# Precise Documentation of Critical Software

David L. Parnas Sergiy A. Vilkomir*
*Software Quality Research Laboratory (SQRL)*
*Department of Computer Science and Information Systems*
*University of Limerick, Ireland*

## Abstract

*This experience and research based paper discusses the reasons that software cannot be trusted and then explains how the use of greatly improved documentation can make software more trustworthy. It shows how tabular expressions can be used to prepare software documents that are both precise and easily used by developers, inspectors, and testers. The paper reviews a number of "tried and true" ideas and illustrates some new refinements in the methods that resulted from recent research. It is intended both to tell developers of techniques available to them and to suggest new research areas.*
***Keywords:*** *critical software, documentation, specifications, testing.*

## 1. Introduction

Telephony, cameras, car radios, and power plants are all vastly improved by the availability of programmable processors. Because the hardware is programmable, it can be mass-produced making the processors are far less expensive to produce than the custom hardware that it replaces. Programmability also means that products can be improved without changing the hardware. Often, the software includes programs that download and install a new version of itself. Moreover, in applications such as power-plant control, we can introduce much more refined behaviour, i.e. makes finer distinctions, than we could make when all such functions were "hard-wired". Software also allows us to collect and display more operating data than would be possible with older technologies.

We do not need to look far to see that people are wary of software. Not too long ago, experts in software-safety were saying that they would not fly in an aircraft that was controlled by software. Today we can read long articles from other experts who believe that software cannot be trusted for a simple job like counting votes. The reasons for this lack of trust are many and complex; we list a few of them in the next section.

## 2. Trustworthiness of software

The following factors impact trustworthiness of software and hardware systems:

- Software is usually very difficult to inspect/understand.
- Software is difficult to test.
- There is often lack of agreement on the functions required of software products.

**Difficult to inspect/understand.** Decades of experience have made it clear that we find it difficult to understand long programs. When trying to understand and verify the correctness of a long program, we must decompose it into small parts and (provisionally) associate a function with each one. We must then convince ourselves that:

- If each part implements its assigned function, the whole program will be correct,
- Each part implements its assigned function.

Frequently, we find that our provisional assumptions do not correspond exactly to what the programmer intended or what the program does. Then, after revising our initial division and function descriptions, we try again. In principle, this iterative process converges and we learn whether or not the program is correct. In practice, we usually give up before we have a complete understanding of the program. The process terminates when we run out of time or patience.

* Now with Software Quality Research Laboratory (SQRL), Department of Electrical Engineering and Computer Science, University of Tennessee, USA

1530-2059/07 $25.00 © 2007 IEEE
DOI 10.1109/HASE.2007.70

This inspection/understanding process is also unreliable because it depends on imprecise descriptions of component functions supplemented by the inspector's memory and assumptions. A program may look correct but be wrong because a small detail was not mentioned or forgotten. People do not trust devices whose design is so complex that nobody seems to understand them. Until we improve our ability to understand and inspect software, it will not be trusted.

**Difficult to test.** One of the most often quoted remarks by the late software pioneer E. W. Dijkstra is "Program testing can be used to show the presence of bugs, but never to show their absence" [4]. This remark is based on the fact that the number of possible tests for even simple programs is very large and it is usually possible to find a program that will pass every test that you have done but fail every other. Consequently, in most cases, we never know when we have tested enough. In sharp contrast to older technologies, interpolation does not work properly for the discrete functions that describe the behaviour of software. The result is that, even after extensive testing has been performed, errors are found by users. One can never be sure that all those errors have been found or when the next one will be encountered. People will not trust a product unless they are convinced that it has been tested adequately.

**Lack of agreement on the desired function.** Most people find it extremely difficult to say what behaviour they want from a computer system. Often, when asked to state requirements, the reply is something like, "I will know it when I see it". After delivery, we hear, "That's not it". People do not trust software when they keep discovering new requirements as they try to use it.

## 3. Comparison with hardware systems

Although we are fully aware that our lives will depend on the work of a large team of Engineers, most of us climb into airplanes without much concern. We know that the products are complex and that the failure of even a small part might put us in danger but we do trust the product. Again, it is easy to list some of the reasons:

The track record of mechanical and electrical systems is good.

Although the products are complex, they have a clearly visible structure that allows them to be divided in parts, each with precise specifications, and each part can be inspected and understood separately.

Testing of mechanical and electrical systems is a well-developed, mathematics and physics based profession. The properties that need to be confirmed are well defined and the methods that lead to confidence are understood.

There is little doubt about what functions are expected from the vehicle and from each of its components. These have been documented, reviewed, and used in the review of internal designs.

If we wish to increase the trustworthiness of software we must improve our ability to inspect, test, and specify software systems and their components.

## 4. Good documentation can make software more trustworthy

If we look at the methods used to engender confidence in mechanical and electrical engineering products, we will see that documentation of the function, design, and components of those products, plays an essential role. The professional Engineering documentation used in those processes is precise, abstract, and, designed for use as a reference document rather than an introduction.

Good documentation helps us to conduct more effective inspections and tests. Precise documents allow Engineers to confirm that if each component meets its stated specification, the product will be satisfactory. Combining testing and mathematical methods, we can confirm that each component does meet its specification.

In the remainder of this section, we discuss what is required of this documentation.

**Documentation must be precise.** Precise documents do not allow differing interpretations. Statements such as "*The system must shut down if the temperature is over limit for too long*" are obviously imprecise. However, although "*The system must shut down if the average temperature over the most recent 3 second period is above 89 C*" is much more precise, it still does not indicate whether we are taking about mean, median, or rms (root of the mean square) averages. Usually, precision in engineering requires the use of mathematical expressions.

**Documentation must be abstract.** We consider documentation to be abstract if it refers only to observable, information and abstracts from (does not mention) anything that a user or using-program could not observe. This is a special case of a more general definition of "an abstraction" as something that represents many distinct things. Abstract documents represent all implementations that have the properties they describe. Abstract documentation of devices with memory must be able to refer to past (as well as present) input and output values.

**Document must be designed for easy retrieval of specific facts.** Reference documents are not introductory or overview documentation; they assume general familiarity with the product and environment. It is endemic in Computer Science to mix reference documentation with introductory material. The usual collection of syntactic information and informal comments that come with software is also not adequate. Separate documentation should be available for those who are first becoming familiar with the general nature and structure of a product.

Reference documents must be designed as repositories suitable for information retrieval. Reference documentation must be an easily accessed source of trustworthy and detailed information about the program and its behaviour. Using a reference document, one should be able to quickly answer questions such as:

- What will this component do if it receives the input sequence ...?
- What are the circumstances that lead to the output of ...?
- How can I get this component to ...?

Reference documentation must be organized according to strict rules that dictate one-and-only-one place for each fact. They are not designed to be read from start to finish but to make it easy to quickly find an accurate answer to a specific question.

## 5. The content of key software documents

The secret of making any product more trustworthy is the old adage, "divide-and-conquer". We need to be able to look at a set of small components in such a way that we are confident that if all of the components are trustworthy the product as a whole can be trusted. With complex products this means that each of the components must be precisely specified and that the correctness of a component can be verified knowing only that component's specification and the specification of the components that it uses. Those specifications are the documents that we have been discussing. Clearly, the process will only work if the documents are accurate, precise, and complete. The process will be easier if the documents are abstract because this reduces the amount of information that must be considered.

Many people are surprised to learn that it is possible to define the required contents of software documentation mathematically. Each document has a distinct purpose and intended audience, but all represent a relation (set of ordered pairs). Most of those relations are functions. For example:

**System requirements documentation** must be a representation of a set of relations between time-functions that represent the visible past history of the system and the values of the system's outputs. The domain of the relation must include all possible histories of the inputs and outputs of the system (see [5]).

**Software component interface documentation** must be a representation of the relation between a sequence of event descriptions (traces) and the present output values of the component. The domain of the relation must be the set of all possible histories of the input and output values.

**Program function documentation** (for terminating programs) must be a representation of the LD-relation between starting and stopping states of the program. The domain of the relation must include all states with the property that if the program starts in that state, termination is possible. The competence set of the LD-relation includes all starting states for which termination is guaranteed [11].

**Component internal design documentation** must be a representation of an abstraction relation and the program relations of the principle programs in the component. The domain of the abstraction relation must include all reachable internal states of the component. The range of the relation is a set of traces that includes all possible histories of the component.

A document is considered an accurate description if every ordered pair that can occur is included in the relation and all pairs that are included can occur. This is discussed in more detail in [13].

## 6. Descriptions of piecewise-continuous relations

The content descriptions in Section 4 are simple but quite abstract. We need to be able to provide readable representations of these relations. If conventional mathematical notation is used the result is often something that its author would not want to read. Fig. 1 is a precise abstract description of the behaviour required of a keyboard testing program used at Dell in Limerick, Ireland. To learn more about the meaning of the relations and predicates that appear in this expression you can read [1] but the exact meaning is not important here.

It is clear that although this expression is complete, precise, and abstract, it is not useful as a reference document. Moreover, it is very tiresome to check the correctness of this expression. Simply counting the parentheses is difficult and a true semantic check would be very difficult and time consuming for most software developers. With a great deal of effort one

could make minor improvements in the format and construction of this expression, but our experience is that software developers asked to write or read documentation in this form will not do so willingly and will argue that it is not worth the effort.

However, since 1977 [6], the senior author has been using a novel form of expression, which we call tabular expressions. A tabular expression that is equivalent to Fig. 1 is shown in Table 1.

(N(T)=2∧keyOK∧(¬(T=_)∧N(p(T))=1)) ∨
(N(T)=1∧(T=_∨(¬(T=_)∧N(p(T))=1)) ∧
(¬keyOK∧¬prevkeyOK∧¬prevkeyesc))
∨((¬(T=_)∧N(p(T))=1) ∧((¬keyOK∧keyesc∧
¬prevkeyesc) ∨(¬keyOK∧keyesc∧
prevkeyesc∧prevexpkeyesc)) ∨
((N(T)=N(p(T))+1) ∧(¬(T=_)∧
(1<N(p(T))<L)) ∧(keyOK)) ∨
((N(T)=N(p(T))-1)) ∧(¬keyOK∧¬keyesc∧
(¬prevkeyOK∧prevkeyesc∧preprevkeyOK)∨
prevkeyOK) ∧((¬(T=_)∧(1<N(p(T))<L)) ∨
(¬(T=_)∧N(p(T))=L))) ∨((N(T)=N(p(T))) ∧
(¬(T=_)∧(1<N(p(T))≤L))∧((¬keyOK∧
¬keyesc∧(¬prevkeyOK∧prevkeyesc∧
¬preprevkeyOK)) ∨ (¬keyOK ∧¬prevkeyOK
∧ ¬prevkeyesc) ∨(¬keyOK∧keyesc∧
¬prevkeyesc) ∨ ¬keyOK∧keyesc∧
prevkeyesc∧prevexpkeyesc)) ∨
((N(P(T)=Fail) ∧(¬keyOK∧keyesc ∧
prevkeyesc∧¬prevexpkeyesc) ∧
(1≤N(p(T))≤L)) ∨ ((N(P(T)=Pass) ∧(¬(T= _)∧
N(p(T))=L) ∧(keyOK))

Fig. 1. Example of an abstract description that is too difficult to read

Tabular expressions allow the expression to be parsed so that it is easy to check for completeness, consistency, and correctness. To check for completeness and correctness of this table, one makes sure that the cases in the header grids are mutually exclusive and cover all cases. One can check the correctness one case (cell) at a time.

Our experience with expressions in this form has been that practitioners will use them and prefer them to long textual paragraphs because they can quickly and easily find the information that they need. Our experience also suggests that errors are less likely using tabular expressions.

When tabular expressions were first introduced to the Software Engineering community, some called them "semi-formal". In fact, they are fully formal - their meaning has been defined in a variety of ways. We define them by showing how an equivalent conventional expression. The above is only one of many types of tabular expressions that a practitioner can use. The same relation can be described in many ways; the practitioner is free to choose the one that is best for readers or writers. The best form for those who prepare a table may not be the best for use as a reference. Algorithms for converting from one form to another have been developed [18].

While a glance at the small examples that fit in publishable papers may give the impression that such expressions are only practical for small examples, our experience shows the contrary. The newer models of tabular expressions that we are using allow the following techniques to be used when things get complex:

- The headers can be hierarchically structured as illustrated above.
- The dimension of the table can be increased from two to three, four or more to allow a speedier "look up".
- The expressions in cells can be tabular expressions.

## 6. Example of precise documentation – TFM specifications

As an example of precise documentation, we consider the Trace Function Method (TFM) for specifying or describing components or modules. TFM has been successfully applied for two industrial projects [16, 1] and is being used at the Software Quality Research Laboratory in Limerick for several projects.

TFM specifications determine values of every output of a software component (module) as a function of the history of events affecting this module. An event is a discrete point in time when the component reads or changes the values of global variables. Each event is described by an event descriptor, which contains names and values of the global variables before and after the event. We call a sequence of event descriptors a trace.

We use tabular representations of mathematical functions. The tables can be created systematically, identifying conditions that divide the traces into mutually exclusive and exhaustive subsets at the every step.

To illustrate TFM, we consider an electronic date display device (Fig. 2).

Table 1.

Tabular expression

**N(T)=**

| | | | **T = _** | **¬(T = _ ) ∧** | | |
|---|---|---|---|---|---|---|
| | | | | **N(p(T))= 1** | **1<N(p(T))< L** | **N(p(T))= L** |
| keyOK | | | | 2 | N(p(T))+1 | Pass |
| ¬keyOK ∧ | ¬keyesc ∧ | (¬prevkeyOK ∧ prevkeyesc ∧ preprevkeyOK) ∨ prevkeyOK | | | N(p(T))-1 | N(p(T))-1 |
| | | ¬prevkeyOK ∧ prevkeyesc ∧ ¬preprevkeyOK | | | N(p(T)) | N(p(T)) |
| | | ¬prevkeyOK ∧ ¬prevkeyesc | 1 | 1 | N(p(T)) | N(p(T)) |
| | keyesc∧ | ¬prevkeyesc | | 1 | N(p(T)) | N(p(T)) |
| | | prevkeyesc ∧ ¬prevexpkeyesc | | Fail | Fail | Fail |
| | | prevkeyesc ∧ prevexpkeyesc | | 1 | N(p(T)) | N(p(T)) |

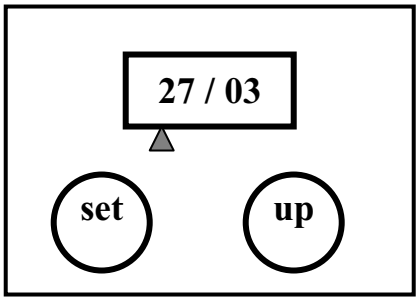


Fig. 2. Electronic date display device

The date display device shows the current date (day and month) and has two buttons ("set" and "up") to set up an initial date. The "set" button determines the mode (i.e., what we are going to set) and switches between "day" and "month". The small triangle indicates the current mode. The "up" button increments the selected quantity by one until "day" equals 31 or "month" equals 12 and then starts the cycle from 1. The initial date is 1/1.

For this example, a trace describes a sequence of button presses, e.g. T1= set.up.up.up.up or T2= sup.up.set.set.up.set.up.up.up. For every trace T, the output function returns a 3-tuple (S(T), D(T), M(T)), where S(T) ∈ {day, month} is the current mode, D(T) ∈ {1, 2, …31} is the day displayed, and M(T) ∈ {1, 2, …12} is the month displayed. We use two standard functions to reflect the history of operation: r(T) is the most recent event descriptor in Trace T.; p(T) is the trace T with the most recent event descriptor removed. For example, if T=set.up.set then r(T)=set and p(T)=set.up. We can describe the required behaviour of the electronic date display device using one tabular expression (Tab. 2).

For real software, TFM tables can be more complicated and additional tables for auxiliary functions are often necessary. However, even for complex programs, the size of a TFM specification is often surprisingly small. This is because the documents are abstract, i.e. they do not show any implementation details or models of those details.

## 7. Use of documentation at different stages of software product development and maintenance

**Software design.** One of the tools used to increase the trustworthiness of mechanical and electrical problems is very demanding review of designs

*before* implementation. Using mathematical tools such as differential equations, one can estimate such quantities as power consumption, heat production, and distortion from circuit diagrams and other design documents. As software is now developed, there is no possibility of doing a careful analysis until code is produced because the documentation, if any is produced, is not precise enough.

The precise mathematical documentation discussed in this paper changes that situation. Research on relational methods such as those described in [17], gives us equations that must be true if a design is correct. Without module interface specifications and internal design documents, these requirements remain in the theoretical domain. With such documents, one can check designs before proceeding further with the implementation. This can save a lot of wasted effort that comes from correctly implementing an erroneous design.

**Software inspection.** While, it is obviously "a good thing" to have design documents that were checked, it is the code that has to be correct if we are to trust software products. It is quite possible to have a very good design but a fatally flawed implementation. Inspecting the code to eliminate the many minor errors that are common has proven incredibly difficult. Even "productive" inspections, inspections that reveal many errors, often fail to find other errors.

Table 2.

TFM specification of Electronic date display device

$(S(T), D(T), M(T)) \equiv$

| T = _ | | | | |
|---|---|---|---|---|
| | | | | (day, 1, 1) |
| ¬ (T = _) ∧ | r(T)=set ∧ | S(p(T))=day | | (month, D(p(T)), M(p(T))) |
| | | S(p(T))=month | | (day, D(p(T)), M(p(T))) |
| | r(T)=up ∧ | S(p(T))=day ∧ | D(p(T))=31 | (S(p(T)), 1, M(p(T))) |
| | | | ¬ (D(p(T))=31) | (S(p(T)), D(p(T))+1, M(p(T))) |
| | | S(p(T))=month ∧ | M(p(T))=12 | (S(p(T)), D(p(T)), 1) |
| | | | ¬ (M(p(T))=12) | (S(p(T)), D(p(T)), M(p(T))+1) |

Software has many subtle interactions between components and one must have an incredible memory for detail to find the errors.

Experience reported in [12] and [10] has shown that precise documentation enables us to conduct very effective software inspections. The software is decomposed into a set of components and each component presented as a set of displays. Using descriptions of the components in the displays and the component interface specifications, we can inspect the displays one-at-a-time and know that if each display is correct, the whole system is correct. This is discussed in more detail in [10].

**Software testing.** Testing is an essential tool for increasing confidence in our products. However, testers in the software area work with three handicaps:

- Because of the lack of precise documentation, preparation for the tests is delayed until code is available and must be done under extreme time pressure.
- Useful coverage measures are "white box" that is they depend on the code.
- When test results are obtained, there can be disagreement on whether the behaviour is correct or not.

Precise documentation helps to alleviate all of these problems because:

- Preparation for tests can begin as soon as design documents are approved.
- Precise design documents tell us exactly which results are acceptable and leave no room for arguments over the correctness of results.
- Precise documents provide a basis for "black box" coverage measures. For example, we can make sure that each cell in a tabular expression is adequately tested.

We can use precise specifications to identify test values where past experience shows that errors are more likely. For example, if we know where certain functions pass through zero or have singularities, we can be sure to test these "interesting points" [3]. Other "interesting values" are the boundaries between different rows and columns in a tabular specification.

Precise documentation is also helpful for statistical testing. The purpose of statistical testing is reliability estimation. Reliability estimation requires an operational profile – a probabilistic characterization of how a software system is expected to be used [2, 9]. This profile is used to make sure situations are tested with a frequency proportional to their expected occurrence. Precise specifications can be used as a basis for describing the operational profile by associating probabilities with each cell in a tabular expression.

The use of precise documentation for improved testing is also discussed in [7, 14, 15, 19].

## 8. The practicality of the approach

Whereas many attempts at technology transfer are intended to transfer research ideas to industry, the ideas discussed in this paper originated with industrial problems and academic research has been used to refine them and put them on a solid mathematical foundation. The first use of tabular expressions was to document the requirements for real-time military software [6]. The U.S. Naval Research Laboratory, where the ideas were first developed has built tools and continues to work with Navy contractors on practical problems. These ideas were then applied in a telephony system at Bell Labs [7], where they were subsequently emulated by other projects. The inspection methods discussed in [12] and [10] were proven to be extremely effective by subsequent experience; in 15 years of use and modification, no missed errors were discovered [21]. More recently, experience using new forms of the documentation has been found to be effective in joint work with Dell [1] and Ericsson [16]. We usually get a mixed reaction from our industrial partners. Engineers like the brevity and precision of documents that they get from others, but feel that they do not have time to prepare them. Managers like the abstract idea of producing precise design documents but seem unwilling to allow for the "up-front-investment" that is needed to produce them. Producing these documents requires that we take the time to answer questions that are usually not answered until the coding phase. Some view this time as unproductive. Our experience has been that the coding phase is shortened and testing times can be reduced if this investment is made. It is difficult, however, to engender faith in future pay-back in managers who have found earlier documentation efforts or earlier attempts at "formal methods" to have little value.

We believe that the first areas to take up these methods must be projects where the software is critical, either for safety reasons or because it is needed for commercial success. In the safety area, our experience suggests that the first step should be to introduce more demanding standards for documentation, testing and inspection [20]. There are now many standards documents but because they do not demand precise abstract documentation they are effectively "toothless". Mathematical methods can put the teeth into such standards.

## 7. References


[1] Baber, R., Parnas, D.L., Vilkomir, S., Harrison, P., O'Connor, T., Disciplined Methods of Software Specifications: A Case Study, Proceedings of the International Conference on Information Technology Coding and Computing (ITCC 2005), April 4-6, 2005, Las Vegas, NV, USA, IEEE Computer Society.

[2] Brown, J.R., Lipow, M., Testing for software reliability, Proceedings of the International Conference on Reliable Software, Los Angeles, California, 1975, pp. 518 – 527.

[3] Clermont, M., Parnas, D. L., Using Information about Functions in Selecting Test Cases, ICSE 2005 Workshop on Advances in Model-Based Software Testing (A-MOST), St. Louis, Missouri, USA, May 15-16, 2005, IEEE Computer Society.

[4] Dijkstra, E. W., Notes on Structured Programming, in Structured Programming, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Editors, Academic Press, London,1972, pp. 1–82.

[5] Heninger, K.L., Specifying Software Requirements for Complex Systems: New Techniques and their Application, IEEE Transactions Software Engineering, Vol. SE-6, January 1980, pp. 2-13.

[6] Heninger, K.L., Kallander, J., Parnas, D.L., Shore, J.E., Software Requirements for the A-7E Aircraft, NRL Memorandum Report 3876, United States Naval Research Laboratory, Washington D.C., November 1978, 523 pp.

[7] Hester, S.D., Parnas, D.L., Utter, D.F., Using Documentation as a Software Design Medium, Bell System Technical Journal, 60, 8, October 1981, pp. 1941-1977.

[8] Hoffman, D.M., Weiss, D.M. (eds.), Software Fundamentals: Collected Papers by David L. Parnas, Addison-Wesley, 2001, 664 p., ISBN 0-201-70369-6.

[9] Musa, J.D., Operational profiles in software-reliability engineering, IEEE Software, Vol. 10 Issue 2, March 1993, pp. 14-32.

[10] Parnas, D.L., Inspection of Safety Critical Software using Function Tables, Proceedings of IFIP World Congress 1994, Volume III, August 1994, pp. 270 - 277. Reprinted as Chapter 19 in [8].

[11] Parnas, D.L., Precise Description and Specification of Software, in Mathematics of Dependable Systems II, edited by V. Stavridou, Clarendon Press, 1997, pp. 1 - 14. Reprinted as Chapter 5 in [8].

[12] Parnas, D.L., Asmis, G.J.K., Madey, J., Assessment of Safety-Critical Software in Nuclear Power Plants, Nuclear Safety, vol. 32, no. 2, April-June 1991, pp. 189-198.
[13] Parnas, D.L., Madey, J., Functional Documentation for Computer Systems Engineering, Science of Computer Programming (Elsevier) vol. 25, number 1, October 1995, pp. 41-61. Reprinted as Chapter 1 in [8].
[14] Peters, D., Parnas, D.L., Using Test Oracles Generated from Program Documentation, IEEE Transactions on Software Engineering, Vol. 24, No.3, March 1998, pp. 161 – 173.
[15] Peters, D., Parnas, D.L. Requirements-based Monitors for Real-Time Systems, IEEE Transactions on Software Engineering, Vol. 28, Issue 2, Feb. 2002, pp. 146-158.
[16] Quinn, C., Vilkomir, S.A., Parnas, D.L., Kostic, S., Specification of Software Component Requirements Using the Trace Function Method, Proceeding of the International Conference on Software Engineering Advances (ICSEA 2006), Oct. 29 – Nov. 1, 2006, Tahiti, French Polynesia.
[17] Schmidt, G., Ströhlein, T., Relations and Graphs - Discrete Mathematics for Computer Scientists. EATCS Monographs on Theoretical Computer Science. Springer, 1993, 301 p.
[18] Shen H., Zucker J.I., Parnas, D.L., Table Transformation Tools: Why and How, Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS '96), Gaithersburg, MD., June 1996, pp. 3-11.
[19] Vilkomir, S.A, Tips, P., Parnas, D.L., Monahan, J., O'Connor, T., Evaluation of Automated Testing Coverage: a Case Study of Wireless Secure Connection Software Testing, Supplementary proceedings of the 16th IEEE International Symposium on Software Engineering Reliability (ISSRE 2005), November 8-11, 2005, Chicago, Illinois, USA, pp. 3.123-3.134.
[20] Vilkomir, S.A., Bowen, J.P., Ghose, A. Formalization and assessment of regulatory requirements for safety-critical software, Innovations in Systems and Software Engineering - A NASA Journal, Vol. 2, Num. 3-4, December 2006, pp. 165-178.
[21] Wassyng, Alan. Private Communication.