

Improving the System/Software Engineering Interface For Complex System Development

Stephanie M. White
College of Information & Computer Science
C. W. Post Campus, Long Island University
Email: stephanie.white@liu.edu

Abstract

At the 2004 Engineering of Computer Based Systems (ECBS) Technical Committee meeting, the ECBS Executive Committee agreed that a guideline on Integrated System and Software Engineering would be beneficial to engineers working at the interface, and they agreed to work on such a guideline. This paper is written in the hope that it will serve as a basis for a discussion group, initiating such a guideline. This paper seeks to improve the integrated system/software engineering process during the phases when system and software developers work most closely together. These phases are system problem definition, software requirements analysis and specification, system solution analysis, and process planning. Lessons learned during twenty years, while working with system and software engineers to define requirements and solutions on aerospace projects are summarized. During this time, problems were noted and their cause determined. As a result, advice is provided.

1. Introduction

There is a long history of poor communication between system and software engineers in performing problem definition and requirements analysis, and in defining a system solution. In 1995, Singh wrote "Software engineering is developing on its own and does not fully participate in hardware-software trade-off analysis and does not fully contribute to the system at its full potential" [1]. Oliver states "The two engineering cultures, the concepts and the best practices have developed independently for four decades" [2].

System and software engineering efforts are still not integrated even though integration would greatly increase the likelihood of producing high quality products. Standards bodies recognize the importance of integrating system and software engineering processes.

The IEEE is adopting the ISO/IEC 15288 System Engineering Standard. The adoption of ISO/IEC 15288 is part of an ongoing joint effort by the newly renamed IEEE Software and Systems Engineering Standards Committee (S2ESC) and Standards Subcommittee 7 of ISO/IEC Joint Technical Committee 1 (ISO/IEC JTC1/SC7) to harmonize their systems and software engineering standards. The S2ESC is also discussing integration of IEEE Standard 830, Recommended Practice for Software Requirements Specifications and IEEE Standard 1233, IEEE Guide for Developing System Requirements Specifications. The Integrated Process and Product Development vision in the Capability Maturity Model[®] Integration (CMMISM) also seeks to integrate disciplines [3]. It contains a common process model for system engineering and software engineering with discipline amplification for each discipline, but does not describe how the disciplines interact.

Thayer states that "Software Systems Engineering begins after the system requirements have been partitioned into hardware and software sub-systems" [4]. Many software engineers would say that this is too late. Computer-based system engineers with a deep understanding of software, hardware, network, and human-machine interface issues should work with system and domain engineers during the partitioning task, even at the system/sub-system allocation level. Computer Based Systems (CBS) engineers have knowledge needed by the system team to develop quality architectures and assess system feasibility. They should be supported by a team of hardware and software engineers who develop benchmarks, prototypes, and simulations, and analyze models and risks.

Integrated systems and software systems engineering involves five high level functions: system problem definition including software requirements analysis, system solution analysis including software design, process planning, process control, and product evaluation [4].

This paper concentrates on the first three of these functions, which address the project “front-end” as these are the functions that have the greatest impact on cost and schedule. Each of the following sections addresses a different function and provides advice that should improve integrated system and software development for complex systems. Section 2 recommends activities that system and software engineers should perform together during the problem analysis and definition phase. Section 3 addresses solution analysis and identifies system decisions that, if made incorrectly, have an adverse effect on software quality and development cost and schedule. Section 4 recommends activities that should be performed during process planning to improve the system/software engineering interface. Section 5 identifies capabilities needed in methods that support integrated system and software engineering. Section 6 discusses method evaluation for a specific project. Section 7 provides advice on tool evaluation. These recommendations are based on the author’s system and software engineering experience with various military aircraft and space projects. These include the F-14 fighter, A-6 bomber, E-2 airborne early warning aircraft, EA-6B electronic counter-measures plane, X-29 forward swept wing aircraft, and demonstration programs for a robot designed to service the space station, and a satellite system for detecting intercontinental ballistic missiles in their boost phase.

2. The interface during system problem definition and software requirements analysis

The following paragraphs specify activities in which system and software developers should work together to improve problem definition.

System and software engineers should work together to identify off-nominal behavior.

Software normally handles off-nominal behavior such as hardware failure, improper data communication, and incorrect data entry. More software is written to handle exceptional behavior than to handle nominal requirements.

System engineers perform Hazard Analysis and Failure Modes and Effects Analysis (FMEA) to identify where hardware failures might occur. Together with software engineers, they also use these methods to discover potential communication failures and situations where human error may occur. When FMEA isn't thorough during system requirements definition, widespread software change during test and integration may be required. As Boehm has indicated, requirements changes during integration can be significant, more than eighty times what a change costs during the requirements

phase [5]. Another way of finding off-nominal conditions is to write scenarios and identify where in the scenario an exception may occur. Exception scenarios may occur infrequently but are important to address if the system is to be reliable. Scenarios can also be used for verification after requirements are defined, to determine whether requirements are consistent and adequate.

Budgets and schedules should include resources for prototyping user interfaces and building simulations where needed.

Systems engineers have begun to reap the benefits of working with software engineers. Users normally do not understand what automation can do for them and have great difficulty expressing their own requirements. When prototypes and simulations are built, users understand what is possible and they provide needed feedback. However automation takes time and resources. System and software engineers have to make sure that managers understand that prototypes and simulations should be built during problem definition and requirements analysis, and provide appropriate budget, time, and resources in the project plan.

Software and system engineers should verify that software behavior models are consistent with the system behavior model, and thus match system requirements.

Subsystem developers must understand how the behavior of their subsystem, whether software, hardware, or hybrid, fits into system scenarios. A system model is necessary that defines system behavior and subsystem interaction. At this level, it is advantageous to model the software using the same methods and tools that system engineers use, to check consistency between models.

Pay attention to both semantic issues and tool interface issues during system/software engineering interaction and propagate changes in both directions.

The use of inconsistent semantics and tools by different disciplines leads to communication and traceability problems. Requirements documents are normally written in natural language (e.g. English) which does not provide a precise definition of what is needed to downstream developers. When systems engineers build conceptual models, they are normally based on function flow. System engineers may use a tool such as Core [6], which is derived from the Ballistic Missile Defense Advanced Technology Center’s Distributed Computing Design System (DCDS) [7]. Software engineers frequently use object-oriented concepts in their essential models. It is nearly impossible to determine if these models are consistent without a mapping from one to the other, which does not exist today.

Manage dependability as a system measure.

Dependability attributes are emergent properties of the system. That is, they are a measure of the system as a whole and not just a sum of measures for the component parts. System engineers should manage hardware and software dependability as a system measure. However, there are recognized differences between hardware and software when addressing dependability. For example, hardware wears out and is repaired or replaced while software is more flexible and can be changed, but too much change can make the software less reliable. Such differences affect the overall evaluation of the integrated system. Dependability requirements should address both product and process. The product part is concerned with specified quantitative measures related to functions, behavior, and data. Measures for such dependability requirements may differ by scenario, mode, or operational profile. The process part specifies required practices that developers shall perform during the system life cycle, to ensure that dependability requirements will be met and can be verified. More information on managing dependability is provided in [8].

Trade capability against resource utilization and performance, to ensure that resources are adequate.

It is especially important for engineers to make these trades in embedded systems, where additional resources will be unavailable once weight and footprint have been allocated to computer hardware and other physical assets. To investigate nonlinearities caused by scale-up of capability and data, prototypes and simulations are needed; prototypes to benchmark performance of algorithms running on to-be-built processors, and simulations to investigate how scaling up capabilities (e.g. 10,000 targets instead of 10 targets) affect requirements for processors and storage.

System and software engineers should build models using multiple views.

A number of views (function flow, dataflow, object-oriented) provide more insight than a single view, but views must be consistent. Some years ago, a group working at Northrop Grumman on the synthesis of system and software models defined system requirements using Requirements Driven Design (RDD), a systems requirements tool based on function flow. When they translated the information to enter it into Cadre, a software requirements tool based on dataflow, they found gaps in the original systems model. When they refined the systems model to include the new information, they found additional gaps, this time in the software model, and this iterative correction continued showing that additional insight was gained from looking at each view.

3. Solution analysis - system decisions that can adversely affect software development

The following paragraphs identify system decisions that, if made improperly, can have an adverse impact on software development.

Allocation

Poor allocation of data and functionality to subsystems, processors, software configuration items, and databases can affect system quality and cost. Excessive bandwidth on communication links, delays in data access, and problems with end-to-end timing are some of the problems that result. Following the principles of low coupling between system components and high functional cohesion within a component normally results in proper allocation.

Improper functional allocation to computers versus humans is also a concern as computers are not good at performing cognitive tasks. Computer-aiding for human decision analysis is normally the best solution.

Lack of component commonality

Lack of concern for commonality is costly. Reuse saves development, test, documentation and maintenance costs. The United States Department of Defense policy mandates reuse in the three services. The government and industry have developed domain-specific software architectures and assets, such as Hughes assets for Air Defense, ARPA assets for guidance and control, Boeing assets for trainers, and Army/Unisys assets for Intelligence and Electronic Warfare [9].

On the other hand, there have been catastrophes associated with reuse when the context or environment changed, for example the Therac-25 accidents and the Ariane 5 Flight 501 failure [10, 11]. It is important to fully understand changes in context between current use and previous successful use, and conduct studies to determine the potential effect of differences.

Point designs

Too often, engineers create a single solution without examining alternate designs. The lack of a full investigation results in poor understanding of design variables and options, and their effect on the system. Engineers should examine potential options, such as allocation of functionality to hardware, software, and humans; distributed versus centralized control or peer to peer; fixed versus scaleable computing strategies; communication options (protocols, network type, mobility); security features; and so on. Alternatives for a large number of system aspects should be considered.

Lack of prototypes and executable models

Problems arise when engineers do not prototype or simulate high risk elements. Executable models allow engineers to experiment with designs prior to actual

build, so that they can find problems and fix them when it is relatively inexpensive to do so.

Engineers should define risks, and identify those that can be mitigated by building prototypes and simulations. They should then identify costs and benefits, and build executable models where the benefits outweigh the costs.

Poor interface management

Because work done by one contractor affects another at the interface, definitive interface specification is important. Sometimes requirements and design are not well defined at the time that subsystem interfaces should be specified. This can be due to an overly optimistic schedule, inadequate knowledge of the domain, or may be unavoidable due to the uniqueness of the project. When it is not possible to fully define an interface, it is important to track and report changes to the interface as soon as problems are identified as changes become more expensive later in the life cycle. Encapsulation of interface software prevents widespread change to software and documentation in many cases, but not in the case where data is added to the software interface.

Insufficient resource reserves

Computer resources (memory and throughput) are normally reserved for engineering error and for growth. New requirements, requested by the customer or discovered during design, are a prime driver of increased resource needs. It may not be feasible to provide required system functionality in the current system, or in future versions when reserves are inadequate. Determining resource needs in advance is difficult when systems are unique, or when functionality does not scale linearly. Target tracking is an example of the latter case. As the number of targets grows, computer resource needs for associating sensor hits with the correct track expand non-linearly. Benchmarks and simulations help in calculating resource needs, but do not always provide sufficient insight.

In embedded systems, power, space, and weight limitations constrain the ability to add computer resources to existing system architectures, so it is especially important to have extra reserves. It is easier to add reserves than in former years. Today's processors have greater throughput and larger memories, are smaller, weigh less, and require less energy and heat dissipation.

Late definition of support software

Software engineers build more support software than target system software for embedded systems. Software engineers create stubs and drivers, but also create more sophisticated project specific software. They build emulations of subsystems that will arrive too late, and emulations of external systems that are too large and expensive to install in the laboratory, or are simply not available. Even though support software must be

delivered to the project before target software is delivered to the customer, definition of support software requirements is frequently informal, late, and sometimes even haphazard. When support software is not developed properly, schedules slip or worse, software is not properly tested before delivery. Software managers should work with system and project managers to define a project schedule that includes requirements specification for support software, and provides the appropriate personnel and time to do it properly.

Late definition of data access requirements for testing

System engineers require data access during test and integration. They should define data access needs during software requirements and design phases. Data that is normally inaccessible during system execution must be made accessible. If system engineers do not write requirements to access this data, the software must be rewritten during the test phase when it may be expensive to do so and may cause schedule delays. It is desirable that test software remain in delivered code so that it can be used for regression testing and for maintenance and so that the software is not corrupted by removing test related code.

Scheduling hardware deliveries too late

Processing hardware should be delivered at the time coding is scheduled to begin. If the hardware is delivered late in the project life cycle, software developers do not have sufficient time to refine and test software on the target machine. This is usually an issue in an embedded system or with advanced technology, where processors and software are both under development and the schedule is tight. Working with management to define feasible schedules is highly recommended.

4. Process planning for early phases

The following paragraphs recommend processes and procedures that improve the system/software engineering interface.

Provide dialogue structure for the system / software engineering interface.

System engineers frequently do not understand what information should be provided to software engineers. System and software engineers should define the correct level of information needed to determine feasibility (e.g. to determine throughput, bandwidth, storage), and the correct level needed for design (e.g. to specify interfaces, scenarios, data relationships, system behavior, required timing, security, reliability and so on). Engineers should also define lists of the various trade-offs that could arise and ways of resolving them. See the paragraphs in section 3 on "point designs" and in section 5 labeled "provide support for design tradeoff analysis" for examples of trade-offs that engineers should consider.

Document an integrated system and software process for analysis and specification.

Specification templates and standards do not explain the work that has to be done to produce the required models and specifications. A process description is necessary.

Work together to manage risk.

System engineers normally have a better understanding than software engineers of the application domain and environment, and teaming, vendor and organization related risks. Software engineers normally have a better understanding of computer processing and software development risks. Both types of expertise are needed to understand how the application environment could adversely affect processing and how events in external and internal organizations could adversely affect software development. Both types of expertise are needed to find ways to mitigate defined risks.

Define standards.

Insufficient standards for naming conventions, units of measurement, modeling methods, and so on can cause significant interface issues and integration problems. Standards should be defined during the planning phase prior to the phase where they are needed, and should be distributed to all members of the development team, including subcontractors.

5. Planning method use

The following paragraphs identify characteristics that good methods should possess. More on these characteristics as they apply to existing methods appears in [12-14]. This paper examines each method attribute as it applies to concurrent system and software engineering. Some of these attributes are not available in practical methods today.

Support collection and relation of system and software requirements and design information in integrated models.

Integrated models support analysis to determine whether system and software requirements are consistent. Integrated models also help in understanding requirements dependencies. Methods should provide a process for constructing the software model from the system model. This is not the norm today unless system and software engineers use the same modeling method, which engineers in these disciplines say doesn't support the needs of both. This is because system engineers start with a "clean piece of paper" and need methods that support early problem definition when changes are frequent and widespread. Software engineers start when the problem has already been defined and there is a system design. They need methods that support

evolution of software requirements (essential) models to design models and implementation.

Integrated methods have to support both needs. The Systems Engineering Domain Special Interest Group (DSIG) of the Object Management Group (OMG) is working on system engineering extensions to UML, the Unified Modeling Language (<http://syseng.omg.org>). However, a requirements language is not sufficient by itself unless it also includes modeling methods, principles, patterns, model analysis and validation, model execution, and test generation.

A combined system/software requirements database is necessary for traceability. Without traceability, system engineers will know when the customer changes a system requirement, but software engineers may not know.

Provide good comprehension of system and software aspects and their inter-relationships.

Complex systems should be modeled using stepwise refinement to support better comprehension and easier construction. It is not possible to consider a complex system at a single level. Using modes (a characteristic way of behaving) to help simplify a problem description was recognized in the early 60(s) [15].

Models should relate scenarios, modes, events, entities, information, and functions between system and software levels, and at each level of refinement.

Function flow models, which system engineers frequently use, identify system functions (capabilities), their inputs and outputs, and their ordering. Object-oriented (OO) models, frequently used by software engineers, identify object functions called methods and method signatures. In OO-models, the order of system operations is defined in scenarios which are linear or branching, and system operations are carried out by sequences of methods defined in object interaction diagrams. While it seems obvious that there is a mapping between entities and functions in the function flow model and their correspondents in the object-oriented model, it is not easy to comprehend the inter-relationships, even in a relatively simple system.

Engineering methods and tools should isolate integrated system and software information which is aspect specific, so that it is readily accessible for review and change (separation of concerns).

Base models on mathematical theory and define a mapping between system and software modeling constructs.

Methods that support development of complex systems should be based on formal concepts. Such concepts support model analysis for consistency, completeness, reachability, determinism, and other desirable model attributes, and they support model execution. Formal concepts include input to output

mapping, algebra (data abstraction), process abstraction, function composition / decomposition, sequential machines, concurrent machines (cooperating sequential processes), extended abstract machines, petri-nets, predicate logic, and temporal logic. Engineers need a mapping between modeling constructs that system engineers use and constructs that software engineers use to analyze integrated models.

Provide the ability to execute system and software requirements (essential) models and design models.

Executable models provide engineers with a better understanding of system behavior. Such models can be used to learn about the evolving system and make design trades. Also, engineers know that the model is correct when it executes as the system should. The benefits of design independent (essential) models are that the solution is not overly constrained and there is more flexibility to arrive at an optimal design. Some system design decisions have to be made prior to creating a software requirements model. Execution of essential models normally requires that engineers add design information, which is labor intensive, especially when analyzing a number of potential designs. Methods and tools should provide a number of built-in model parts to support various types of design decisions.

Use a temporal order approach which does not impose any unnecessary order.

System and software models are complex. When we combine the two, the resulting model is larger and thus more difficult to manage. Methods are needed that reduce complexity. Partial order approaches reduce complexity as the specific order need not be defined. One statement covers many sequences. In addition, partial order methods are better than linear and branching approaches in discriminating between concurrency and non-determinism [16]. When automated, partial order approaches are executable.

Provide support for design tradeoff analysis.

Many system design trades affect software, so software as well as system engineers should be involved in making these trades. Examples of such trades include distributed versus centralized control; processor type, for example signal processor versus vector processor; software allocation, for example whether a satellite system software element should be located in space or in the ground station. Engineers need guidelines for making these trades. Engineers create hardware and software prototypes, benchmark processor performance, and build simulations to make these trades. These engineers need a capability for tracking and relating alternatives, studies, decisions, and the effect of these decisions, so that they can revisit the studies and decisions in the face of new and changing requirements, improved knowledge, and better understanding.

Provide a capability for defining non-functional requirements.

System engineers should manage dependability as a system measure, and pass down software and hardware dependability requirements to ensure the required level of system dependability. Engineers need methods and tools that support specification and allocation of all non-functional requirements, and measures that can be used during development to make sure these requirements are being met.

Support extensive model verification including consistency of system and software models.

Formal model views for system and software can be checked for consistent logic, and semi-formal views can be checked according to method rules. For example, all software functionality should relate to system functionality, and all external software interfaces should appear in the system model. Extended state machines (a formal view) can be analyzed to make sure that software, integrated with other system aspects, is deterministic [12].

Engineers define assertions about the system and demonstrate that they are true, to show that the system has certain properties. With an integrated modeling capability, engineers could prove that the integrated system (subsystems, software and hardware) has certain properties, for example that an event eventually happens or that an invariant holds.

Support definition of acceptance tests.

Acceptance tests are the basis for system acceptance by the customer. These tests are normally black-box: inputs are supplied and outputs are examined. Design independent models can be useful in defining acceptance tests. Pertinent model aspects are event-action causal relationships in which events and actions are received from or affect the external environment, interface definitions, and quality measures such as timing, accuracy and reliability.

Using an integrated system/software model, engineers may be able to save financial, human, and laboratory resources by writing tests that satisfy both software and system testing requirements.

Identify items that are to-be-determined (TBD) in specifications.

It is important to define omissions in specifications so these are not overlooked during further definition. When there are TBD(s) in a system specification, software engineers know that they will get this information at a later time. These engineers have to determine how the lack of information affects architecture, design trades, and risk, and need method and tool support to do this.

Support reuse by capturing the context (environment) in which the reusable component and the system operate.

The Therac 25 deaths and the Ariane 5 flight 501 rocket failure have shown us that reuse does not always produce a positive result [10, 11]. System and software engineers have to work together to determine the differences in context between the current use and previous successful use. In the case of the Therac 25 X-ray therapy machine, the original system had a hardware mechanism that prevented incorrect positioning from occurring. In the case of the Ariane, the original rocket had a lower trajectory. Methods and tools should support context capture.

Manage system and software measurement.

The following categories of system and software measures should be collected and analyzed: (1) Schedule and Progress (2) Resources and Cost (3) Product Size and Stability (4) Product Quality (5) Process Performance (6) Technology Effectiveness and (7) Customer Satisfaction [17]. Determining process and product measurements requires information from system, software, hardware, quality, communication, and other domain engineers. Modeling provides information for a number of measurement categories. Models support estimation of product size. Progress can be seen by the model size, state, and quality. Model quality is also a good predictor of eventual product quality. Process performance can be gauged by model completeness, for example are all data descriptions documented. Technology effectiveness in front end phases is measured in part by effectiveness of prototypes, models, and simulations. Customer satisfaction in front end phases relates in part to their satisfaction with the system model.

6. Evaluating methods for a specific project

The previous section provides insight into desirable method characteristics. This section provides advice, when evaluating methods for a specific project.

Select high risk or complex portions of project requirements for examination and specification.

When evaluating a method for use on a specific project, engineers should apply the method to complex aspects of their system to see if the method scales to their application. You can scale down a method that works on large problems, but you can't scale up a method that worked for you on small problems.

Evaluate method views with the highest risk first.

This approach is consistent with Barry Boehm's spiral process for software development where the first prototype resolves the highest risk [18]. Views that are high risk may eliminate the method from consideration. A view may be high risk because it is complex to develop and costs too much in contrast to the benefit. A

view may also be high risk because of a concern that it may not be appropriate for the domain.

Allow domain engineers to express requirements using their own terminology and methods.

On method evaluation projects, to overcome communication problems, we allowed domain engineers to use their own terminology and views. As time progressed, we found that both the requirements engineer and the domain expert acquired a measure of understanding of the other's area of expertise, thus providing checks and balance on the work performed.

7. Evaluating tools

This section provides advice on evaluating tools based on a number of good and bad experiences.

Use a real (not toy) specification for tool evaluation.

To evaluate a requirements specification tool for use in a specific domain, it is important to use a sufficiently complex application in that domain. Use of a simple specification may not reveal problems. Our tool evaluation revealed the lack of logical and algebraic operators, which would have prevented us from expressing compound conditions and mathematical calculations.

Have domain and system engineers available to interpret the application used to assess the tool.

Domain engineers are needed to answer questions concerning the application. It is normally helpful to communicate using terminology that is familiar to the domain expert as opposed to terminology familiar to software engineers so that there is less chance of miscommunication. Inconsistencies in terminology may occur, and software engineers should be alert for this.

8. Summary and conclusions

Based on discussions, research, and experience, this paper provides suggestions to improve concurrent engineering involving system and software engineers. Problem definition, requirements analysis, solution synthesis, software design, and planning phases are addressed. The section on methods discusses the need to integrate system and software models. The two engineering disciplines have different needs (concept definition and demonstration versus implementation), so it remains to be seen whether a common modeling method can be devised.

9. Acknowledgements

Several ideas for this paper were drawn from a working group chaired by the author on the State of Practice in Computer-Based Systems Engineering [19].

A number of ideas were the result of a “birds of a feather” session held at the Software Productivity Consortium and attended by Alan Kocka, who worked for Northrop Grumman Corporation at the time, Sanford (Sandi) Friedenthal, Lockheed Martin Corporation, and the author. I would like to thank all members of both groups.

10. References

- [1] R. Singh, “Harmonization of Software Engineering and System Engineering Standards”, *Proceedings of the 2nd IEEE Software Engineering Standards Symposium (ISESS'95)*, IEEE Computer Society Press, 1995, pp. 262-267.
- [2] D. W. Oliver, “Systems Engineering & Software Engineering, Contrasts and Synergism”, *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering (CASE '95)*, IEEE Computer Society Press, Aug. 1995, pp. 186-193.
- [3] Software Engineering Institute (SEI), Carnegie Mellon University, *Capability Maturity Model[®] Integration (CMMISM), Version 1.1, CMMISM for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing*, Pittsburgh, PA, March 2002.
- [4] R. H. Thayer, “Systems Engineering, A Tutorial”, *IEEE Computer*, IEEE Computer Society Press, April, 2002, pp. 68–73.
- [5] B. W. Boehm, *Software Engineering Economics*, Prentice Hall, NJ, 1981.
- [6] D. M. Buede, *The Engineering Design of Systems, Models and Methods*, John Wiley & Sons, NY, 2000.
- [7] M. Alford, “SREM at the Age of Eight; the Distributed Computing Design System”, *IEEE Computer*, Vol. 18, No. 4, IEEE Computer Society Press, Apr. 1985, pp. 36 - 46.
- [8] B. Melhart and S. White, “Issues in Defining, Analyzing, Refining, and Specifying System Dependability Requirements”, *Proceedings of the 7th IEEE International Conference on the Engineering of Computer Based Systems*, IEEE Computer Society Press, April 2000.
- [9] S. White and M. Edwards, “Domain Engineering: Challenges, Status and Trends”, *Proceedings of IEEE International Symposium on the Engineering of Computer Based Systems*, IEEE Computer Society Press, Mar. 1996.
- [10] N.G. Leveson and C.S. Turner, “An Investigation of the Therac-25 Accidents”, *IEEE Computer*, Vol. 26, No. 7, IEEE Computer Society Press, July 1993, pp. 18-41.
- [11] G. LeLann, *The ariane 5 flight 501 failure – a case study in system engineering for computing systems*, Technical Report 3079, INRIA, December 1996, <http://www.inria.fr/RRRT/RR-3079.html>.
- [12] S. White, *A Pragmatic Formal Method for Computer System Definition*, PhD Dissertation, University Microfilms, Ann Arbor, Michigan, 1987.
- [13] S. White, "A Comparative Analysis of Requirements Methods", *Proceedings of IEEE International Symposium on Requirements Engineering*, IEEE Computer Society Press, Apr. 1994.
- [14] S. White, “A Comparative Analysis of Object-Oriented and Other Methods for Modeling Computer-Based Systems”, *Proceedings of the IEEE International Conference on Engineering of Computer Based Systems*, IEEE Computer Society Press, May 2004, pp. 13 - 20.
- [15] W. R. Ashby, *An Introduction to Cybernetics*, John Wiley and Sons, New York, 1963.
- [16] A. Pnueli, “Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends”, in *Current Trends in Concurrency, Overviews and Tutorials*, ed. By J.W. deBakker, W. P. deRoever, and G. Rozenberg, Springer Verlag, Lecture Notes in Computer Science, Vol. 224, 1986, pp. 510 – 584.
- [17] J. McGarry et al., *Practical Software Measurement: Objective Information for Decision Makers*, Addison Wesley, Boston, MA, 2002.
- [18] B.W. Boehm, “A Spiral Model of Software Development and Enhancement”, *IEEE Computer*, IEEE Computer Society Press, May 1988, pp. 61-72.
- [19] S. White, M. Alford, J. Holtzman, S. Kuehl, B. McCay, D. Oliver, D. Owens, C. Tully, and A. Willey, "Systems Engineering of Computer-Based Systems", *IEEE Computer*, Vol. 26, No. 11, IEEE Computer Society Press, Nov. 1993, pp. 54 - 65.