables never used, eternal loops, etc. Another use of static analysis is to compute software metrics. Software metrics is a set of measurements that can be performed on the computer programs in the target system. Typical measurements are: size of program, number of subroutines, maximum size of subroutines, number of conditions, number of loops, etc. Such measurements can be very useful to assess the complexity of the program, the vulnerability to errors, and the amount of work needed to perform the safety assessment. In many cases, there exist computerized tools that perform such software measurements. Reversed engineering and symbolic execution may be used to show correspondence between the specification and the implemented code of the target program. If computerized tools are used for this activity, it would be advantageous, but not necessary, that these tools are made independent of the production of the target system.

Testing is the execution of the program with selected testing data to demonstrate that it performs its task correctly. Ideally, the test data should be selected so that all potentially residual faults should be revealed. The testing is mandatory for any program. The amount of testing depends upon the criticality of the program. The optimal test strategy is the one which maximizes the probability to reveal all possible residual program faults. Various test strategies are: random testing, systematic testing based on specification, systematic testing based on program structure, testing data reflecting a real process, testing with special emphasis on safety critical functions.

An aspect of safety critical systems that must be particularly focused is the real time aspect. One must verify that the system can fulfill its tasks within specified time limits. Both static analysis and statistical testing can be used in this respect. The use of interrupts should be minimized. However, if interrupts are used, one must show that their usage and masking during time and data critical operations are checked for correct operation in-service. One should also check that all inadvertent activation of unused interrupts are handled in a fail safe way. A particular problem occurs if the target system consists of a set of distributed processors operating concurrently. There are certain properties such a system should have, and some potential problems one should look out for: correct communication between different modules, proper synchronization and time sequencing, potential deadlocks checking, correct use of shared resources and content of one module is protected against the destruction by others.

Any faults, bugs and errors found during this phase should be carefully documented in the record of verification activities.

3.5. Integration HW/SW

The process of hardware/software integration is the combining of verified hardware and software modules into a system that is capable to perform specified functions. This process consists of the following parts:

- assembling hardware modules according to the system design drawing,
- assembling software module by a linkage processor,
- load the software into the hardware,
- verifying by testing that the hardware/software interface requirements have been satisfied and that the software is capable of operating in this particular hardware environment.

Before the integration of HW/SW, the “System Integration Plan” shall be prepared that specifies the standards and procedures to be followed in the HW/SW integration and document those provisions of the overall quality assurance plan that are applicable to the system integration. The “System Integration Plan” shall establish a library of SW and HW modules as a means of system configuration control. This library shall provide revision control for all HW and SW modules to be used in the system.

The system verification assures that the verified HW and SW modules have been properly integrated into the system and that the HW and SW are compatible and performs as required. The system shall be as complete as is practical for this testing. In the process of verification, the certain aspects of individual hardware and software modules may be better tested at the integrated system while, on the other hand, it is not feasible to test all functional requirements of individual modules at the system level. The result of the integrated system verification shall be documented in the “HW/SW Integration Report”.

3.6. Validation

The purpose of validation (FAT, SAT) is to check that the integrated hardware and software satisfies the defined requirements. The validating system shall be exercised through static and dynamic simulation of input signal present during the normal operation, anticipated operational occurrences and accident conditions requiring the system action. Each reactor safety function should be confirmed by representative test of each trip or protection parameter, both singly and in combination. The tests should cover the signal range, the voting or other logic and logic combinations. The tests should ensure that accuracy and response times are confirmed, and that correct action is initiated for any equipment failure or failure combination. At the end of the validation process, a computer system validation report shall document the activities and the results of the validation process.

3.7. Operation and system training

Operation means all activities regarding the commissioning and the operation of the system. It concerns the computer system and the staff (operators). A test program shall be provided to verify the integrity of the installed digital computer based safety critical system with respect to response, calibration, functional operation and interaction with other systems. In the end, a commissioning, i.e. a test report presenting test results and conformity to acceptance criteria, shall be established

The software system being brought to approval will, after the final implementation at the site, be handled either by technical staff, operated by operators, or probably in combination of both of them. From a safety point of view, the quality of the software system is obviously extremely important, but also users' competence can have an impact on overall safety.

Therefore, a training program as part of the project should be developed and arranged for the personnel who will use the system, to ensure that they are able to handle the system in the best way. Proper system operation and user manuals must of course be available as an integral part of the software system.

Depending on the nature of the system, in some cases simulator training should be recommended. This is probably one of the best ways how to perform system training altogether.

3.8. Maintenance

The reason for maintenance may be:

- anomaly report,
- change of functional requirements after delivery,
- technological evolution,
- change in operating conditions.

During the maintenance, a modification request on the software can occur. However, the modification demand may also be requested during the development phase because of functional requirement changes or anomaly report as a result of tests. To deal with software changes, the following items shall be examined: technical feasibility, impact upon hardware (e.g. memory extension), impact upon software, impact upon performance (e.g. speed, accuracy) and set of documents which is necessary to review. If the change of the software is accepted, a scope of changes to be introduced shall be investigated. For a change in the software functional specification, the whole software development process for any part of the system impacted by the change shall be re-examined. A change during the development phase or maintenance after delivery shall be reviewed in terms of its potential impact upon corresponding lower levels. After the implementation of the modification, the whole verification and validation process shall be performed again according to the software modification analysis. A software modification report shall sum up all the actions made for modification purposes.

4. Configuration management

The "Configuration Management Plan" should document the method to be used for identifying software product items, controlling and implementing changes, and recording and reporting the change implementation status. The plan should be applied to the entire software life cycle and is especially important in the environments where software could have an impact on safety measures.

The "Configuration Management Plan" should be used both for software and the associated documentation. By applying configuration management both to software and documentation, one is assured that both are being updated in parallel and coincide way at any time.

There are several key items in a "Configuration Management Plan", and some of them will be treated below.

4.1. Organization

It is very important to define an organization around the configuration management, stating persons and their dedicated responsibilities. The persons being responsible for the configuration handling have an independent status to those producing software and documentation in the project, and should typically report to the project manager.

4.2. Identification of items

The decision about which items have to be placed under configuration control must be decided at an early stage within the project. The items of the configuration management should be documents (e.g. system requirements and design documents, verification and testing documents, system and user manuals, maintenance documents) and software (e.g. all software produced according to the specification, special test software, standard software/system software , pre-existing software).

4.3. Secure storage

Documents and software being placed under configuration control must be stored in a safe place. With a safe place one means for instance: backups are taken, diverse backups are stored at different sites, and security measures are implemented to prevent accidental or intentional modification of software or documents by unauthorized personnel, and correct version labeling is assured.

4.4. Change control procedures

Once software or documents are being placed under configuration control, changes to these items have to be made according to strict procedures. The reasons for changes may vary. Typical examples are detection of errors, functional improvements, additional functions, and implementation improvements. The request for changes may also come during a project and after the software delivery, i.e. in the maintenance period. All change control requests must be registered by the configuration management organization and a decision must be made to which change control level the request should be brought. It is important to note that certain changes in a software product could lead to the consequences that should be evaluated and submitted for approval.

4.5. Revision control

A software product may consist of very many integrated parts. It is dependent upon heterogeneous products, whose typical examples are: operating system, version a.a; compiler, version b.b; library, version c.c, etc. It is quite obvious that there is a need for the configuration management organization to keep a track, not only of the specifically produced software, but also of the environmental software products. Complete revision control procedures stating how to cope with newly released software products must be established and strictly observed.

5. Defense against software common cause failure and diversity

The rationale for the defense against software common cause failure (CCF) is that any software fault will remain in the system or channel concerned until detected and corrected, and will cause failure if an adverse signal trajectory challenges it. If two or more systems or channels contain the fault, and they are exposed to adverse signal trajectories within a sensitive time period, both (or all) systems or channels will fail. The potential of software CCF should therefore be considered during the design.

All CCF occurring during the operation of an I&C system are the consequence of human errors of some kind made during the different phases of the life cycle of the system. Human errors made before the software design start lead to faults in requirements and potential system failures against which software engineering alone cannot provide a defense. Due to such errors, CCF are defined here as I&C system CCF. Human errors made during the software engineering process lead to software faults and potential system failures which are defined here as software CCF.

A basic defense against software CCF is the production of software to meet correct requirements with as few faults as practicable, and the demonstration that this has been achieved.

Statements of requirements and software specifications shall be regarded as of special importance in the control of CCF. Independent verification of requirements and software specifications and the use of formal methods provide the defense against faults caused by human factor during formulation of requirements and software specification processes. The formulation of requirements shall be reviewed by a different technical group than the originators for consistency, accuracy, completeness and freedom from ambiguity.

If the analysis shows that two systems include common modules of software, which can be subjected to the same signal trajectories, those modules should perform correctly for those trajectories. The program start, cycle, operating system, services and communication services shall be independent of signal trajectories.

Abnormal hardware conditions, hardware failures, abnormal nuclear installation conditions and events which might cause software CCF should be reviewed and the requirements and design arranged so that simultaneous failure of redundant channels or functional paths with common software is prevented.

The procedures and tools used during the design, development and coding of the software for configuration control and software built should be administratively controlled and managed, taking into consideration the possibility of deliberate or accidental corruption. Changes should be formally controlled only after documents have reached a given level of approval.

Conditions of software sabotage, virus or software bomb planting able to cause software CCF should be prevented by suitable methods (isolation of the system from communication access, use of write protected memory for software, administrative control and access control, etc.).

The following methods may be used to provide defense against the effects of failures or to reduce the likelihood of failures occurring simultaneously:

- design which makes potential software failures behave safely or limits their effects,

- defenses for revealed failures of memory violation, arithmetic exception, overwriting, etc.,
- design of channels or systems so that a coincident failure of two channels or systems is very unlikely due to demonstrated differences in the trajectories affecting the systems identified above,
- design of channels or systems using asynchronous operation; this may be used to show defense against the same processors in different channels being subjected to identical trajectories at the same time,
- monitoring of software tasks and cycle to detect unrevealed inhibition of systems caused by a software CCF, which stops operating tasks; processes such as deadman timers with independent alarms should be considered.

Diversity should be utilized to improve the defense of software against CCF. Diverse software features include functional diversity, i.e. different functions used to achieve the same safety goal, and software diversity. Diversity at the system level can include utilization of different basic technology, such as computers versus hardwired design or different types of computers, hardware modules and major design concepts, or different classes of computer technique such as PLCs, microcomputers, or minicomputers. Differences should be implemented in design an implementation methods. Different programming languages, compilation systems, libraries, tools, programming techniques, system and application software and data structures can be utilized to enhance the diversity.

Diversity should be also applied on aspects of management approach – two designs following deliberate dissimilar development methods, separation of design teams, restriction of communication between teams, formal communication for resolution of ambiguities in requirements, use of different logic definition processes, differences in documentation methods and use of different staff.

On the one hand, utilization of the diversity will improve the protection against CCF. On the other hand, the diversity has also some drawbacks. Examples of these drawbacks are greater complexity, increase risk of a spurious actuation, more complex specification and design, maintenance and modification problems.

6. Conclusion

This lecture contains recommended methodology for the development and design of high quality and reliability software utilized in systems important to nuclear safety. The basic principles for the software development, the software life cycle and requirements on its individual phases were defined and discussed. The configuration management and defense against common cause failures were approached.

A lot of the above described methodology has been used at the Department of Nuclear Reactors, FNSPE CTU in Prague during the safety and control system upgrade of the VR-1 training reactor.

References

[1] Neufelder, A. M.: Ensuring Software Reliability, Marcel Dekker, Inc., New York, 1993

[2] Dahll, G., Kvalem, J.: Guidelines for Reviewing Software in Safety Related Systems, SKI Report 94:9, 1994

[3] Software for Computers in the Safety Systems of Nuclear Power Plants, IEC-880, Geneva, 1986

[4] Draft IEC880-1: Software for Computers in the Safety Systems of Nuclear Power Plants, as a first supplement to IEC-880, Geneva, 1997

[5] Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations, IEEE P-7-4.3.2, 1994

[6] Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems, NUREG/CR-6463, 1996

[7] Guidelines for the Use of the C Language in Vehicle Based Software, MISRA Association, 1998

Curriculum Vitae

Ing. Martin Kropík, CSc.

Born:	August 22, 1963 in Roudnice nad Labem
Affiliation:	Czech Technical University (CTU) in Prague Faculty of Nuclear Sciences and Physical Engineering (FNSPE) Department of Nuclear Reactors
Postal address:	V Holešovičkách 2, CZ 180 00 Prague 8, Czech Republic
E-mail address:	kropik@troja.fjfi.cvut.cz

Education:

1986	M.S. (Ing.) at the Department of Physical Electronics, FNSPE CTU in Prague
1993	Ph.D. (Csc.) at the Departments of Physical Electronics and Nuclear Reactors, FNSPE CTU in Prague

Professional career:

1987-1991	postgraduate student, Department of Physical Electronics, FNSPE CTU in Prague
1991-1993	researcher, Department of Nuclear Reactors, FNSPE CTU in Prague
1994-1996	research scientist, Department of Nuclear Reactors, FNSPE CTU in Prague
since 1996	assistant professor, Department of Nuclear Reactors, FNSPE CTU in Prague

Scholarships and study stays:

- Institute for Physics and Technology, Braunschweig, Germany, 1985, 1989, 1990, 1994, 1995, 1996
- Queen Mary and Imperial College, London, Great Britain 1994
- Atomic Institute, Vienna, Austria 1995, 1998
- Ruhr University, Bochum, Germany 1999, 2002
- Halden Reactor Project, Halden, Norway, 1997, 2002, 2004
- Brookhaven National Laboratory, Upton, USA, 2004

Research areas:

- control and safety systems of nuclear installations
- quality and reliability of computer systems and software
- digital and microcomputer technology
- programmable logical devices

- data acquisition and evaluation,
- nuclear reactor diagnostics and operation

Teaching experience:

- lectures Control and Safety Systems of Nuclear Installations, Fundamentals of Electronics, Programmable Logical Devices and Computer Control of Experiments at the Department of Nuclear Reactors, FNSPE CTU in Prague
- advisor of diploma theses (7)
- advisor of Ph.D. students (4)
- training at VR-1 reactor for Czech and foreign students
- lectures and training in IAEA and ENEN courses

Grant projects (solver or co-solver):

- CTU grants (3)
- Czech-Austrian Aktion project
- University promotion grants - FRVŠ (6)
- IAEA grants (2)

Publications:

- over 100 journal, conference papers and research reports