HADMÉRNÖK

IV. Évfolyam 4. szám - 2009. december

Kun István
kunistvan47@gmail.com

INFORMATIKAI RENDSZEREK MEGBÍZHATÓSÁGÁNAK MATEMATIKAI MODELLEZÉSE

Absztrakt

A cikkben a szoftverek megbízhatóságának problémakörével foglalkozunk, amely lényegesen eltér a megbízhatóságelmélet általános problémakörétől. Ez utóbbi az informatikán belül döntően inkább a hardvermegbízhatóságra érvényes. Foglalkozunk a két problémakör különbségeivel. Ismertetjük az alapvető matematikai modelleket, amelyek két fő csoportra oszlanak: a metrikákra és a prediktív statisztikai modellekre. Kiemelten érdekesek a komplexitási metrikák, amelyek egymástól lényegesen különböző, de egyaránt mély matematikai alapokon nyugszanak. A komplexitási metrikákra épülnek a folyamat-mutatószámok, amelyek a szoftver tervezésének és kódolásának projektjét, a prediktív statisztikai modellek pedig a tesztelés projektjét segítik az időigények reális becslésében.

In this paper we deal with the problem sphere of the software reliability, essentially different from the general problem sphere of reliability. This latter is valid, within information technology, rather on hardware reliability. We deal with the differences of the two problem spheres. We expound the basic mathematical models, which have two main groups: metrics and predictive statistical models. Especially interesting are the complexity metrics, which are laid on two essentially different but equally deep mathematical bases. A realistic assessment of time consumption can help project planning, which is based in software design and coding mainly on complexity metrics, while in software testing mainly on predictive statistical models.

Kulcsszavak: *informatika, megbízhatóság, matematika, szoftver ~ information technology, reliability, mathematics, software*

1. Kritikus szoftverhibák

Mi az elmélet?

A gondosan tesztelt szoftver soha nem hibásodik meg, mert amíg a számítógép működik, a szoftver nem változik, nem öregedik.

És mi a valóság?

A tényleges helyzetet néhány példa teszi szemléletessé, l. (Pan, 1999):

- Falklandi háború (1982): a Sheffield rombolót azért süllyesztette el egy Exocet rakéta, mert a védelmi radarrendszer szoftvere a rakétát tévesen „barátinak" minősítette.
- Öbölháború (1991): egy Patriot ellenrakéta irányítórendszere 10 másodpercenként 0.000000095 sec-ot tévedett, és az akkumulálódó hibák miatt 100 órás működés után nem találta el a támadó rakétát (28 halott)
- Ariane 5 európai űrrakéta: az Ariane 4 jól működő szoftverében egy apró módosítást végeztek, és az Ariane 5-ben indításkor tűz keletkezett.
- USA, Kalifornia állam: egy helyi telefonközpontban egy kezelőprogramban 3 sort kicseréltek, és a teljes szolgáltatás megbénult.

2. A szoftver-megbízhatóság értelmezése

A hardvereszközök működőképessége egyedileg változik, és akár pillanatonként módosuló véletlen tényezőktől (elhasználódás mértéke, környezeti hatások) függ.

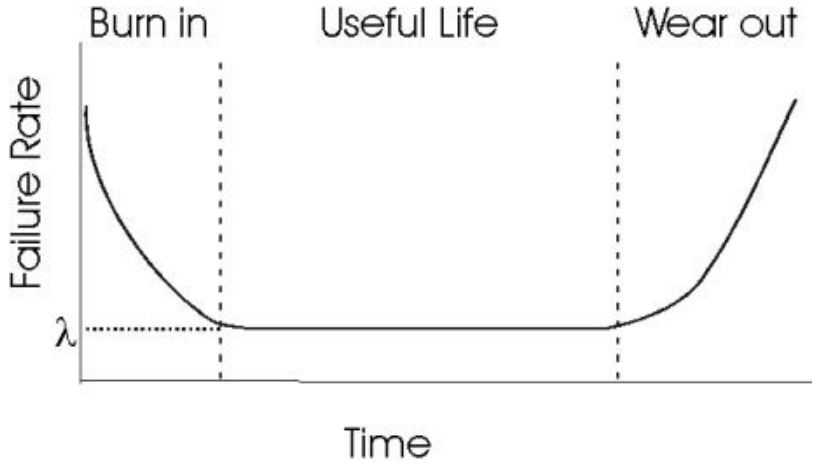
Ezzel ellentétben a szoftverek példányai azonosak, és – az adathordozók hibáitól eltekintve – időben állandóak. A hibátlan programrész mindig hibátlanul működik, a hibás programrész viszont mindig hibásan.

A szoftverek megbízhatóságának statisztikai jellegét tehát nem a szoftver belső tulajdonságai adják, hanem a használat módja: milyen gyakorisággal lépünk be a hibás programrészbe. Még ha a program tele van hibákkal, akkor sem vesszük ezt észre, ha nem használjuk a hibás programágakat. Tipikus példája ennek az a mikroprogram-hiba, amit a Pentium processzor piacra dobásakor egy matematikatanár fedezett fel, mert egy számelméleti algoritmus hibásan működött. A gyári teszteléskor senkinek nem jutott eszébe ilyen algoritmust kipróbálni.

A felhasználót a teljes rendszer hibátlan működése érdekli, gyakran nem is tud különbséget tenni a hardver és a szoftver hibája között. Tipikus szituáció az adathordozó meghibásodása miatt működésképtelenné váló szoftver. Ilyenkor gyakran nehéz elválasztani egymástól a kétféle hibát.

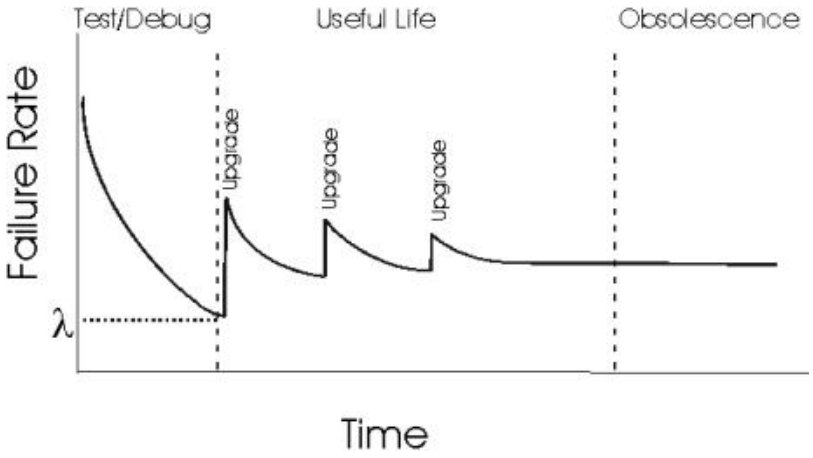
A következő táblázat rendszerezi az említett különbségeket.

HARDVER	SZOFTVER
A hiba oka egyaránt lehet a tervezés, a gyártás, a használat és a karbantartás.	A hiba oka majdnem mindig a tervezés.
A hiba lehet az elhasználódás vagy más, energiával kapcsolatos jelenség következménye. Ennek gyakran vannak korai figyelmeztető jelei.	Nincs elhasználódás. Nincsenek korai figyelmeztető jelek.
Javítással növelhető a megbízhatóság.	Az egyetlen javítási lehetőség az áttervezés (újraprogramozás).
A megbízhatóság függhet bejáratási vagy elhasználódási jelenségektől.	A megbízhatóság nem változik a a működési idővel, csakis a hibakeresésbe fektetett munkával.
A meghibásodás valószínűsége függ az eltelt működési (vagy tárolási) időtől.	A meghibásodás nem függ így az időtől. Akkor történik, amikor hibás programrészbe lépünk.
A megbízhatóság környezeti tényezőktől is függ.	A környezeti tényezők nem érintik a megbízhatóságot, legfeljebb az adathordozón keresztül.
A megbízhatóság elméletileg megjósolható a tervezés és a használat alapján.	A megbízhatóság semmilyen fizikai folyamat alapján nem jósolható, mert a tervezés emberi tényezőitől függ.
A megbízhatóság gyakran növelhető tartalékegységek alkalmazásával.	A megbízhatóság nem növelhető tartalékegységek alkalmazásával, ha a párhuzamos programrészek azonosak: amennyiben az egyik hibás, akkor a másik is. A tartalékolás csak akkor segít, ha a párhuzamos ágakban eltérő szerzők által készített eltérő programok vannak.
A rendszer komponenseinek megbízhatósága olyan törvényeket követ, amelyek a komponenseket ért terhelés és más tényezők alapján bizonyos mértékig megjósolhatók.	A hibák általában nem jelezhetők előre az egyes programlépésekre vonatkozó megállapításokból. A programhibák véletlenszerűen szóródva helyezkednek el, és bármelyik utasítás lehet hibás.



A hardver meghibásodási rátája az élettartam során kádgörbe szerint változik.

1. ábra. A hardver meghibásodási rátájának alakulása



A szoftver meghibásodási rátája farkasfog-szerű: az upgrade ciklus elején megnő, a ciklus végére lecsökken.

2. ábra. A szoftver meghibásodási rátájának alakulása

A szoftver megbízhatósága (és tőle elválaszthatatlanul minősége) tehát némileg kilóg a klasszikus hardver-orientált megbízhatóságelmélet kereteiből. Részletes tárgyalását l. (Kun-Szász-Zsigmond, 2002). A szoftver megbízhatósági rátájának a hardverétől való lényeges eltérését illusztrálja az 1. és 2. ábra.

A matematikai modelleknek két alaptípusa van:

- Metrikák
- Prediktív statisztikai modellek

A metrikákat a NASA szoftverhiba-adatbázisa osztályozza, l. (Long, 2008). További részletes ismertetésüket tárgyalja (Porkoláb-Pataki-Sipos, 2006).

3. A szoftver mutatószáma

A szoftver-mutatószámok alkalmazásának célja elsősorban a minőség mérése, ezen túlmenően pedig a szoftverkészítés folyamatának hatékonyságvizsgálata, továbbá az előállítási folyamat további alakulásának előrebecslése, ide értve a határidőket és költségeket.

A metrika valamilyen számszerű mérési eljárással a szoftverhez rendelt függvényértéket jelent, ahol a független változók a szoftver mért adatai.

Típusai:

- Fizikai méret alapú metrikák
- Hibaszám-metrikák
- Stílus-metrikák (szemantikai elvárásokat – pl. egyértelműség – fogalmaznak meg)
- Komplexitási metrikák

A szoftver tesztelésének, utólagos karbantartásának és módosításának költsége nagyrészben a program strukturális bonyolultságának függvénye. Egy jól használható szoftver-bonyolultsági mérték már a fejlesztési fázisban felfedheti a szoftver kritikus pontjait, elősegítheti az áttekinthetőbb, módosítható és újrafelhasználható kód készítését, és becsléseket adhat a fejlesztési folyamat várható költségeire.

3.1. A méret mutatószámai

A szoftver valamilyen értelemben vett „fizikai” mérete az előállításhoz szükséges munkamennyiség legegyszerűbb jellemzője.

A program sorainak száma
Látszólagos primitívsége ellenére a gyakorlatban jól bevált. Nincs olyan másik metrika, amelyik minden tekintetben jobb lenne. Előnye, hogy általa mindenki kö4nnyen fogalmat alkothat a program bonyolultságáról. Hátránya is éppen egyszerűsége: sokan úgy gondolják, hogy ez az egyetlen, valóságos szoftvermetrika, amely minden fontos dolgot mérni tud.

A programfejlesztési pályázatkiírás oldalszáma
A felhasználói dokumentáció oldalszáma
A JACKSON-diagram csúcsainak száma
Funkcióleíró pontok száma a specifikációban
A program hosszának és dokumentációjának aránya

Különösen érdekesek a dokumentációt is figyelembe vevő mérőszámok, mert a dokumentáció készítését sok szoftverfejlesztő másodlagos fontosságúnak tekinti.

3.2. Hibaszám alapú metrikák

Működési zavarok száma

Alkalmassága különböző célokra
A minőség megítélésére: mérsékelten.
A minőség összehasonlítására más programokkal vagy korábbi állapottal: mérsékelten.
Előre meghatározott kritériumoknak való megfelelésre: igen.
A programkészítés közbeni döntések megalapozására: igen.
A jövőbeni célok kitűzésére: mérsékelten.
A fogyasztói megelégedettség mérésére: nem.

Ezer programsorra jutó működési zavarok száma

Alkalmassága különböző célokra:
A minőség megítélésére: mérsékelten.
A minőség összehasonlítására más programokkal vagy korábbi állapottal: igen.
Előre meghatározott kritériumoknak való megfelelésre: igen.
A programkészítés közbeni döntések megalapozására: nem.
A jövőbeni célok kitűzésére: igen.
A fogyasztói megelégedettség mérésére: mérsékelten.

Problémák
Különböző programnyelvek erősen különböző programméretekhez vezetnek.
A forráskódban található megjegyzés-sorok száma eltorzíthatja az eredményt.
Sok forráskód üres sorokat tartalmaz a könnyebb áttekinthetőség érdekében.

Ezer nem-megjegyzés jellegű programsorra jutó működési zavarok száma

Alkalmassága különböző célokra
A minőség megítélésére: igen.
A minőség összehasonlítására más programokkal vagy korábbi állapottal: igen.
Előre meghatározott kritériumoknak való megfelelésre: igen.
A programkészítés közbeni döntések megalapozására: nem.

A jövőbeni célok kitűzésére: igen.
A fogyasztói megelégedettség mérésére: mérsékelten.

Problémák
Külön programok vagy speciális fordítási opciók kellenek a mérőszám meghatározásához.
Csökkenti a megjegyzések beírására való készséget, ezen keresztül pedig a program átláthatóságát és karbantarthatóságát.

Egy felhasználóra jutó működési zavarok száma

Alkalmassága különböző célokra
A minőség megítélésére: nem.
A minőség összehasonlítására más programokkal vagy korábbi állapottal: mérsékelten.
Előre meghatározott kritériumoknak való megfelelésre: mérsékelten.
A programkészítés közbeni döntések megalapozására: nem.
A jövőbeni célok kitűzésére: nem.
A fogyasztói megelégedettség mérésére: igen.

Problémák
Nehéz meghatározni az adott szoftverterméket használók számát; nehéz meghatározni egy adott felhasználónál a használat mértékét.

Ezer tesztelési órára jutó működési zavarok száma

Alkalmassága különböző célokra
A minőség megítélésére: nem.
A minőség összehasonlítására más programokkal vagy korábbi állapottal: mérsékelten.
Előre meghatározott kritériumoknak való megfelelésre: igen.
A programkészítés közbeni döntések megalapozására: nem.
A jövőbeni célok kitűzésére: igen.
A fogyasztói megelégedettség mérésére: igen.

Problémák
Az eredmények erősen függenek a teszt szigorától, kiterjedtségétől és intenzitásától (egyetlen teszt sokszori megismétlése nem adja ugyanazt az eredményt, mint több teszt párhuzamos alkalmazása).

Ha a tesztfutásokat ember indítja, az eredményt befolyásolja a szakképzettség.

A tesztelés során végrehajtott programkód mennyisége függ a tesztelés módszerétől, ezért egyidejűleg olyan más mérőszámot is célszerű alkalmazni, amely megadja, hányszorosan fedi le az adott teszt a programot.

Ezer tesztelési órára jutó működési zavarok száma fedettségi mérőszámmal

Alkalmassága különböző célokra
A minőség megítélésére: igen.
A minőség összehasonlítására más programokkal vagy korábbi állapottal: igen.
Előre meghatározott kritériumoknak való megfelelésre: igen.
A programkészítés közbeni döntések megalapozására: igen.
A jövőbeni célok kitűzésére: igen.

A fogyasztói megelégedettség mérésére: igen.

3.3. Stílus-metrikák

Általános minőségi elvárásokat (pl. stabilitás, áttekinthetőség) mérnek egyszerű forráskód-jellemzőkön keresztül.

Kódsorok átlagos száma metódusonként
Metódusok átlagos száma osztályonként
Osztályok átlagos száma teljes programonként
Abszrakt osztályok aránya az összes osztályok között
A modulból kimenő kapcsolatok átlagos aránya az összes (bemenő+kimenő) kapcsolathoz képest.

3.4. Komplexitási metrikák

A komplexitási metrikák egyaránt lehetnek termékjellemzők (az elkészült szoftver statikus tulajdonságai) és folyamatjellemzők (a szoftverkészítési folyamat által felhasznált idő és munkamennyiség).

Típusai:

- Objektumorientált metrikák
- HALSTEAD-féle komplexitási metrika (információelméleti alapú)
- MCCABE-féle ciklomatikus komplexitási metrika (gráfelméleti alapú)
- HENRY-KAFURA-féle információáramlási metrika (eljárás külső kapcsolatai)

A Unix operációs rendszeren bizonyították, hogy a MCCABE- és a Halstead-komplexitás erősen korrelál egymással és a hibaszámmal is, míg a HENRY-KAFURA komplexitás statisztikailag független tőlük.

3.4.1. Objektumorientált metrikák

Az objektumorientált programtervezés általános elterjedése óta gyakran az ilyen eredetű programjellemzők alapján célszerű bonyolultsági elemzést végezni. Legfontosabb típusait összefoglalja (Porkoláb-Pataki-Sipos, 2006).

Súlyozott metódusok osztályonként
Osztály bonyolultsága: az osztályon definiált metódusok bonyolultságának az összege.
Előnye, hogy a bonyolultság a kódot felhasználó osztályokra és az azokból származtatott további osztályokra tovaterjed.

Öröklődési hierarchia mélysége
Osztály bonyolultsága: hány öröklődési szintnyi „távolságra" van a „gyökérosztálytól”.

Leszármazott osztályok száma
Osztály bonyolultsága: közvetlenÜl leszármaztatott gyerekosztályok száma.
Előnye, hogy fokozottan figyelembe veszi a kód újrafelhasználását, ami a hibaterjedés egyik fő oka.

Osztályok közöti kapcsolódás
Osztály bonyolultsága: az adott osztályhoz kapcsolódó osztályok száma.

Kapcsolódás: az egyik osztály felhasználja a másik metódusait vagy attribútumait (l. a 3.4.4. pontban a HENRY-KAFURA metrikát).

3.4.2. A HALSTEAD-féle komplexitási metrika

A bonyolultság nem triviális mérőszámai közül legismertebb a M. H. HALSTEAD által kidolgozott komplexitási metrika. Ez a forrásnyelvi kód szövegelemzésén és információelméleti meggondolásokon alapszik.

A következő alapmennyiségekre van szükség:
n_1 — a különböző operátorok (eljárások, függvények, műveletek) száma;
n_2 — a különböző operandusok száma;
N_1 — az operátorok összesített előfordulási száma;
N_2 — az operandusok összesített előfordulási száma.
Az alapmennyiségekből lehet kiszámítani a komplexitási metrikákat:

A program szókincse

Képlete: $n = n_1 + n_2$
Jelentése: a programban található különböző azonosítók száma.

A program hossza

Képlete: $N = N_1 + N_2$
Jelentése: a programban található azonosítók összes előfordulásainak együttes száma.

A program térfogata

Képlete: $V = N \log_2 n = \mathrm{ld} n^N$
Jelentése: a program kódolásához szükséges bitek száma.

A komplexitási mérőszámok nyelvészeti eredetűek. A program szókincse a programban, mint szövegben található különböző szavak száma. A program hossza a programban, mint szövegben található összes szavak száma. Mint látjuk, ez a két fogalom közel áll az elnevezés hétköznapi értelmezéséhez.

A program térfogata kissé bonyolultabb fogalom. Induljunk ki abból, hogy n különböző szó kódolásához elméletileg $\log_2(n)$ bitre van szükség. Ugyanis a szavakat ilyenkor kódolhatjuk a 0, 1, 2, 3, … n-1 számokkal, és a számokat kettes számrendszerben is felírhatjuk. Ha n éppen 2-nek egy pozitív egész kitevőjű hatványa, akkor n-1 kettes számrendszerbeli alakja pontosan $\log_2 n$ számú 1 értékű bitet tartalmaz, pl. $15 = 1111_2$. Ha pedig n 2-nek két szomszédos pozitív egész kitevőjű hatványa közé esik, akkor ugyan $\log_2 n$ nem egész szám, de n kettes számrendszerbeli alakjához pontosan ugyanannyi bitre van szükség, mint a nála éppen nagyobb 2-hatványnak, vagyis annyira, mint $\log_2 n$ felfelé egészre kerekített értéke. A kódoláshoz szükséges bitek számát a szókincs információtartalmának tekinthetjük. (A $\log_2 n$ értéket egyébként az n elemszámú halmaz HARTLEY-információjának nevezzük.) Vagyis a program térfogata a programban szereplő azonosítók összes előfordulási száma szorozva az egy azonosító kódolásához szükséges bitek számával, tehát éppen a programnak, mint szövegnek a kódolásához szükséges bitek száma. Ezt a mennyiséget a program bonyolultságának, komplexitásának mérésére használjuk. A fogalom azért célszerű, mert egyaránt függ a program fizikai méretétől és a benne előforduló szókincs összetettségétől.

A HALSTEAD-mérőszámok alkalmazása bizonyos esetekben problémákat vet fel.

- Vannak olyan programozási eszközök, amelyek nem tekinthetők sem operátornak, sem operandusnak. Ilyenek például a címkék vagy a ciklusutasítások.

- Egy azonosító többféle operátort vagy operandust is jelölhet a program különböző pontjain. Így például egy változónév a magas szintű programozási nyelvekben több modulban is előfordulhat, mint a modul lokális változója, továbbá a + jel egyaránt jelenthet aritmetikai vagy logikai összeadást és előjelet, tehát három teljesen különböző dolgot.
- A komplexitási metrika bünteti a programozási nyelvbe eleve beépített látszólagos redundanciát. Például a másodfokú egyenlet megoldóképletének algebrai jelölésmódja a következő alakú: $(-b+\text{sqrt}(b^2-4*a*c)/(2*a)$. Ugyanez a formula a szintén gyakran alkalmazott fordított lengyel jelölésrendszerben: $/+-b\text{sqrt}-^2b*4*ac*2a$. Az algebrai jelölésmód a zárójelezés miatt nagyobb programhosszt von maga után. A legtöbb programozási nyelvben mégis ezt használják, mert az emberi gondolkodásmód miatt kevésbé bonyolultnak tűnik.

A HALSTEAD-metrika használata komoly szakmai vitákat váltott ki [BENMEL]. Bírálói is elismerik azonban, hogy a tervezés szakaszában jó előrejelzést adhat a program bonyolultsága miatt várható hibák számáról.

3.4.3. A MCCABE-féle ciklomatikus komplexitási mutatószám

Rajzoljuk fel egy program G vezérlésfolyam-gráfját a következő módon: a gráf csúcsai az utasítás-szekvenciáknak (elágazás nélküli utasítás-sorozatoknak) felelnek meg, élei pedig a program által a szekvenciák között létrehozott lehetséges közvetlen átmeneteknek. Ekkor felírhatjuk az úgynevezett ciklomatikus mutatószámot.

Képlete: $v(G) = e - n + 2$

ahol

v (nű) – a ciklomatikus komplexitás

e – a gráf éleinek száma

n – a gráf csúcsainak száma

MCCABE javasolta a $v(G)$ számot a program komplexitásának mérésére, ily módon iránymutatónak a program fejlesztéséhez és teszteléséhez. A tesztelési munka tervezésére azért alkalmas, mert bizonyítható, hogy $v(G)$ pontosan egyenlő a vezérlésfolyam-gráf független (legalább egy eltérő élt tartalmazó) útjainak maximális számával, vagyis a tesztelendő útvonalak maximális számával (hiszen ennél több tesztelési útvonalat megadva biztosan találnánk köztük két azonos útvonalat).

Strukturált programok esetében nincs szükség a program gráfjára sem, hiszen ebben az esetben könnyű meghatározni az elágazási pontok számát.

A ciklomatikus komplexitást a szoftver kockázati szintjének jellemzésére is használják. Az általános ajánlás szerint a 10-nél alacsonyabb mutatószám-értékre kell törekedni.

Ciklomatikus komplexitás	Kockázati szint
1-10	Egyszerű program, kevés kockázat
11-20	Mérsékelten bonyolult program, mérsékelt kockázat
21-50	Bonyolult program, magas kockázat
Több mint 50	Nem tesztelhető program, nagyon magas kockázat

A MCCABE mutatószám problémái:

- Nem kezeli külön az egyszerű (nem többfelé ágaztató) feltételes utasításokat.
- Nem veszi figyelembe a program deklarációs részét.

Vannak a ciklomatikus komplexitási mutatószámnak olyan módosításai, amelyek kiküszöbölik a fenti problémákat.

3.4.4. A HENRY-KAFURA-féle információáramlási metrika

HENRY és KAFURA a programmodulok közötti információáramlás alapján definiált egy bonyolultsági mértéket. A programmodulba bemenő és az onnan kimenő adatutakat és vezérlésátadásokat számolja össze.

Más néven fan-in - fan-out elemzést tartalmaz.

Fan-in: az adott modult hívó modulok száma.

Fan-out: az adott modul által hívott más modulok száma.

Ezek után a HENRY-KAFURA mérték:

$$\text{Komplexitás} = \text{Programhossz} \times (\text{Fan-in x Fan-out})^2$$

Ez egyetlen modulra vonatkozik, összegzéssel állapítható meg a teljes program komplexitása.

Előnye, hogy már a tervezés közben is jól használható.

Hátránya, hogy kizárólag a programmodulok külvilággal folytatott kommunikációját veszi figyelembe, ami a valóságos bonyolultságnak csak egyik tényezője.

4. Folyamat-mutatószámok

4.1. A Walston-Felix folyamat-mutatószámok

C. E. WALSTON és C. P. FELIX, az IBM kutatói 60 nagy szoftverprojekt különböző adatainak statisztikai elemzése alapján alkották matematikai modelljüket a szoftverkészítés folyamatának jellemzésére.

Munkamennyiség

Képlete: $E = 5{,}2\, L^{0{,}91}$

ahol

E – a program létrehozásához szükséges munkamennyiség, ember-hónapokban;

L – a program forráskódjának hossza 1000 programsor egységben.

Látható, hogy E értéke nem egyenesen arányos a programsorok számával, hanem annál kissé lassabban növekszik.

Termelékenység

Képlete: $P = \dfrac{L}{E} = \dfrac{L}{5{,}2\, L^{0{,}91}} = 0{,}19\, L^{0{,}09}$

ahol:

P – egy ember által havonta átlagosan létrehozott programkód hossza, ezer sorban kifejezve.

Az előbbi megjegyzést folytatva, itt még világosabban látszik, hogy a program hosszának növekedésével párhuzamosan, bár igen enyhén, de nő a havonta átlagosan előállítható sorok száma. Ez eléggé természetes, hiszen a rutinszerű, méretarányos programkód-írási munkát megelőzi egy kevésbé rutinszerű, tehát a mérettől kevésbé függő tervezési tevékenység.

A formula nem tükrözi az egyes programok közti nehézségi különbségeket, amelyek növelhetik, vagy csökkenthetik a termelékenységet. Ennek a hiányosságnak a kiküszöbölésére a szerzők később bevezettek egy termelékenységi szorzót, amelynek értéke a program jellegétől függ, és ugyancsak korábbi projektek adataiból becsülhető.

4.2. A Halstead-féle folyamat-mutatószámok

A HALSTEAD-féle komplexitás-fogalomra alapozva folyamat-mutatószámokat definiálhatunk.

A program nehézsége

Képlete: $D = \frac{n_1}{2}\frac{N_2}{n_2}$

Látható, hogy a program nehézsége egyenesen arányos a programozó által vagy automatikusan elvégzett operátor-deklarációk számával és egy operandus átlagos előfordulási számával.

Munkamennyiség

Képlete: $E = VD$

ahol

E – a program létrehozásához szükséges munkamennyiség, ember-hónapokban;
V –a HALSTEAD-féle komplexitási mutatószámoknál megismert programtérfogat;
D – a program nehézsége.

A mutató előzetes kiértékelésekor a specifikáció ismeretében jól becsülhető a különböző operátorok és operandusok száma: n_1 és n_2, a fejlesztők programozási stílusa alapján következtetni lehet az operandusok átlagos előfordulási számára is: N_2/n_2, de a program hossza ekkor még nem ismert. HALSTEAD rámutatott arra, hogy közelítőleg fennáll:

$$N \approx n_1 \log_2 n_1 + n_2 \log_2 n_2$$

Ennek alapján lehetséges egy viszonylag egyszerűen kiszámítható közelítő formula alkalmazása.

Képlete: $E = \frac{n_1 N_2 (n_1 \log_2 n_1 + n_2 \log_2 n_2) \log_2 n}{2 n_2}$

ahol

n_1 — a különböző operátorok (eljárások, függvények, műveletek) száma;
n_2 — a különböző operandusok száma
$n = n_1 + n_2$

E értékét itt a programkódolás úgynevezett elemi döntéseiben mérjük. Egy elemi döntés lényegében a programkód adott helyén szükséges szimbólum kiválasztását jelenti. Ennek időigénye is kiszámítható.

Elemi döntés időigénye

Képlete: $T = \frac{E}{18}$

ahol:

T – egy elemi döntés meghozatalához szükséges, másodpercben mért idő.

Amennyiben csak a HALSTEAD-mutatószámoknál szereplő programhossz ismert, itt is lehetséges némileg egyszerűbben kiszámítható közelítő formula alkalmazása.

Képlete: $T = \frac{N^2 \log_2 n}{72}$

ahol:

N –a programban található azonosítók összes előfordulásainak együttes száma;
n – az $N = n \log_2 (n/2)$ egyenlet (csak közelítően kiszámítható) megoldása.

5. Prediktív statisztikai modellek

A legfontosabb alapmodellek már évtizedekkel ezelőtt megszülettek, l. (Mellor-Bendell, 1986).

5.1. Duane modellje

A modell jellemzői:

- futási időre vonatkozik
- determinisztikus
- nem számlálja a hibákat
- csak a meghibásodásokkal foglalkozik, a javítással nem
- a programot fekete doboznak tekinti
- folyamatosan érkező és összesített adatokkal is tud számolni

A modell alapötlete:

- a komplex műszaki rendszerek bizonyos mértékig a szoftverhez hasonlóan kezelhetők
- az előforduló hibákat teljesen ki lehet javítani
- az egyszer már előfordult hibákat a felhasználó azonos formában nem ismétli meg

A modell koncepciója

Úgynevezett „tanulógörbe”, amely a használat során bővülő ismereteket figyelembe veszi az előrejelzésben

Jelölések

U – összes figyelembe vett programfutási idő
$c(u)$ – u idő elteltéig jelentkezett hibák száma
$q(u)$ – u idő elteltekor érvényes meghibásodási ráta
a, b – skálaparaméterek

Matematikai feltevések

A tapasztalatok szerint az $(u,q(u))$ koordinátájú pontok logaritmikus beosztású papíron egy negatív irányszögű egyenes közelében helyezkednek el, vagyis logaritmusaik kapcsolata lineáris regresszióval jól leírható:

$$\log[q(u)] = (b-1)\cdot\log(u) + \log(a)$$

ahol $b < 1$; innen

$$q(u) \approx a \cdot u^{b-1}$$

Ugyancsak a tapasztalatok szerint

$$q(u) \approx \frac{c(u)}{u}$$

tehát az előfordult hibák számának növekedésével így alakul a $(0,u)$ időintervallumban bekövetkező meghibásodások számának várható értéke:

$$c(u) \approx a \cdot u^{b}$$

A modell értékelése

A programfutási idővel és az újonnan előkerülő hibákkal számolva a modell előrejelzési képessége igen jó; rosszul használható abban az esetben, ha a kezdeti adatok erősen eltérnek a feltevésektől, mert ezekre nagyon érzékeny a modell.

5.2. Jelinski és moranda modellje

A modell jellemzői

- naptári idő
- sztochasztikus
- a folyamatosan érkező adatokat is fel tudja dolgozni
- gyakoriságokkal számol
- fekete doboz
- a hibák száma véges
- a hibák csak akkor számítanak, ha megjelennek
- a javítás tökéletes
- a meghibásodási ráta csak a hibák megjelenésével változik

Jelölések

T – az első programfutástól számított naptári idő
t – a legutóbbi hibától számított naptári idő
t_i – az i-1 és i sorszámú meghibásodások közti naptári idő
n – a programban található összes hibák száma
c – az adott időpontig megtalált és kijavított hibák száma
$q(c)$ – c számú hiba után érvényes meghibásodási ráta
z – skálaparaméter
Σ – összegzés 1-től c-ig

Matematikai modell

Feltételezzük, hogy

$$q(c) = z \cdot (n - c)$$

vagyis a következő időegység alatt felbukkanó hibák számának várható értéke a programban maradt hibák számával arányos. Bizonyítható, hogy n optimális becslését kapjuk a következő egyenletből:

$$\sum \frac{1}{n-i} = \frac{c \cdot T}{n \cdot T - \sum i \cdot t_i}$$

és ennek felhasználásával becsülhető z is:

$$z = \frac{c}{n \cdot T - \sum i \cdot t_i}$$

A modell értékelése

Eredeti formájában kevéssé vált be a gyakorlatban. Kiindulópontja több későbbi, sikeres modellnek.

5.3. Musa modellje

A modell jellemzői

- programfutási idő
- sztochasztikus
- a folyamatosan érkező adatokat is fel tudja dolgozni
- gyakoriságokkal számol
- fekete doboz
- a hibák száma véges

- a hibák csak akkor számítanak, ha megjelennek
- a javítás tökéletes
- a meghibásodási ráta csak a hibák megjelenésével változik

Jelölések

t – a legutóbbi hibától számított programfutási idő
n – a programban található összes hibák száma
$c(t)$ – az adott időpontig megtalált és kijavított hibák száma
$y(t)$– t futási idő elteltekor a következő meghibásodásig hátralevő idő várható értéke (a megbízhatóságelméletben szokásos jelölése: MTTF)
$m(t)$ – $c(t)$ várható értéke
C – tesztkompressziós faktor; azt fejezi ki, hogy a tesztelés során hányszor gyorsabban akadunk hibára, mint normál programfutás során

Matematikai modell

A t futási összidőig megtalált hibák számának várható értéke:

$$m(t) = n \cdot \left\{ 1 - \exp\left(-\frac{C}{y(0) \cdot n} \cdot t \right) \right\}$$

ahol $\exp(x) = e^x$, tehát az e alapú exponenciális függvény. Látható, hogy a jobboldalon a kapcsos zárójelen belüli rész t növekedésével 1-hez tart, így $m(t)$ is tart n-hez. A legközelebbi meghibásodásig eltelő idő várható értéke (MTTF):

$$y(t) = y(0) \cdot \exp\left(\frac{C}{y(0) \cdot n} \cdot t \right)$$

A jobboldalon az exponenciális részben t együtthatója pozitív, ezért a jobboldal végtelenhez tart. Tehát sok programfutás után a legközelebbi meghibásodásig eltelő idő minden határon túl nő. A megbízhatósági függvény:

$$R(t) = \exp\left(-\frac{t}{y(t)} \right)$$

Ennek alapján meghatározható, hogy ha a legközelebbi meghibásodásig várhatóan eltelő időt y_1-ről y_2-re akarjuk növelni, ehhez hány hibát kell megtalálnunk és kijavítanunk:

$$\Delta_c = (y(0) \cdot n) \cdot \left\{ \frac{1}{y_1} - \frac{1}{y_2} \right\}$$

Ugyancsak meghatározható, hogy ehhez mekkora programfutási időre van szükség:

$$\Delta_t = \frac{y(0) \cdot n}{C} \cdot \ln\left(\frac{y_2}{y_1} \right)$$

A modell értékelése

A programfutási idő alkalmazásával a gyakorlatban is bevált. Figyelembe veszi, hogy különbség van a tesztelési és felhasználási célú programfuttatás között. Lehetőséget ad a továbbfejlesztésre, új paraméterek bevezetésével jobb illeszkedés elérésére. Hátránya, hogy nem tesz különbséget az egyes hibák veszélyessége között.

Példa

Egy nagy program feltételezhetően kb. 300 hibát tartalmaz. A program indításától az első hiba megjelenéséig átlagosan eltelő idő (MTTF) 1.5 óra. A tesztkompressziós faktor becsült értéke 4.

Mennyi ideig kell tesztelni a programot ahhoz, hogy a megmaradó hibák számát 300-ról 10-re csökkenthessük?

Megoldás:

$$\Delta_c = (y(0) \cdot n) \cdot \left\{ \frac{1}{y_1} - \frac{1}{y_2} \right\}$$

Behelyettesítve:

$$300 - 10 = (300 \cdot 1{,}5) \cdot \left\{ \frac{1}{1{,}5} - \frac{1}{y_2} \right\}$$

Innen

$$y_2 = 45 \text{ óra}$$

Továbbá

$$\Delta_t = \frac{y(0) \cdot n}{C} \cdot \ln\left(\frac{y_2}{y_1} \right)$$

Behelyettesítve:

$$\Delta_t = \frac{300 \cdot 1{,}5}{4} \cdot \ln\left(\frac{45}{1{,}5} \right) = 382{,}6$$

Innen

$$\Delta_t = 382{,}6 \text{ óra}$$

Tehát ennyi tesztelési időre van szükség, hogy a programhibák száma 10-re csökkenjen.

Végül

$$R(t) = \exp\left(-\frac{t}{y(t)} \right)$$

Behelyettesítve:

$$R(50) = \exp\left(-\frac{50}{45} \right) = 0{,}33$$

Vagyis a 382.6 órányi tesztelés elvégzése után legalább 50 óra hibamentes működés valószínűsége 0,33 lesz.

5.4. Goel és Okumoto modellje

A modell jellemzői

- programfutási időt alkalmaz
- sztochasztikus
- gyakoriságokkal számol
- fekete doboz
- a hibák száma véges, de véletlen
- a hibák csak akkor számítanak, ha megjelennek
- a javítás nem feltétlenül tökéletes
- a meghibásodási ráta csak a hibák megjelenésével változik

Jelölések

t – a legutóbbi hibától számított programfutási idő
n – a programban található összes hibák számának várható értéke
$c(t)$ – az adott időpontig megtalált hibák száma
$m(t) – c(t)$ várható értéke
z – arányossági tényező a meglevő és megmutatkozó hibák között

Matematikai modell

Az időegység alatt megmutatkozó hibák átlagos száma egyenesen arányos a programban még benn található hibák átlagos számával.

A t futási összidőig megtalált hibák számának várható értéke:

$$m(t) = n \cdot \{1 - \exp(-z \cdot t)\}$$

Látható, hogy a jobboldalon a kapcsos zárójelen belüli rész t növekedésével 1-hez tart, így $m(t)$ is tart n-hez.

Feltételezzük, hogy $c(t)$ Poisson-eloszlást követ, paramétere és várható értéke $m(t)$.

Értékelés

Goel és Okumoto modellje egyaránt közel áll Musa, valamint Jelinski és Moranda modelljéhez. Egyetlen új paraméter bevezetésével modellezhető a nem tökéletes javítás. Legyen p annak a valószínűsége, hogy sikeres a javítás. z helyett pz paramétert használhatunk.

Hivatkozások

Kun, I., Szász, G., Zsigmond, Gy. (2002): *Minőség és megbízhatóság I.-II.*, LSI, Budapest.

Long, J. (ed.) (2008): *Metrics Data Program*, National Aeronautics and Space Administration, http://mdp.ivv.nasa.gov/index.html

Mellor, P., Bendell, A. (eds) (1986): *Software Reliability. State of the Art Report*, Pergamon Infotech Ltd, Oxford-New York-Toronto.

Pan, Jiantao (1999): *Software Reliability*, http://www.ece.cmu.edu/~koopman/des_s99/sw_reliability/

Porkoláb, Z., Pataki, N., Sipos, Á. (2006): *Jelenleg használatos szoftvermetrikák összehasonlító elemzése*, Tanulmány, GVOP-3.2.2.-2004-07-0005/3.0, Budapest.