**Schuster György**

# SZOFTVER MEGBÍZHATÓSÁG

*Az előző évi előadásunkban megvizsgáltuk, hogy mely tényezők befolyásolhatják a szoftverek megbízhatóságát és definiáltuk azokat a mérőszámokat, amelyek megbízhatóság mérésére alkalmasak. Ha a szoftver széles körben kerül felhasználásra, akkor viszonylag könnyű dolgunk van, mivel a nagyszámú felhasználás statisztikai értelemben úgy viselkedik, mintha egyedi felhasználás esetén hosszú ideig vizsgáltuk volna az alkalmazást. De mi történik akkor, ha a szoftver egyedi, vagy csak minimális számú példányban fut, ekkor „bajban vagyunk". Nem beszélve arról az esetről, amikor a már akkor valamilyen becslést szeretnénk kapni az elkészülendő termékről, amikor az még el sem készült. Intézetünkben pontosan erre az esetre kezdtünk meg egy kutatást. Ez az előadás az eddigi eredményeket és kutatási irányokat mutatja be.*

***Kulcsszavak**: szoftver, megbízhatóság, szint, tesztelés, folyamat*

## BEVEZETŐ

Klasszikus műszaki alkotások esetében viszonylag könnyű előre becsülni a megbízhatóságot. Az alkalmazott anyagok és technológiák viszonylag jól leírhatók és kellő mennyiségű információ áll rendelkezésre, hogy időbeli viselkedésük megjósolható legyen. Ez informatikai termékek esetén a hardverre is igaz. A szoftverek esetén sajnos ezt nem tudjuk elmondani. Ezért a hangsúly sokkal inkább áttevődik a tervezési fázisra, majd a megvalósításra. Ezt vizsgálni nehéz, ezért a fejlesztő szervezeteket kezdték vizsgálni a szervezeti felépítés és a működési folyamatok szempontjából. Ez ugyan ad támpontot ahhoz, hogy az elkészült terméktől milyen statisztikai mutatókat várhatunk el. Várható ez elég jól korrelál a cég besorolásával, de ennél jobb lenne, ha erősebb mutatókat használhatnánk.

### CMM és CMMI a szoftverfejlesztési képesség – érettség vizsgálata

A CMM (Capability Maturity Model) képesség-érettség modellt az 1980-as évek végén dolgozták ki az USA Védelmi Minisztériumának megbízásából szoftverfejlesztő szervezetek megbízhatóságának felmérésére és javítására. A módszer folyamat szemléletű megközelítést alkalmaz, amely nem csak szoftver fejlesztésben, hanem tetszőleges integrált rendszer jellegű termékfejlesztésben alkalmazható. Több országban a legszélesebb körben elfogadott alapmodellé vált.

Hatást gyakorolt az ISO/IEC 12207:1995 szoftveréletciklus folyamatokat leíró szabványra és az ISO/IEC 15504 (SPICE, Software Process Improvement and Capability dEtermination) szoftverfolyamat értékelési szabványtervezetekre is. CMMI (Capability Maturity Model Integration), mint a CMM szoftver-, mind a rendszerfejlesztésre alkalmazható integrált verziója 2000. év második felében a Carnegie Mellon Egyetem Software Engineering Institute fejlesztésében Amerikai Védelmi Minisztérium támogatásával.

A modell kidolgozásának célja az eddigi szoftver-, rendszer- és termékfejlesztésben leggyakrabban alkalmazott modellek összevonása egyetlen modellé, amelyet fejlesztéssel foglalkozó szervezet alkalmazhat. A CMMI tehát egy olyan keret, amelyből kiindulva különböző

szervezetek számára lehet modellt szabni folyamatainak minősítésére és ezek javítására.

A CMMI modell a következő szabványokat, modelleket és megközelítéseket vonja össze:

- Capability Maturity Model for Software (SW-CMM) v2.0 draft C;
- Electronic Industries Alliance Interim Standard (EIA/IS) 731, Systems Engineering Capability Model (SECM);
- Integrated Product Development Capability Maturity Model (IPD-CMM) v0.98;
- modell kompatibilis az ISO/IEC 15504 és a SPICE szabvánnyal.

A CMM és a CMMI szabványok eltérően a klasszikus minőségbiztosítási elvektől már nem csupán megfelelőséget definiálnak, hanem úgynevezett érettségi modelleket is alkalmaznak, mellyel az adott céget vagy szervezetet, illetve annak projektjeit minősítik.

A CMM öt érettségi szintet állít fel, ezek:

- kezdeti, vagy kaotikus szint;
- ismételhető szint;
- meghatározott szint;
- menedzselt szint;
- optimalizált szint.

A CMM szintek meghatározásához a cég teljes szervezeti egészét vizsgálják. Úgy tekintik a céget mint amelyben csak egyetlen folyamat van, a szervezeti szintű folyamat, ez maga a szoftverfejlesztési folyamat, amelybe beletartoznak:

- a szoftverfejlesztésben részt vevő emberek;
- a szoftverfejlesztésben alkalmazott technológia;
- a szoftverfejlesztésben alkalmazott módszerek;
- a szoftverfejlesztésben alkalmazott eszközök.

A szervezeti szintű folyamatnak bizonyos jellemzői/összetevői vannak. Ezen jellemzők alapján dönthető el, hogy a szervezet milyen érettségi szinten áll.

### Kezdeti vagy kaotikus szint

Tipikusan kezdő cégekre jellemző a tipikus túlvállalás. Ez azt jelenti, hogy több feladatot vállalnak be, mint ami a teljesítő képességük, illetve elvállalnak olyan feladatokat, amelyek jelentősen meghaladják a tudásukat.

Ennek egyenes következménye az, hogy a folyamatosan túllépik a határidő és a költségkeretet. A siker egyes személyek lététől és „hősiességétől” függ.

Egy sikeresen befejezett projekt nem jelenti azt, hogy a következő is sikeres lesz.

Az ilyen szervezetek vagy nagyon gyorsan a következő szintre lépnek, vagy megszűnnek.

### Ismételhető szint

Ezen a szinten már megjelenik a szoftverek konfiguráció kezelése és a cégnél létezik minőségbiztosítás. A projekteket nyomon követik, tervezik és létezik követelmény kezelés. A cég vezetése megfelelő és kellő hozzáértéssel rendelkezik.

Ezek a szervezetek már képesek a piacon működőképesek maradni és fejlődni a következő

szintre. Egy elindult projekt sokkal nagyobb valószínűséggel lesz sikeres, mint az előző esetben.

### Meghatározott szint

Minden pozitív jellemzőt „örököl" az előző szintről, ezeket javítja és kibővíti. A termék szintű menedzselés megjelenik, a különböző fejlesztői csoportok közötti kommunikáció követelmény, a kölcsönös belső auditok és szemlék megjelennek. Szervezeti szintű folyamatszemlélet szintén megjelenik.

Lényeges szempont, hogy a munkatársakat folyamatosan képzik.

Fontos megjegyzés, hogy ezek a cégek már nagyrészt specializáltak. Tehát nem foglalkoznak mindennel, csak azzal, ami a profiljuk.

### Menedzselt szint

A szoftver minőség menedzselése. A folyamatokat mennyiségi szempontból menedzseli. A spontán, vagy tervezett folyamatváltozások statisztikai vizsgálata megtörténik.

Ennek következménye, hogy a folyamatok nagy mértékben felgyorsulnak.

### Optimalizált szint

A folyamatváltozások menedzselése és optimális folyamatok keresése. A technológiai változások menedzselése, hibamegelőzés. Az esetleges eltérések, gyenge teljesítmények és hibák okainak folyamatos keresése.

A fentieknek köszönhetően a fejlesztési ciklusidők tovább rövidülnek és a fejlesztés biztonsága nagymértékben javul.

Érdekes megfigyelés, hogy csak az első szint ugorható át bizonyos segítséggel. A többi szint elérése nagyrészt folyamatos fejlődés következménye. Az is érdekes megfigyelés, hogy a felső két szint elérése nem csak tudásszint, hanem kulturális háttér függvénye is. Erre egy következő példa:

Egy jó nevű szoftver fejlesztő cég létrehozott egy leányvállalatot egy szoftverfejlesztésről híres ázsiai országban megcélozva az optimalizált érettségi szintet. Ez sehogy sem sikerült. Nagyon komoly erőfeszítésekkel a meghatározott szintet voltak képesek tartani.

### Hiba kalkuláció

A megbízhatóság számításánál célszerű a meghibásodás bekövetkezési valószínűséget számítani.

Ennek oka az, hogy a működési valószínűség $P_w$ komplementere a nemműködés a $P_f$.

Tehát:

$$P_w = 1 - P_f$$

Számos esetben a probléma matematikai tárgyalása így egyszerűbb.

Az elsőnek vizsgált eset az egyszálas futású program. Ez az eset gyakorlatilag nem fordul elő, azonban komoly jelentősége van a további vizsgálatok során.

Tegyük fel, hogy a ismertek a kérdéses program statisztikai adatai. Ekkor a legegyszerűbb matematikai megközelítés az adott időtartamra történő meghibásodásra a Poisson eloszlás:

$$P_f(\mathrm{X=k}) = \frac{\lambda^k}{\mathrm{k!}} e^{-\lambda}$$

ahol $\lambda > 0$az adott időtartamra a meghibásodás várható értéke, k = 1, 2, 3, ... pedig a meghibásodások száma.

Probléma: az általunk vizsgált esetben így megadható a meghibásodás valószínűsége. Kérdés továbbá is az, hogy honnan kapunk megbízható információt a $\lambda$ paraméterről.

Magyarázat: a szoftverek alapvetően jól konstruált, helyesen megvalósított alkotások. Így a hagyományos megfigyelési módszerek nem vezetnek eredményre.

Adott egy a fenti feltételeknek megfelelő szoftver. A szoftver már alkalmazásba került és több – akár – tízezer példány is fut különböző felhasználóknál. Az így összegyűjtött meghibásodási adatokat vizsgálva következtetést lehet levonni a keresett $\lambda$ paraméterre.

Példa: tegyük fel, hogy egy adott szoftver 1000 példánya fut. Az átlagos meghibásodási ráta 100 óránként egy meghibásodás erre a populációra. Ezt egyetlen futó programra vetítve 100 000 óránkénti meghibásodást jelent.

Azonban a szoftver tervezőinek szempontjából az lenne a kedvező, ha ezt az értéket már nem a futó program kibocsájtása után, hanem még – akár a tervezési fázisban – lehetne meghatározni.

Kérdés: lehet-e ezt a módszert konkurens rendszerekre alkalmazni?

A válasz: igen az önállóan futó programrészletek vizsgálatára egyértelműen, de csak az adott feladatokra és szálakra. Ezt a problémát tovább bonyolítja a közös erőforrás használat és a processzek közötti kommunikáció (Inter Process Communication).

Amennyiben feltételezzük, hogy egy önállóan futó és vizsgált szoftverelem meghibásodása a teljes rendszer hibás működéséhez vezet, akkor a számítható hiba valószínűsége független valószínűségi változók esetén:

$$P_{\mathrm{f}\Sigma} = \sum_{\mathrm{i=1}}^{n} P_{f_i}.$$

Feltételezzünk egy négy szálból álló alkalmazást (IPC kizárva), amely minden szálát tekintve azonos várható értékkel hibásodik meg, és egy meghibásodás a teljes rendszer hibás működéséhez vezet.

Ekkor eredményként azt kapjuk, hogy a meghibásodás valószínűsége négyszeresére nő[1].

IPC esetben a folyamatok közötti kapcsolat újabb hibalehetőséget rejt magában. Ennek klaszszikus esete a két folyamat közötti létrejövő halálos ölelés (dead-lock). Ez akár szemaforok, akár MUTEX-ek esetén előfordulhat.

[1]A komplexitás növekedése a megbízhatóságot csökkenti.

Példa:

Adott két folyamat A és B és két MUTEX X és Y. B prioritása magasabb, mint A prioritása. A kérdéses szituáció:

1. A folyamat fut és megszerzi X MUTEX-et;
2. az ütemező (scheduler) leállítja A folyamatot, mert B folyamat futásra kész és B-t elindítja;
3. B folyamat megszerzi Y MUTEX-et;
4. B folyamat blokkolttá válik X MUTEX-en;
5. A folyamat fut, mivel B blokkolt állapotba került;
6. A folyamat blokkolttá válik Y MUTEX-en.

A folyamat nem tud futni, mert Y MUTEX felszabadítására vár, és ezért nem tudja feloldani X MUTEX-et, amelyen B folyamat várakozik, és ezért B nem tudja felszabadítani Y MUTEX-et.

Ez a példa a legegyszerűbb eset, amelyet viszonylag könnyű észrevenni és kiküszöbölni. Probléma akkor van, ha ez a jelenség nem páronként következik be, hanem több folyamat kerül „egyfajta körbevárás" (cyclic blocking) állapotba.

Ennek az állapotnak a bekövetkezési valószínűsége becsülhető. Két folyamatra a következő módszer használható az előző példa alapján:

Tételezzük fel, hogy mind A, mind B folyamatok egyenletes valószínűségi eloszlás alapján használják X, illetve Y MUTEX-eket. Ez alapján:

- A folyamat normál futást tekintve $P_X$valószínűséggel tartózkodik X MUTEX által védett területen;
- B folyamat normál futást tekintve $P_Y$valószínűséggel tartózkodik Y MUTEX által védett területen;
- A folyamat futási valószínűsége $P_A$;
- B folyamat futási valószínűsége $P_B$;
- A és B folyamat egyidejű futásának valószínűsége: $P_{\mathrm{AB}}=\mathrm{P}_A P_B$;
- A két MUTEX „átlapolódásának" valószínűsége: $P_{\mathrm{dead-lock}}=\mathrm{P}_A P_B P_X P_Y$.

A megoldás nem tűnik bonyolultnak, viszont a kapott eredmény érezhetően „kedvezőtlen", ha futási – megbízhatósági szempontból nézzük. Tesztelési szempontból nagyon kedvező, mert a hiba gyorsan kiderül.

A fenti példa elnagyolt. Több tíz, esetleg több száz folyamat esetén ilyen jellegű hiba bekövetkezésének valószínűsége ciklikus lényegesen csökken, de felderíthetőségének esélye is drasztikusan csökken.

Hasonló eredményt kapunk, ha kissé eltávolodva a szoftver szempontoktól a hardveres hatásokat is vizsgálunk. Egy hardver által létrehozott állapot feldolgozása is időbe telik.

Tipikus esete ennek egy hálózati foglaltság feldolgozása, ahol a kérdéses egység a hálózati médiumot az adott pillanatban szabadnak érzékeli, de amire azt használtba venné, addigra egy másik egység már használja. Ezt az időt szoft időszeletnek nevezik.

Sajnos ez akkor is fennáll, ha az ilyen jellegű feldolgozást hardver végzi. Ezt különböző eljárá-

sokkal ki lehet küszöbölni, például CSMA/CD, vagy CSMA/CA. A kérdéses médium terheltségétől függően firm real-time és hard real-time esetben megengedhetetlen időbeli csúszások következhetnek be.

Ennek feltétele azonban az, hogy a rendszer erőforrásai a kritikus időszakban nem bizonyulnak elegendőnek. Ha a hardver és szoftver erőforrások elegendőek, akkor a rendszer a legkedvezőtlenebb állapotot is le tudja kezelni és nem kerül időzavarba. A számításra az általunk alkalmazott eljárás a következő:

$\rho_i$ jelöli az adott erőforrás használatának relatív gyakoriságát, ahol

$$\rho_i = \frac{\sum_j \tau_{\mathrm{ij}}}{T_i}$$

$\tau_{\mathrm{ij}}$ a kritikus szakasz időtartama, amíg a probléma jelentkezhet (az i-edik folyamatra számítva), ha ebben az időtartamban egy másik folyamat is el akarja érni a kérdéses erőforrást, akkor ütközés következik be.

$T_i$ a vizsgált időtartam (az i-edik folyamatra).

Ennek $T_i$végtelenbe vett határértéke $p_i$, a kritikus szakasz bekövetkezési valószínűsége.

Akkor keletkezik hiba, vagy eltérés, ha egy bizonyos számú folyamat ütközik. Ha egy adott erőforrás eléréséhez tartozó $p_i$valószínűségek egyformák, akkor egy adott rész állapot binomiális eloszlást mutat.

$$p_{\mathrm{ik}} = \binom{n}{k} p_i^k (1 - p_i)^{n-k}$$

A hiba vagy eltérés bekövetkezéséhez n folyamatból k folyamatnak kell az adott szakaszon bekövetkezni. Azonban akkor is hiba keletkezik, ha *k*-nál több folyamat lép be a kérdéses időben.

$$p_f = \sum_{\mathrm{j=k}}^{n} \binom{n}{j} p_i^j (1 - p_i)^{n-j}$$

ahol:

$p_f$ az eltérés, vagy hiba bekövetkezési valószínűsége.

Ennek elkerülése úgy lehetséges, hogy a rendszer tervezése során annyi erőforrást biztosítunk, hogy a legnagyobb terhelés esetén se léphessen fel probléma.

Memória szivárgás (memory leaking) a következő eset, amelyet vizsgálunk. Az eltérés, majd a hiba oka az, hogy egy adott folyamat memória jellegű erőforrást foglal le, amelyet adott valószínűséggel nem szabadít fel, ezáltal a rendelkezésére álló memória fokozatosan elfogy.

A hiba modell egy egyszerű alprogram, amely több kilépési ponttal rendelkezik az egyik kilépési ponton, ahol a kérdéses erőforrás felszabadítás nem történik meg $p_i$valószínűséggel, ezáltal csökkentve a rendelkezésére álló memória méretét.

$p_i$ becslése szintén a relatív gyakoriság segítségével határozható meg és egyenletes eloszlást feltételeztünk.

Tegyük fel, hogy:

$M_a$ a rendelkezésre álló memória;

$\Delta m$ az allokált, de fel nem szabadított memória mennyisége;

$k$ a futtatások száma;

$\tau$ a futtatások átlagos „periódus” ideje;

$T_f$ a hiba bekövetkezésének várható idő intervalluma.

Az eltérés és vagy a hiba bekövetkezésnek feltétele, az hogy a rendelkezésre álló memória elfogyjon.

$$0 > M_a - k\Delta m \text{ ebből } T_f > \tau \frac{M_a}{\Delta m}$$

A memória szivárgás hiba. Azonban számos esetben a szoftverfejlesztők nem foglalkoznak vele, hanem előírják a rendszer adott időnként történő újraindítását.

Következő kérdés a szoftver vizsgálata Markov folyamatként [5].

Markov folyamatnak az a sztochasztikus folyamat tekinthető, amelynek jövőbeli viselkedése csak a rendszer jelenlegi állapotától függ [5]. Röviden ezt a szakirodalom úgy fogalmazza meg, hogy a Markov folyamat nem emlékezik.

Ezt a peremfeltételt a szoftver rendszerek nem teljesítik. Erre egy nagyon egyszerű példa a rendszer adott időpillanatig történő erőforrás használata.

Vegyük a következő példát! Adott egy folyamat, amely komoly háttértároló hozzáféréssel dolgozik. Köszönhetően a nagy igénybevételnek a háttértároló tartalma töredezett lesz, így hozzáférési ideje (access time) meghosszabbodik. Ez egyértelműen egy múltbéli hatás.

Amennyiben a rendszer rétegekre bontott struktúrával rendelkezik a „felső” rétegek egyfajta felügyeleti és javító funkciókat is ellátnak – különös tekintettel a SIL3 és SIL4 besorolású rendszerek esetén. Ekkor sem célszerű a rendszert Markov folyamatként kezelni.

Az eddigiek folyamán olyan hibákat mutattunk be, amelyek valamilyen tervezési hiba alapján eltéréshez és vagy hibához vezettek. Minden esetben kerestük a részfolyamat bekövetkezési valószínűségét. Ezt a valószínűségi értéket azonban rendkívül nehéz meghatározni. Ennek oka, az, hogy a kritikus szakaszok bekövetkezési valószínűsége olyan kicsi, hogy a kérdéses folyamatot akár évekig kell megfigyelni, vagy nagyszámú elem megfigyelésével kell a kívánt értéket meghatározni.

További problémát jelent, hogy a szoftverek kivétel nélkül konkurens rendszerek. Így a megfigyelésük is jóval komplexebb feladat, mint bármilyen hagyományos műszaki rendszeré.

A szoftver gyártók és felhasználók a kockázati elemeket még a szoftver kibocsájtása előtt szeretnék meghatározni. Erre az előbb említett módszerek nem alkalmazhatók.

A megoldás a szoftver előállítójának megfigyelése. Az előállító várhatóan több szoftvertermékkel rendelkezik, amelyből éles körülmények között számos példány fut. Az ilyen szoftverelemek adatainak összegyűjtésével már értékelhető statisztikai mintával rendelkezünk.

Jelenlegi kutatásainkban azt vizsgáljuk, hogy a szoftverhibák bekövetkezése mennyire véletlenszerű, illetőleg az adott előállító milyen trendet mutat az hibák számát tekintve.

Erre a szoftver fejlesztési folyamatot úgy tekintjük, mint egy idősort és a kérdésre a választ a Hurst analízissel határozzuk meg [7].

Megjegyzés: a Hurst analízis idősorokra lett kifejlesztve, azonban a szoftver futása nem egyértelműen időfüggő annyira, hogy idő intervallumokat vizsgáljunk, ezért az időt a kódsorok számával helyettesítjük.

Adott számú kódsorra határozzuk meg az eltérések és hibák előfordulási számát.

Jelölje

$E_i$ adott számú kódsorra az eltérések és hibák száma, i = 1, 2, ... m;

$m$ a vizsgált idősorok száma;

$N$ a minták kódsorainak száma;

$E_N$ az átlagos hibaelfordulás.

$$E_N = \frac{1}{m}\sum_{\mathrm{i}=1}^{m} E_i$$

Számítsuk ki a vizsgált tartományok kumulált átlagait.

$$X(\mathrm{i,N}) = \sum_{\mathrm{i}=1}^{m} E_i - E_N$$

Majd számítsuk ki az $X(\mathrm{i, N})$ maximális és minimális értékének különbségét.

$$\mathrm{R}=\max X(\mathrm{i,N}) - \min X(\mathrm{i,N})$$

Ezt az értéket a teljes sorra vonatkoztatott standard tapasztalati szórással $S_N$(standard deviation) normáljuk.

Ekkor:

$$\frac{R}{S_N}=\mathrm{N}^H$$

A $H$érték az úgynevezett Hurst exponens, amelynek az értéke a hibákat és a trendeket jellemzi.

A H = 0,5érték azt jelenti, hogy a hibák előfordulása a hibák előfordulása véletlenszerű.

A H > 0,5érték azt mutatja, hogy az idősor erősen trendtartó. Tehát, ha egy időszakban a hibák csökkentek, akkor a következő időszakban ugyanaz a trend várható.

A H < 0,5 érték az idősorban hullámzást jósol. Tehát egy időszakos csökkenést emelkedés, illetve emelkedést csökkenés követ.

A módszer ígéretes. Problémája kettős. A szoftver előállítói a kisebb hibákat hajlamosak eltitkolni, mert így a cég hírnevén – esetleg – nem esik csorba. A másik probléma, hogy a felhasználók vagy nem ismerik fel az eltéréseket, vagy nem foglalkoznak a problémával. Mindkét esetben a statisztikai minta torzul.

## KÖVETKEZTETÉSEK ÉS ÖSSZEFOGLALÁS

A szoftver megbízhatóság valószínűségi jellemző. Nehéz olyan valószínűségi modellt találni, amely jól becsülhetővé teszi az értékét. Több szempontból vizsgálva kijelenthetjük, hogy egységes és minden szempontból megfelelő elméletet nem tudunk megnevezni.

Az ismertetett matematikai eljárások alkalmazásához is mindenképpen szükséges a kérdéses szoftverről valamilyen előzetes tudás, amelynek megszerzése sem történhet teljes bizonyossággal.

Az általunk kutatott eljárás a szoftver előállítójának vizsgálatát célozza. Az általa előállított termékek statisztikai vizsgálata valamilyen megfogható és körüljárható támpontot ad a szoftver további vizsgálatához.

Különböző módszerekkel, szabályokkal és teszteléssel a megbízhatóság javítható, de a tapasztalat azt mutatja, hogy hasonlóan a más ember által előállított rendszerekhez a szoftverek megbízhatósága nem lehet 100%. A megbízhatóság alapvetően a tervezés fázisában eldőlhet, de komoly problémát jelent, hogy az előállított kód emberfüggő.

**FELHASZNÁLT IRODALOM**


[1] Van Solingen, R. and Berghout, E. (1999) The Goal/Question/Metric Method: A Practi-cal Guide for Quality Improvement and Software Development. McGraw-Hill Inter-national.

[2] Tokody Dániel, Schuster György, Papp József Study of How to Implement an Intelligent Railway System in Hungary In: Anikó Szakál (szerk.) SISY 2015: IEEE 13th nternational Symposium on Intelligent Systems and Informatics: Proceedings. Subotica, Szerbia, 2015.09.17-2015.09.19. Subotica: IEEE Hungary Section, 2015. pp. 199-204.

[3] Long, J (ed.) (2008) Metrics Data Program, National Aeronautics and Space Administratio http://mdp.ivv.nasa.gov/index.htm

[4] Norman F. Schneidewind, "Reliability Modeling for Safety Critical Software", IEEE Transactions on Reliability, Vol. 46, No.1, March 1997

[5] Markov process (mathematics) - Britannica Online Encyclopedia

[6] Marszal, Edward, "Safety Integrity Level Selection – Systematic Methods Including Layer of Protection Analysis", The Instrumentation, Systems, and Automation Society, Research Triangle Park, NC, USA, 2002.

[7] Szilágyi Győző Attila: A légi balesetek fraktáldimenziója, (on-leni doc), url: http://www.repulestudomany.hu/folyoirat/2016_2/2016-2-14-0311_Szilagyi_Gyozo_Attila.pdf


***SOFTWARE RELIABILITY***


*We investigated the influential factors of software reliability and we defined those indexes, which are suitable to measure software reliability in our last year presentation .If the software is widely used it is quite easy to observe the given application. The reason for this that in statistically meaning the given software acts as single application would have run for very long time. But what does it happen when the observed software is unique or it has limited applications to run. In this case we are in „trouble". The other problem is that the software is not finished and we have to get some prior information on the product. In our institution we have started a new research on this topic. This presentation shows our new research trends and results.*



***Keywords**: software, reliability, level, testing ,process*



| | |
|---|---|
| Dr. Schuster György (PhD) | Dr. Schuster György PhD. |
| intézet igazgató | Director of Institute |
| Óbudai Egyetem | Instrumentation and Automation Kandó Kálmán Fa- |
| Kandó Kálmán Villamosmérnöki Kar | culty of Electrical Engineering |
| Műszertechnikai és Automatizálási Intézet | Óbuda University |
| schuster.gyorgy@kvk.uni-obuda.hu | schuster.gyorgy@kvk.uni-obuda.hu |
| orcid.org/0000-0002-8573-3670 | orcid.org/0000-0002-8573-3670 |


http://www.repulestudomany.hu/folyoirat/2017_2/2017-2-02-0363_Schuster_Gyorgy.pdf