Proceedings of the 5th WSEAS International Conference on Applied Computer Science, Hangzhou, China, April 16-18, 2006 (pp931-936)

A Novel Metric of Software Quality: Structural Availability

SHUBIN CAI[1], SHIXIAN LI[1], ZHONG MING[2], SISHAN GU[1] and XINHONG ZENG[2]

[1] Computer Science Department, SUN Yat-sen University, 510275, Guangzhou, China

[2] Information Engineering Faculty, Shenzhen University, Shenzhen, 518060, China

Abstract: Most present researches of software quality separate software availability from hardware availability thoroughly. However, software availability is defined as the probability that software is operating according to requirements at a given point in time. Hardware error, fault or failure surely has negative impact on software operation and decreases software availability. Applications are running on Operation System and/or some platforms, which are called runtime environment. Users tend to consider unrepeatable and unobvious hardware or environmental errors as application errors. Therefore, such factors should be considered in software availability. Abstract machine is a theoretical foundation for software programming. An Abstract Machine with Hardware Reliability (AM-HR) and an Abstract Machine with Environment Reliability (AM-ER) is proposed in this paper. Based on AM-ER, A novel metric named structural availability of software is presented. Utility of this metric in Change Management is shown to exhibit the promising prospect of structural availability.

Key-Words: Software Quality, Metric, Software Availability

1 Introduction

Today, great reliance is placed on software products, to the point where software has assumed a critical and strategic role in organizations' business[1]. Though software industry hasn't completely overcome the "software crisis", now it is facing a bigger challenge. That is to develop more complicated software products within the constraints of time and resources and without the sacrifice of quality.

Although people have been discussing software quality for decades, software quality research is still relatively immature, and it is difficult for a user to compare software quality across products [1, 2]. Stakeholders with different job roles are found to focus on different sets of software characteristics. It has been said that there are as many definitions of quality as writers on the subject [3]. We summarized such views as below: 1.) the product-based view, such as testing results or mathematical proofs based on formal specification; 2.) the manufacturing/ process-based view, which focus on Quality Assurance or Quality Management in a software process; 3.) the user-based view, which often have surveys for customer satisfaction; 4.) the project view, such as the earned-value approach in [4].

In spite of such different aspects, software quality has come to some consensus. The most widely used models of software quality are McCall Model, Boehm Model and ISO/IEC 9126 Model. These models classify software quality into (factors) characteristics, criteria and metrics, which are hierarchical models. Metrics are measured to indicate characteristics of software quality. In fact, most of the metrics are measured by experts' qualitative evaluation. Only a few quantitative metrics are measured from testing, which can be regarded as in a more engineering way.

In this paper, we proposed a novel metric of software quality named structural availability. This metric is measured quantitatively, thus it can be compared between software. The paper is organized as follow. Section 2 is an overview of the related work. In section 3, we extended a popular Abstract Machine with hardware and environment reliability. In section 4, we proposed the definition of structural availability from structural analysis of software. In section 5, we exhibit the utility of structural availability in change management. Then we come to a conclusion in section 6.

2 Related Work

Boehm classified software quality of an organization into 3 levels [5]. In Consciousness level I, we deliver software according to documented requirement specification. In level II, customers' satisfaction plays a central role in software quality. In level III, all the systems' success-critical stakeholders' concerns are negotiated and mutually satisfactory or win-win set of quality factors are achieved. Currently, most organizations are in level II. Many researches in the field still focus on customers' satisfaction. However, we provide some guidelines for negotiation with customers in Change Management to complete a win-win goal in this paper, which is an attempt to Level III.

The Earned-value approach [4] is a new and effective quantitative measurable metric of economic factor of software. A project is broken down into several tasks and each task is associated with budgets, schedules and earned-value. As project proceeding, Budgeted Cost of Work Scheduled (BCWS), Budgeted Cost of Work Performed (BCWP) and actual Cost are reviewed. If BCWP is significantly less than BCWS or actual Cost, or both, the project is overrunning its schedule, budget or both, and corrective action must be performed. This valued-base approach is very useful for economic of software quality, while the structural availability is a metric of reliability of software. Both these metrics complete the metrics for software quality.

Nassib[6] summarized software reliability measurement researches, which extrapolate the mathematics of hardware reliability theory to the prediction of software reliability. Simple measures of MTBF (Mean Time Between Failure), MTTF (Mean Time To Failure), MTTR (Mean Time To Repair) and Availability are used, where

$$MTBF = MTTF + MTTR \text{, and}$$

$$Availability = \frac{MTTF}{MTTF + MTTR} \times 100\%$$

Such metrics are overall and run-time measurement of software quality. Prediction models are proposed to calculate availability based on LOC (line of codes), default found in testing and historical records of the organization. These metrics have less utility on analyzing, designing and implementing phases of a software application. While using sensitivity analysis, structural availability can be used to improve the designing and implementing phases of the software.

Bernad Wong tried to find out which metrics are appropriate as a measure for each characteristic (usability, functionality, operational, technical, institutional, service and economic) of software based on value-chain models [3]. His survey showed that Values (sense of belonging, excitement etc) and Consequences (be more efficient, less stressed, meets expectations etc) metrics are appropriate for the measurement. Since usability is much broader than availability, availability isn't examined in detail here. Chulani et al [7] tries to find out metrics having the most significant impact on customer satisfaction based on CUPRIMDSO (Capability, Ease of use, Performance, Reliability, ease of Installation, Maintainability, Documentation, Service/Support, Overall satisfaction of the product) Model. As the number of PMR (Problem Management Report) or Critical Situations (a critical problem report) received increases, customer satisfaction with Reliability tends to decrease. Though structural availability is proposed and regarded as a metric of software reliability, surveys like these researches may be helpful to confirm our intuition, which will be carried out in our further research.

In order to improve the objectivity of software quality evaluation, Li [8] proposed a practicable software evaluation process model based on some quantitatively measured metrics of software quality, for example, Default Density. The structural metric can be serves as another objective metric for software quality evaluation in their evaluation process model.

In [9] a software quality management model and platform based on CMM is proposed to help software organization achieve the high level maturity. Our new metric can also be used as a metric for evaluating in their model.

Almost every researcher in software quality area agrees that quality should be viewed objectively and unmeasured quality is just talk. Measurements are essential to improving existing processes and methodologies over time, and gauging future software projects. But by now, only a few metrics of software quality can be measured objectively. The Structural Availability is a novel metric that can be measured objectively. We believe that it will enhance the software quality measurement research and application.

3 Extended Abstract Machine

If we consider a computer-based system, simple measures of software reliability are MTBF, MTTF, MTTR and Availability. Although software quality is thought to be irrelevant to hardware quality, it may not that irrelevant when software availability and customer satisfaction is concerned.

Availability measure is defined as the probability that software is operating according to requirements at a given point in time. Hardware error, fault or failure surely has negative impact on software operation and decreases software availability. Further more, application users tend to consider unrepeatable and unobvious hardware or environmental errors as application errors, and lower down their satisfaction with software.

For example, **have you ever rebooted computer to "fix" "software problems"?**

If errors are unrepeatable, how sure can you say that it is a software problem? One may argue that the problem is related to unrepeatable and unpredictable run-time environment. But, have you ever rebooted computer to **fix operation system start-up problem?** In this case, less environmental factors are involved.

Or, more usually, have you ever refreshed a web page to fix a display problem? We are aware of hardware failures and faults, but we are used to ignoring hardware errors. Take Network devices for example, physical circuit never promises a high reliability. The reliability is achieved by error control of upper network devices or software such as hardware implementing Data Link Layer protocol or software implementing Transport Layer protocol. Though high reliability is declared by TCP, almost everyone has encountered transport error, such as downloading an incomplete file. Hardware errors are commonly encountered in our daily life using network-based applications. Updating online is a trend of software update management. The bigger the size of updating files and the larger the amount of users, the more probability of downloading error-embedded software occurred. Therefore, hardware and environmental factors should be considered in software availability.

On the contrary, we should note that hardware and environmental factors should not be considered in correctness of software quality, which is an important achievement of software quality in the early 1960, 70s. The underlying assumption of this paper is that software is or is to be developed correctly.

Abstract machine is a theoretical foundation for software programming. An Abstract Machine with Hardware Reliability (AM-HR) and an Abstract Machine with Environment Reliability (AM-ER) is extended from a traditional Abstract Machine.

Definition 1. Abstract Machine AM = (C, E, S, $\Rightarrow$), where C is the sequence of instructions (or code) to be executed, E is the evaluation stack containing intermediate values during execution and S is the storage storing the values of variable in the instructions. AM has configurations of the form <c, e, s>$\in$Conf $\subseteq$ C$\times$E$\times$S and transition relation (execution) $\Rightarrow \subseteq$ Conf$\times$Conf.

Since AM is a basic notion in computer science field, the further explanation is omitted here. Syntax of instructions is of less relevance to our goal, also is omitted. Then we extended the AM to an Abstract Machine with Hardware Reliability, namely AM-HR.

CPU, Memory, Hard disk, Main board (I/O Bus) and other devices constitute a typical personal computer. These devices have much higher availability than Network devices. In particularly, web-based applications' availability should be concerned with hardware reliability.

Let *H* denote the set of hardware devices in interest, and ***2****(H)* denote the power set of *H*. Each hardware device *h*$\in$H has an availability denoted as *A(h)*, which can be accessed from hardware vendor. Each element *hs*$\in$***2****(H)* has an availability, denoted as *A(hs)*, which is calculated from

$$A(hs) = \prod_{h_i \in hs} A(h_i) \qquad \textbf{(1)}$$

Definition 2. AM-HR = (C, E, S, H, $\Rightarrow_{HR}$), where H is hardware involved in executing instructions, and C, E, S have the same meaning in AM. AM-HR has configurations of the form <*c, e, s, hs*>$\in$Conf-H $\subseteq$ C$\times$E$\times$S$\times$**2**(H) and transition relation $\Rightarrow_{HR} \subseteq$ Conf-H $\times$Conf-H $\cup$ {*undefined*}. If <*c, e, s*>$\Rightarrow$<*c', e', s'*> in AM, then <*c, e, s, hs*>$\Rightarrow_{HR}$<*c', e', s', hs'*> with probability of *A(hs)*, and <*c, e, s, hs*> $\Rightarrow_{HR}$ *undefined* with probability of 1 – *A(hs)*.

Though we don't assign a specific syntax of instructions of AM or AM-HR in this paper, *hs* can be recognized from analyzing each instruction in the syntax, and *A(hs)* can be calculated from formula (1).

The transition relation $\Rightarrow_{HR}$ becomes a probability relation in AM-HR. That means, only if the relevant hardware devices work properly, the transition can be done correctly as expected, otherwise transition will lead to an *undefined* value.

Traditionally, for an instruction sequence, we either have its finite computation sequence of transition to a terminal configuration where c=ε or have an infinite computation sequence named looping. Now, in AM-HR, we may have a finite computation sequence of transition ends with "*undefined*", we call it an **error configuration**. Computation or software becomes **unavailable** when error configuration encountered.

Definition 3.

If $< c_1, e_1, s_1, hs_1 > \Rightarrow^*_{HR} < c_n, e_n, s_n, hs_n >$, the availability of computation sequence of c_1 to c_n is denoted as $A(c_1 \Rightarrow^*_{HR} c_n) = \prod_{i=1}^{n} A(hs_i)$.

Theorem 1. Looping computation sequence has availability $A(c \Rightarrow^*_{HR} \varepsilon)$ = 0.

Proof: Firstly, each hardware device's availability *A(h)* <1.

Secondly, as shown in formula (1), each set of hardware devices *hs* has availability *A(hs)* calculated from *A(h)* , thus *A(hs)*<1.

Finally, a instruction sequence c causing looping computation sequence has availability $A(c \Rightarrow^*_{HR} \varepsilon) = \prod_{i=1}^{\infty} A(hs_i)$, where each *A(hs$_i$)* <1, then $A(c \Rightarrow^*_{HR} \varepsilon)$=0. □

This is an interesting result. Firstly, AM theory suppose we only care about execution result. If machine never halts, the useful result can't be got at any time. The computation sequence means nothing to us and obviously, its availability equals 0.

However, when software, such as Operation System or daemon programs is concerned, the AM assumption may not hold. The computation sequence is useful or available when it can react to our request, but not when it terminates. Thus, we can't take the availability $A(c \Rightarrow^{*}_{HR} \varepsilon)$ as the practical measurement of an application. A "*functional check point*" is introduced in section 4 to solve this problem.

When developing an application, usually we don't interact with hardware directly, but OS and other environmental services. AM-HR isn't powerful enough to manage this situation. We extended it to an AM-ER to deal with environmental factors.

What are environmental factors? We'd like to indicate that environment is defined according to application domain, i.e. environment can't be identified by AM itself. Environmental factors are imported from outside specification and granularity that we have interested in. Besides hardware factors, environmental factors can be recognized from among "*call*" instruction in AM. If a procedure call reference to a procedure in interest, it is not counted as environmental factor. Otherwise it could be counted as environmental factor, such as system invocation. For example, a socket-based application, when calling a read method of a socket, the following computation sequence in AM till "return" instruction of that "call" is marked as environmental factors. And we called such computation sequence as environmental computation sequence.

In AM-ER, the environmental computation sequence in AM-HR is compressed in a big step of configuration transition.

Definition 4. AM-ER = (C, E, S, Env, $\Rightarrow_{ER}$), where Env is the environment involved in executing instructions, and C, E, S have the same meaning in AM. AM-ER has configurations of the form <c, e, s, env>$\in$Conf-Env $\subseteq$ C×E×S×Env and transition relation $\Rightarrow_{ER} \subseteq$ Conf-Env ×Conf-Env $\cup$ {*undefined*}.

1.) If <*c, e, s, hs*>$\Rightarrow_{HR}$<*c', e', s', hs'*> in AM-HR and isn't marked as environmental factors, then *env=hs*, *env'=hs'* and <*c, e, s, env*>$\Rightarrow_{ER}$<*c', e', s', env'*> with probability of *A(env)*, and <*c, e, s, env*> $\Rightarrow_{ER}$ *undefined* with probability of 1 – *A(env)*.

2.) If <*c, e, s, hs*>$\Rightarrow_{HR}$<c_1, e_1, s_1, hs_1>$\Rightarrow_{HR}$ …$\Rightarrow_{HR}$<c_n, e_n, s_n, hs_n>$\Rightarrow_{HR}$<*c', e', s', hs'*> and <c_1, e_1, s_1, hs_1>$\Rightarrow_{HR}$ …$\Rightarrow_{HR}$<c_n, e_n, s_n, hs_n> is marked as environmental factors, then <*c, e, s, env*>$\Rightarrow_{ER}$<*c', e', s', env'*> with probability of *A(env)*, and <*c, e, s, env*> $\Rightarrow_{ER}$ *undefined* with probability of 1 – *A(env)* where *env'=hs'*, *env*= $\{hs, (c_1 \Rightarrow^{*}_{HR} c_n), hs_n\}$ and *A(env)*= $A(hs) \times A(c_1 \Rightarrow^{*}_{HR} c_n) \times A(hs_n)$.

In practical, it is very difficult to calculate the value of $A(c_1 \Rightarrow^{*}_{HR} c_n)$, since environmental factors are out of our control. We suggest assigning a symbol for every distinct system invocation at first. Then we can do some sensitive analysis after the overall availability formula is gained, and find out critical environmental factors, to improve the software quality.

Definition 5.

If $< c_1, e_1, s_1, env_1 > \Rightarrow^{*}_{ER} < c_n, e_n, s_n, env_n >$, the availability of computation sequence of c_1 to c_n is denoted as $A(c_1 \Rightarrow^{*}_{ER} c_n) = \prod_{i=1}^{n} A(env_i)$.

4 Structural Analysis

In practical, many useful applications don't fulfill the AM result-on-termination assumption. We introduced a functional check point to solve it.

When Functional Check Points (*FCP*) is met, the machine has just generated useful output for user. *FCPs* are among I/O function points specified in requirement specification. For example, in a method *printOrder* of a class *SaleOrder, FCP* is the set of "*return*" codes in the method, since printing a sale order is a meaningful output for users, and the body of *printOrder* has completed this function.

When doing structural analysis for structural availability, we should do as follow:

1. Identify the boundary between environment E and software S in interest
2. Calculate or estimate (structural) availability of relevant environmental factors $A(env_i)$, where $env_i \in E$. We suggest using symbols at first.
3. Identify functional check points and estimate the operation frequency of each functional check point $Fcp_i \in$S, denoted as $\lambda_S(Fcp_i)$, satisfying $\sum_{Fcp_i \in S} \lambda_s(Fcp_i) = 1$.
4. For each functional check point Fcp_i, find out reachable computation sequence C_j from the starting configuration of the software to the configuration with functional check point in current instruction, i.e. find out code paths to Fcp_i. The paths consist a set denoted as Path(Fcp_i), then estimate or calculate the average operation frequency of each $C_j \in$Path(Fcp_i), denoted as $\lambda_{Fcp_i}(C_j)$, satisfying $\sum_{C_j \in Path(Fcp_i)} \lambda_{Fcp_i}(C_j) = 1$
5. For each computation sequence C_j, calucate the availability of it by

$$A(C_j) = A(c_{j1} \Rightarrow^*_{ER} c_{jn}) = \prod_{i=1}^{n} A(env_i)$$

6. Calculate the availability of each functional check point Fcp_i

$$A(Fcp_i) = \sum A(C_j) \times \lambda_{Fcp_i}(C_j)$$

7. The overall availability of software S is called the structural availability of S,

$$SA(S) = A(S) = \sum_{Fcp_i \in S} A(Fcp_i) \times \lambda_S(Fcp_i)$$

$$= \sum_{Fcp_i \in S} \sum_{C_j \in Path(Fcp_i)} A(C_j) \times \lambda_{Fcp_i}(C_j) \times \lambda_S(Fcp_i) \quad \textbf{(2)}$$

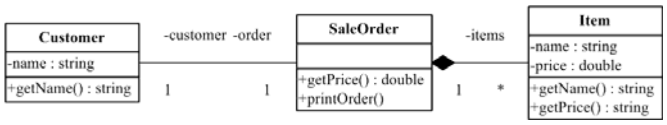

Fig. 1. A diagram of SaleOrder, Customer and Item

We will illustrate how to calculate the structural availability using the class diagram. Figure 1 is a very simple example of a sales application.

printOrder prints the names of customer and items using the *getName* methods in their corresponding classes and a *print* function provided by system, which is reconginzed from the sequence diagram. *Item* is deployed as a local Class while *Customer* is a remote class from deploy diagram. The Network-relevant devices has availability of *al*, while Network-irrelevant devices has availability of *ah*. *Print* is a system invocation has availability *am*. *printOrder* completes a meaningful function to users and the *return* instruction in which is regarded as a functional check point. From the entrance point to the exit point, there exist many different computation sequences, though only an instruction sequence is written. For example, the *getName* method of class *Item* may be called 1, 2, 3 or more times in different computation sequences C_1, C_2 or C_3. These computation sequences have different frequency, denoted as $\lambda_{printOrder}(C_1), \lambda_{printOrder}(C_2)$ and $\lambda_{printOrder}(C_3)$ etc. The availability of these computation sequences is calculated as

$$A(C_1) = ah \times al \times am, A(C_2) = ah^2 \times al \times am$$

and $A(C_3) = ah^3 \times al \times am$ respectively.

And the overall structural availability is $SA(SaleOrder) = \sum A(C_i) \times \lambda_{printOrder}(C_i)$.

From the above formula, we can see that the more *Items* the *Order* contained in actural need, the more critial the *Item* class is in the sucessful operation of this system comparing with other components. Thus more attention should be paid to it. Changes made to it should be checked with more cautions.

From other sensitivity analysis of *SA*(*SaleOrder*) we can learn more about the quality of our system and prove the design of class diagram, sequence diagram and deploy diagram.

We have only one *FCP* in the above case. If getPrice in *SaleOrder* also writes the result on the screen, it can be regarded as a FCP. We suppose user use *getPrice* 9 times than *printOrder* in using the application, which means $\lambda_{SaleOrder}(printOrder)=0.1$ and $\lambda_{SaleOrder}(getPrice)=0.9$. Then the structural availability of this application is 0.1×A(*printOrder*) +0.9×A(*getPrice*), where A(*printOrder*) and A(*getPrice*) can be calculated similar to the first case. And similar sensitivity analysis could also be done to find out critical components in this system.

We havn't investigated software fault tolerance and recovery mechanisms in detail here.but showing a simple example. Suppose p_1, p_2 are two compoents having similar function, if one works correctly, then *p* can generate meaningful result to users with coordinating component p_3. Then $SA(p) = SA(p_3) \times [1 - (1 - SA(p_1))(1 - SA(p_2))] = SA(p_3) \times [SA(p_1) + SA(p_2) - SA(p_1)SA(p_2)]$

5 Structural Availability in Change Management

Today, we all admit that software requirement will continue changing. A well-functioning Configuration and Change management (CCM) is a major part of software quality assurance and becomes more critical for software success. But [10] showed that CCM is only supported in RUP (Rational Unified Process) , while it is ignored in MSF (Microsoft Solution Framework) and XP (*eXtreme* Programming). However, RUP still fail to answer a vital problem that **which changes should be accepted and which should not**. In private interviews with software project managers, they all admit that inexperienced engineers tend to accept all kinds of changes customers required without careful consideration and often make mistakes. The professional knowledge these engineers have ensure them to implement the system after all. But these mistakes usually lead to software project's exceeding of time and budget. Standish[11] carried out a famous long term investigation of software project named Chaos. The challenge project (time-exceeded or budget-exceeded) in chaos is 53% in 2004, much higher than 33% in 1996. While succeed rate is 29% in 2004 and 27% in 1996. Software industry cries for methods to handle this problem.

We tried to propose some guidelines to deal with this problem here. The assumption we have here is that users view product quality (maybe not always consciously) as the trade-off among reliability (or availability), time of delivery, and cost that best meets their needs. Thus, reasonable changes are classified into 3 categories according to their impact on *Structural Avaiability*.

1. Changes relevant to code sequences (functions, classes or modules) which have greater contribution to structural availability. We need more time to test and verify the changes in order to ensure high software availability. In a word, we don't encourage these changes. We should be cautious to accept such changes.
2. Changes will significantly change the code sequences' percentage order by contribution to structural availability, especially the greater ones. Though such changes are reasonable, we should not welcome it. These changes are out of our initial design purpose of this software. Since the center of the software shifted, our initial development plan may become nothing. We are more likely to fail in delivery the software in time and budget, if we accept these changes but without reconsideration and re-plan to time and budget. In a word, we are unwilling to accept this kind of changes. We should inform customers the big risks of accepting such changes.
3. Other changes. This kind of changes has less impact on structural availability. We may welcome these changes.

Project manager may negotiate with customers using suggestions guidelines stated above, in order to achieve a win-win goal of the project.

6 Conclusion

Abstract machine is a theoretical foundation for software programming. An Abstract Machine with Hardware Reliability (AM-HR) and an Abstract Machine with Environment Reliability (AM-ER) is extended from a traditional Abstract Machine. The AM-HR, AM-ER proposed in this paper bridge hardware, environment factors into account for software availability. This is a novel approach of availability research. The separation of hardware availability and software correctness in 1970s is a great step in software quality researches. But when software availability is concerned, hardware availability should be taken into consideration.

Based on AM-ER, A novel metric named structural availability of software is proposed. Change Management becomes more and more important in today's software developing. The problem "which changes should we accepte" for project manager is very hard to answer. Some suggestions to accept or reject a change are outlined. We believe that these guidelines will encourage a promotion in software project success rate.

7 Acknowledge

This research is supported by National Social Science Foundation of China (#05CTQ001), Guangdong Natural Science Foundation (#04011304), and Shenzhen Science Technology Plan (#200422).

References:

1. Bernard Wong, Sunita Chulani, June Verner and Barry Boehm: Second Workshop on Software Quality. International Conference on Software Engineering, Proc. of the 26th Inter. Conf. on Software Engineering. (2004) 780-782
2. Bernard Wong, June Verner, Sunita Chulani and Barry Boehm: Third Workshop on Software Quality. International Conference on Software Engineering, Proc. of the 27th Inter. Conf. on Software Engineering, (2005) 688-689
3 Bernad Wong: The Software Evaluation Framework 'SEF' Extended, Information and Software Technology, (2004) Vol 46, 1037-1047.
4. Barry Boehm, Li Guo Huang: Value-Based Software Engineering: A Case Study. Computer, (2003) Vol.36, 33-41.
5. Nancy Eickelmann, Jane Huffman Hayes, (eds.): New Year's Resolutions for Software Quality. IEEE Software, (2004) Vol. 21, 12-13.
6. Nasib S. Gill: Factors Affecting Effective Software Quality Mamagement Revisted. ACM SIGSOFT Software Engineering Notes, (2005) Vol. 30. 1-4.
7. S.Chulani, B.Ray, P.Santhannam, R. Leszkowicz: Metrics for Managing Customer View of Software Quality, Ninth International Software Metrics Symposium (METRICS'03), IEEE, (2003), 189-199
8. LIHu, SHI Xiaohua, YANG Haiyan, GAO Zhongyi: Software Quality Evaluating Technique. Journal of Computer Research and Development. (2002) Vol. 39, No. 1, 61-67
9. Li Mingshu, WANG Qing: Software Quality Management Based Process Control. ACTA ELECTRONICA SINICA. Vol.30, No.12A, 2032-2035
10. Wolfgang Zuser, Stefan Heil, Thomas Grechenig: Software Quality Development and Assurance in RUP, MSF and XP – A Comparative Study. In: International Conference on Software Engineering Proceedings of the third workshop on Software quality. ACM, New York (2005) 1-6.
11. Chaos, Standish. http://www.standishgroup.com/chaos_resources/index.php