Szolnoki Tudományos Közlemények XVI.
Szolnok, 2012

Schuster György[1] – Terpecz Gábor[2]

KRITIKUS SIKERTÉNYEZŐ VAGY ELKERÜLHETETLEN VESZÉLYFORRÁS[3]

A szoftver mindennapi életünk meghatározója eszközeink nagy része rendelkezik valamilyen működtető programmal, amiről sokszor tudomásunk sincs és természetesnek vesszük ezen eszközök specifikus működését és hibátlan működését. Azonban ezek a programok - még a viszonylag egyszerűek is - bonyolultak, illetőleg létrejöttük során bonyolult műveletsoron mennek keresztül. Kérdés, hogy milyen mértékben bízhatunk szoftvereinkben és hogyan tesztelhetjük ezeket. Ez az előadás ezzel a kérdéssel foglakozik.

CRITICAL SUCCES FACTOR OR INEVITABLE RISK SOURCE

Software is definitive part of everyday life. Most devices, vehicles and even small facilities have some kind of working programmes, above which we don't have any knowledge and we consider natural hose devices specific and faultless working. However these programmes – even relative simplest – are complicated or their production procedure is quite complex. The questions are up to which extent we can trust in our software or how we can test them to avoid problems. Our paper deals with these questions.

BEVEZETÉS

1991 február 5.-e éjszaka egy Patriot rakéta elvét egy iraki Scud rakétát, amely egy katonai körletet talál el. A szomorú eredmény 29 halott és 98 sebesült. Az ok szoftver hiba. [1]

1996 június 4-én az Európai Űr Ügynökség által indított Ariane 5 rakéta az indítás utáni 37 másodpercben élesen megváltoztatta pályáját minek következtében a fedélzeti önmegsemmisítő mechanizmus működésbe lépett. A vizsgálat megállapította, hogy a meghibásodás oka szoftverhiba. [2]

2008. október 7-én a Quantas QF72-es Airbus A330/300 járatának fedélzeti rendszere nyugat ausztráliai standard idő szerint 12:40:28-kor 37000 láb magasan hibát jelzett, majd a robotpilóta automatikusan kikapcsolt. [3] Kézi magasság korrekció után a robotot újra üzembe helyezték. 12:42:27-kor a gép élesen – mintegy –8,4 fokos süllyedésbe ment át –0,8g túlterheléssel és süllyedt 650 lábat, majd 1,3 g-s terheléssel visszaállt a kívánt állásszögre.

Ez az esemény megismétlődött kissé enyhébben 12:45:08-kor. Az eredmény 51 sérült a fedélzeten. A repülőgép biztonságban leszállt.

[1] PhD, egyetemi docens, OE KVK MAI, schuster.gyorgy@kvk.uni-obuda.hu
[2] OE KVK MAI, terpecz.gabor@kvk.uni-obuda.hu
[3] Lektorálta: Prof. Dr. Szabolcsi Róbert, egyetemi tanár, Nemzeti Közszolgálati Egyetem Katonai Repülő Tanszék, szabolcsi.robert@uni-nke.hu

Az esemény oka egy navigációs berendezés meghibásodása és egy szoftver hiba. És folytathatnánk a sort, ha nem is a végtelenségig, de igen hosszan. Felmerül a természetes kérdés, hogy hogyan történhetett meg, hogy olyan szoftverek kerültek ki a fejlesztők keze alól, amelyek emberéletet követeltek, súlyos sérüléseket okoztak, illetve komoly anyagi károkkal járó meghibásodást eredményeztek.

A fejlesztők talán felelőtlenek voltak, vagy esetleg nem értettek a szoftveríráshoz? A válasz egyértelműen nem.

Ez a cikk arra szeretne rávilágítani, hogy a szoftver fejlesztők milyen kihívásokkal találkoznak, milyen fejlesztési elveket alkalmaznak és milyen módon próbálják a szoftverek helyességét tesztelni.

ÉLETCIKLUS

A közhiedelemmel ellentétben a szoftverfejlesztés nem a program megírását jelenti. A fejlesztés egy folyamat, amelynek jól elkülöníthető fázisai vannak.

Az irodalom több életciklus leírást ismer, ebben a cikkben a Royce-féle modell német hadsereg által kibővített változatát tekintjük érvényesnek.

Az életciklus modell lépései:

1. kockázatelemzés és követelmények. Ebben a fázisban tisztázni kell a pontos követelményeket, elkészíteni a specifikációt és azt jóváhagyatni a „megrendelővel”;
2. rendszer tervezés (logikai rendszerterv). Az előző lépésben elkészült specifikáció alapján rendszer szintű tervezés. Ekkor készül el az összefoglaló – áttekintő rendszerterv. Amennyiben a „megrendelő” alkalmas arra a rendszertervet célszerű a „fejlesztőnek” jóváhagyatni;
3. program tervezés (fizikai rendszerterv). Ebben a fázisban a szoftver részletes tervezése folyik. A „fejlesztő” itt definiálja a szoftver elemeket (unit), tervezi meg az adatszerkezeteket és a szükséges algoritmusokat, itt definiálja az interfészeket, végül a rendszer integrációs folyamatát;
4. kódolás. A program fizikai megírása ebben a fázisban történik a fizikai tervek alapján. Ez a fázis általában az életciklusnak 20–25%-át teszi csak ki;
5. tesztelés. Az elkészült szoftver elemeket, ezek kapcsolatát és a teljes szoftver tesztelését ebben a fázisban végzi a „fejlesztő” és ha „megrendelő” erre alkalmas, akkor ő is. A szoftver tesztelést egy további fejezetben részletesen tárgyaljuk;
6. átadás. A specifikációnak megfelelően pontról – pontra, teszt eredményekkel alátámasztva a „fejlesztő” átadja a „megrendelőnek” az elkészült szoftvert;
7. nyomkövetés, karbantartás. A „fejlesztő” későbbi szoftver projektjei során felhasználja az előzőekben megírt program elemeket. Ennek során előfordulhat, hogy valamilyen hibát, vagy eltérést tapasztal, illetve a „megrendelő” jelezhet problémát (ez persze lehet garanciális probléma is), akkor a „fejlesztő” a megfelelő javításokat elvégzi és implementálja.

Ide tartozik két gondolat: ez nem jelenti azt, hogy a „fejlesztő” köteles a „felhasználó” specifiká-

ción kívüli kéréseit, követeléseit megvalósítani.

A másik gondolat, hogy a „javítást” nagyon alaposan tesztelni kell, mert egy kismértékben hibás szoftver elem is jobb, mint egy működésképtelen.

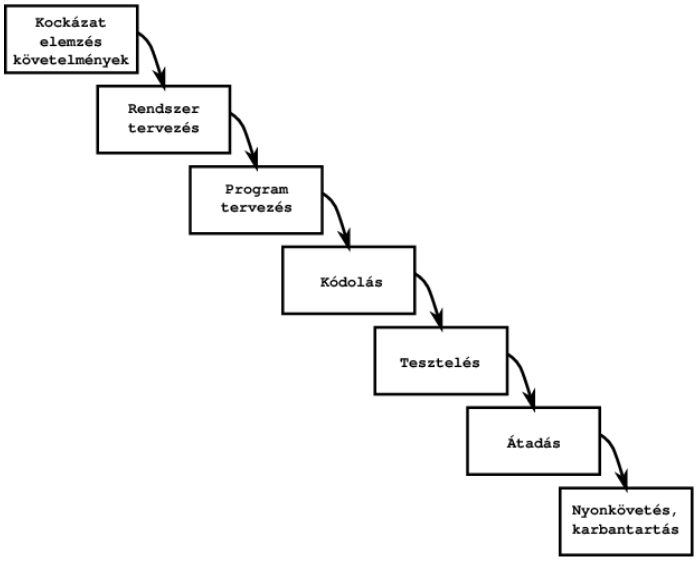
Az életciklus fázisait be kell tartani, nem kerülhetők meg. Minden egyes fázist megfelelő módon dokumentálni kell.

Életciklus modellek

Az életciklus fázisai alapján úgynevezett életciklus modelleket hoztak létre, amelyek alapján a szoftver fejlesztése történik.

Az első modell az úgynevezett „csináljuk míg nem megy” (do until done). Ez a modell a biztonságkritikus szoftver fejlesztésben szóba sem kerül, csak a rend kedvéért említjük meg.

A következő a „vízesés” modell. Jellemzője az, hogy az életciklus fázisait módszeresen lépésről – lépésre hajtják végre. Visszalépésre nincs lehetőség.

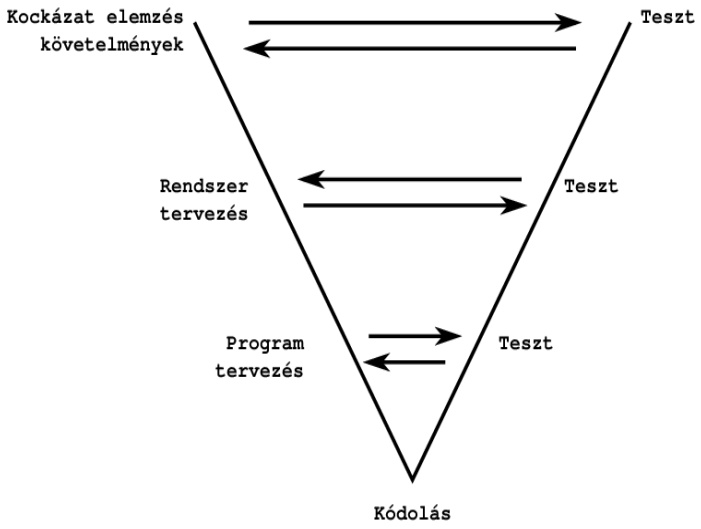


1. ábra A vízesés modell

A vízesés modell

A módszer előnye, hogy nagyon módszeres, ezért nagyon egyszerű nyomon követni és alapvetően jól kezelhető kevésbé gyakorlott szakemberek számára. Kisebb szoftver projekteknél jól használható.

Hátránya, hogy nincs visszalépési lehetőség. Így ha a tervezés során valamilyen hibát vétettünk ez a hiba az egész fejlesztésen végigvonul. Továbbá hajlamossá tesz a túldokumentálásra.



2. ábra V modell

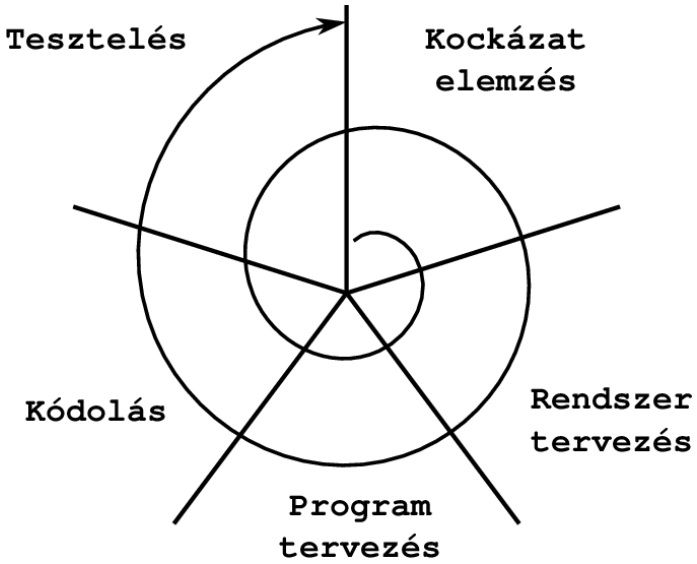
A V modell

1992-ben a német védelmi minisztérium fejlesztette ki és szabványosította a német hadsereg számára a V életciklus modellt. A modell minden tervezési lépésére létezik – vele szemben – egy tesztelési fázis. A modell rendkívül nagy hangsúlyt fektet az adott fázis tesztelésre.

Célszerű, hogy az adott fázist tesztelő személy, vagy csoport független legyen a fejlesztő személytől, vagy csoporttól. Ha ez teljesül, akkor csökken a szubjektív befolyásolás lehetősége.

A Spirál modell

A szekvenciális modellek közül az első a spirál modell. Ez a modell ciklikusan ismétli az életciklus fázisait folyamatosan javítva a szoftver minőségét, illetve lehetővé teszi a változó követelményekhez való alkalmazkodást.



2. ábra: Spirál modell

Előnye az, hogy az esetlegesen felmerülő tervezési problémákat ciklusról ciklusra javítani lehet. Hátránya az, hogy egyedi fejlesztéseknél túl hosszú időt vesz igénybe, a dokumentációs igény nagy. Néhány szakember szerint a spirál modell nem más, mint a vízesés modell ciklikus ismétlése.

Az inkrementális fejlesztés

Szintén szekvenciális módszer – a rendszer kis elemeit lépésről lépésre építi fel. Ezzel a módszerrel a rendszer fokozatosan épül fel, ezáltal a unit és integrációs tesztek (látszólag) könnyebben elvégezhetők. Kisebb rendszerek esetén nagyon jó módszer.

Az evolúciós modell

Párhuzamosan több, egymással versengő szoftverelemeket készítenek a kezdeti fázisokban, ezeket folyamatosan összehasonítják, a legjobbakat fejlesztik tovább.

A módszer előnye, hogy a tervezés várhatóan jó lesz és ezáltal az életciklus későbbi lépései rendkívüli módon felgyorsulnak. Hátránya, hogy erőforrás igényes és a tervezési fázis hosszadalmas.

Tesztelés

Az életciklus egyik kérdéses fázisa. Nyugodtan tételszerűen kijelenthetjük, hogy gyakorlatilag egy szoftvert teljesen nem lehet tesztelni. Ennek bizonyítására vegyük a következő példát!

Adott egy program, amely három 16 bites egész számot összead. Ekkor az eredmény három bájton fér el és 2^{48}-on (281 474 976 710 656) teszt esetünk van.

Ha feltételezzük, hogy másodpercenként egymillió teszt esetet vizsgálunk meg, akkor jó közelítéssel 9 év alatt végeznénk a kimerítő teszttel. De arról sem feledkezzünk meg, hogy 786432 terabájt méretű teszt adatbázisra lenne szükségünk. És ez csak három short int összeadása volt.

Mit lehet hát tenni, ez a kérdés. Erre találták ki a tesztelési módszertanokat, eljárásokat és eszközöket.

Tesztelési módszerek

A tesztelés két alapvető módja a statikus és a dinamikus tesztelés. Mindkét módszer alkalmazáshoz az adott program, vagy programrészlet kódjának rendelkezésre kell állni. Definiáljuk ezeket a módszereket.

- **Statikus tesztelés:** a tesztelő csoport nem futtatja a kódot, csak és kizárólag különböző szempontok szerint elemzi.
- **Dinamikus tesztelés:** a tesztelő csoport a kódot adott beállítások mellett futtatja és a kapott eredményeket hasonlítja össze az elvárt eredményekkel.

A két tesztelési módszer nem helyettesíti egymást és egyik sem végleges megoldás. Együttes alkalmazásuk segíthet a problémák felderítésében és a hibák kiküszöbölésében. Szerencsére manapság mindkét módszerhez rendelkezésre állnak segédeszközök.

Unit teszt

A program megírása közben az első elemek a unitok. Unit lehet az értelmezéstől függően egy

szubrutin, vagy függvény, de lehet egy komplett modul is. Mivel a unit a specifikáció és a tervek alapján készült, már tesztelhető.

A tesztelés főbb lépéseit egy viszonylag egyszerű C függvény segítségével mutatjuk be, amelybe szándékosan egy hibát rejtettünk el.

A függvény feladata, hogy ASCII karakterként megadott bemeneti hexadecimális értékből egy int típusú változót adjon be, ha a bemeneti érték tartományon kívüli a visszatérési érték legyen –1.

```
1.   int ascint(char c)
2.      {
3.      int i;
4.      if(c>='0' && c<='9') i=c-'0';
5.      else
6.         {
7.         if(c>='A' && c<='F') i=c-'A'+10;
8.         else
9.            {
10.           if(c>='a' && c<='f') i=c-'a'+10;
12.           }
13.        }
14.     return i;
15.     }
```

A fenti programlista szándékosan hibás.

Statikus tesztelési lépések

Kód felülvizsgálat (code review): a programegység elkészülte után a forráslistát egy meghatározott összetételű csoport vizsgálja át. A csoport összetétele:

- moderátor: aki a felülvizsgálati folyamatot levezeti;
- a program írója, vagy írói;
- prezentáló: az a személy, amely a programlistát ismerteti (lehetőleg ne legyen a program írója);
- jegyző;
- felülvizsgáló: az a személy, aki a kód – specifikáció megfelelőségére figyel;
- megfigyelő: szerepe az, hogy az esetleges figyelmen kívül hagyott szempontokra és hibákra felhívja a figyelmet.

A felülvizsgálathoz a programegységnek a következő követelményeket kell teljesítenie:

- a kódnak el kell készülnie (az adott szintig, ez néha egy – függvényt is jelenthet),
- az adott kódnak legalább részben működnie kell,
- a kód olvasható legyen (geometriai tagolás, kommentezés és kitöltött fejlécek),
- kéznél legyenek a követelmények és specifikációk.

A kód felülvizsgálat lépései

1. kód áttekintése, ahol a szerző prezentálja a megoldást a specifikáció és a követelmények

figyelembevételével;
2. a prezentáló olvassa a kódot;
3. a jegyző rögzíti a felmerülő kérdéseket, problémákat és megoldásokat;
4. a moderátor biztosítja, hogy minden lépés rögzítésre kerüljön.

A felülvizsgálat eredményeként felmerülő problémák egy változtatási követelmény listába kerülnek, amely: a megjegyzések listáját, a változtatások prioritási sorrendjét, a felelős személy megnevezését és a határidőt rögzíti.

Az átdolgozás után el kell készíteni: az új programlistát, a változtatások listáját. Ez alapján új felülvizsgálat indítható.

Felhasználás – hozzárendelés párok módszer: valahol a programban egy változót felhasználunk (például egy aritmetikai kifejezésben) ekkor a programot visszafelé olvasva minden egyes elágazást visszakövetve megvizsgáljuk, hogy a változó kapott-e szabályosan értéket.

Példa: az előzőekben ismertetett programlista alapján az **i** változó a 14. sorban kerül felhasználásra, először értéket kap 10. sorban, ha a **c** bemeneti a kisbetű tartományban van, ez akkor következik be, ha a **c** nem volt a nagybetű tartományban (8. sor else).

A második értékadás a 7. sorban történik, mikor a **c** változó a nagybetű tartományban van, ez akkor következik be, ha a **c** nem a szám tartományban van (5. sor else).

A harmadik értékadás a 4. sorban történik, ahol a c változó a szám tartományban van.

Kérdések:

- maradt-e lehetséges bemeneti érték, vagyis lefedtük-e a **c** változó teljes értelmezési tartományát (ez a kérdés már előremutat a tartomány tesztelésre)?

A válasz nem.

- Kap-e ekkor értéket az **i** változó?

A válasz nem.

- Mi lesz ekkor **i** értéke?
- A válasz nem definiált.

Következtetés:ez egy hiba!

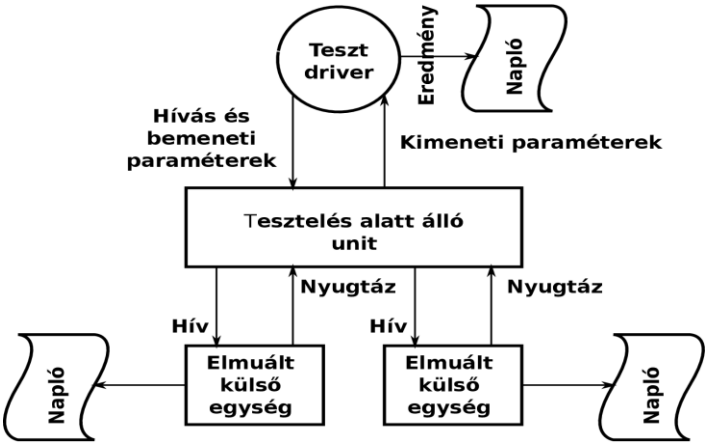
Javítás: `11.        else i=-1;`

Az újbóli felülvizsgálat nem talált hibát.

Dinamikus tesztelés

A tesztelendő program egység dinamikus tesztelése során a tesztelő személy, vagy csoport a kódot különböző szempontok szerint futtatja. Mivel a unit egyedülálló programrészlet gondoskodni kell egy tesztelési környezetről. [4]

A dinamikus tesztelésnél a program bemeneti adatait a többféleképpen adhatjuk meg. A legelterjedtebb módszerek a következők:



3. ábra Teszt környezet programelemek teszteléséhez

Vezérlési út tesztelés

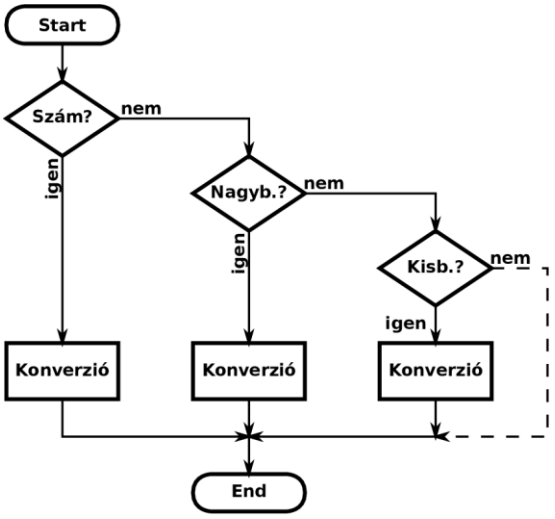
Alapötlete az, hogy a unit az adatokon műveleteket végez, a műveletek végrehajtási útja az adatoktól függ.

Ez az út a folyamatábra alapján lerajzolható. Cél annak az ellenőrzése, hogy az adathoz (adatokhoz, vagy bemeneti vektorhoz) tartozó út helyes-e.

Bár ezt a módszert dinamikus tesztekhez soroltuk, de a felkészülés során előre célszerű statikus módszerekkel a várható utakat megrajzolni.

A statikus tesztelési szakasz alatt kiderülhetnek az adatfolyam anomáliák. A dinamikus részben az előre meghatározott adatokkal teszteljük a programrészletet.

Az 5. ábrán látható a mintapéldánk vezérlési út diagrammja. A szaggatott vonal hiba helyét mutatja. Ebben – a nagyon egyszerű – esetben az anomália már a rajzoláskor látható. A példa csak egész jellegű változókkal dolgozik és a számítási algoritmus is jó kézben tartható.



4. ábra A mintapélda folyamatábrája

Dinamikus tesztelés esetén erre a vezérlési útra kell felépíteni a bemeneti teszt adatokat, az úgynevezett teszt vektorokat.

Felmerül a természetes kérdés, hogy ha az anomália ilyen szépen kiderült miért van szükség mégis a dinamikus tesztre? Az ne felejtsük el, hogy itt egy (szándékos) kódolási hibát vétetettünk, de előfordul olyan eset is amikor a kód helyes, csak például kerekítési pontatlanság okoz hibát. Lásd Patriot esemény [1]

Adatfolyam tesztelés

Ennek a tesztelési módszernek az alapötlete az, hogy a unit bemeneti adatokat kap, ezekkel az adatokkal műveleteket végez, majd az eredménnyel visszatér. Az adatok útja a programrészlet folyamatábrájába berajzolható. A teszt során azt vizsgáljuk, hogy az adat a memóriában megfelelő módon elérhető-e és az adatelem értéke helyes-e, hogy az eredmény a későbbiekben helyes legyen.

Ennek a tesztelési módszernek is, mit az előzőnek, van statikus része. A statikus szakaszban analizáljuk a forráskódot - a potenciális hibák már itt kiderülhetnek. A dinamikus szakaszban teszt adatokkal futtatjuk a programot és ellenőrizzük a visszakapott adatokat.

Milyen hibák fordulhatnak elő:

- az adatelem értéket kap, majd feldolgozás előtt újból értéket kap (melyik értékadás helyes);
- az adatelem felhasználásra kerül, de nem kap értéket;
- az adatelem kap értéket, de a felhasználás előtt töröljük;
- az adatelem rendelkezésre áll és inicializált, de nem használjuk fel.

Az adatfolyam tesztelés alatt a következő adattípusokat használjuk fel: nem definiált, definiált, újra definiált, abnormális.

Az értelmezett műveletek: értékadás, felhasználás, törlés.

Mint látható ez az eljárás nem a programozási nyelvek által használt adattípusokat és műveleteket jelenti, hanem a tesztelés szempontjait vesszük alapul.

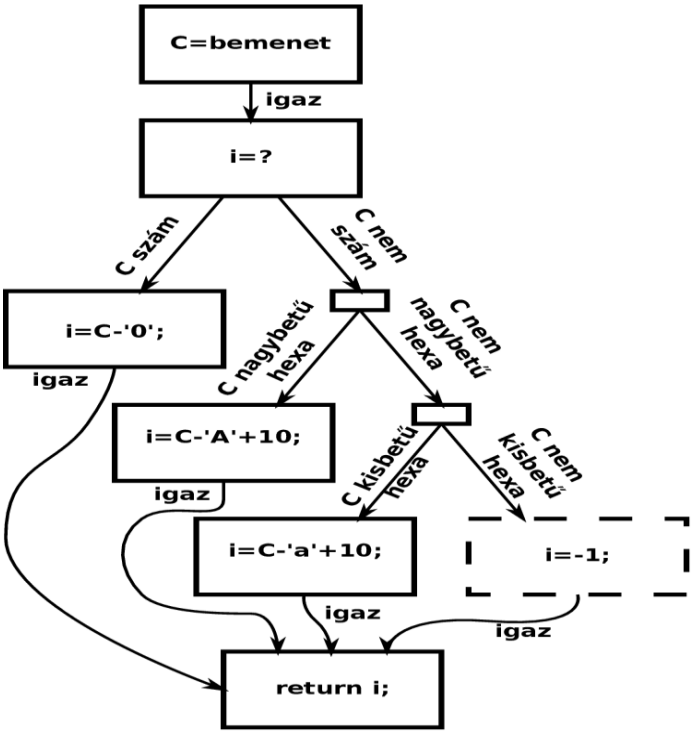
A hibák elkerülésére az egyik módszer a program “bedrótozása”, amely azt jelenti, hogy olyan kódrészleteket helyezünk a kódba, amelyek a hiba elkövetésekor hibaüzeneteket küldenek és elősegítik a statikus kódanalizátor használatát.

Dinamikus adatfolyam tesztelés esetén az adott változót nyomon követjük és ellenőrizzük a belépési és kapott értékeket. Ellenőrzésre kerül, hogy a változó korrekt értékkel rendelkezik-e a művelet előtt, illetve korrekt értéket kap-e a műveletben, illetve feltételes utasításnál megfelelő-e a kérdéses útvonal.

A tesztelés lépései:

1. rajzoljuk meg a unit folyamatábráját;
2. válasszuk ki a tesztelési kritériumokat;
3. határozzuk meg a kérdéses adatfolyam utat;

4. határozzuk meg az úthoz tartozó elágazásokat;
5. számítsuk ki a kérdéses úthoz tartozó bemeneti értékeket.



5. ábra A mintapélda adatfolyam ábrája

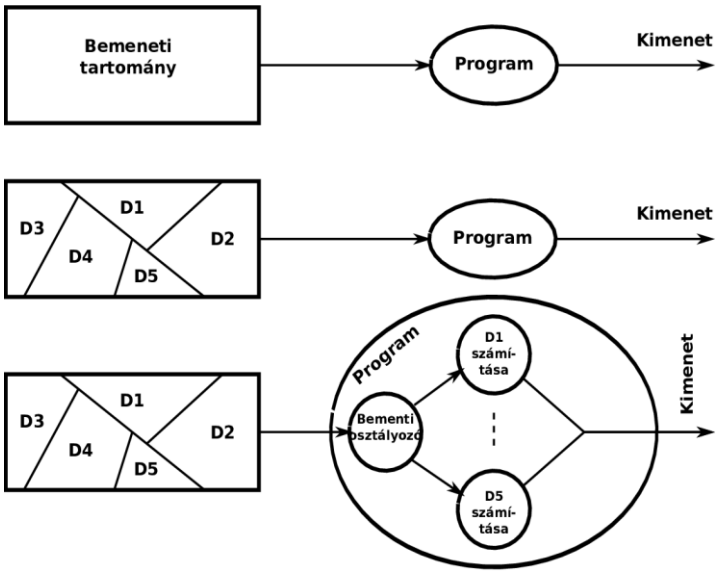
A szaggatott vonallal keretezett négyszög egy kimaradt műveletet mutat. Ebben az esetben is már rajzoláskor kiderült a hiba.

Tartomány tesztelés

Az eddigiek alapján látható, hogy egy programelem tesztelési szempontból két lényeges eleme a bemeneti adatok értelmezési tartománya és a végrehajtási út a bemenettől a kimenetig.

Az azonosítható hibák fajtái:

- számítási hibák: amelyben az adott bemenetre a végrehajtási út helyes, de a visszakapott érték hibás;
- a bemeneti tartomány hibák: a bemeneti érték hibás végrehajtási útra kényszeríti a programot.



6. ábra Tartomány tesztelés

A példánk alapján látható, hogy a bemenet egy adott érték egy utat meghatároz. Továbbá egy bemeneti adatok egy halmaza meghatároz egy utat. Azt a halmazt, amely azonos utat határoz meg tartománynak nevezik. Cél, hogy minden bemeneti tartományt meghatározzunk.

A tartománytesztelés különbözik az adatfolyam és a vezérlési út teszteléstől, mert ezek a módszerek alapvetően azt feltételezik, hogy a programban valahol hiba van és ezt keressük a fenti módszerekkel.

Ezzel szemben a tartomány tesztelés a programot egy tartomány osztályozó automataként kezeli és a következő témaköröket vizsgálja: mi a tartomány forrása, mi a tartományhiba típusa és melyek azok a teszt adatok, amelyek a hibához vezettek.

Példánkban négy tartományt találunk, ezek: ha a kérdéses ASCII kód számot képvisel, ha a kérdéses szám a hexadecimális nagybetű, ha a kérdéses érték hexadecimális kisbetű és az utolsó, ami viszont nem összefüggő tartomány ha előzőek egyike sem.

A tesztelés során a teszt adatok a specifikáció szerint a tesztelőtől származnak. Az utolsó tartományban a függvény helytelen – véletlenszerű - adatot ad vissza (ha szerencsénk van). A hiba forrása a kérdéses 11. sor hiánya.

Ez a példa – mit azt már korábban említettük – egész jellegű változókkal dolgozik, tehát a kerekítési hibák nem okoznak problémát. Azonban, ha sok számítási lépés után kapjuk a kimeneti értéket a probléma sokkal bonyolultabbá válik.

Ilyen esetben a bemeneti értékeket reprezentáló teszt vektorok túl sok teszt lehetőséget vetnek fel, ezért a tesztelést a kritikus tartomány határokon végezzük. A teszt célja, hogy kiprovokáljuk azokat a hibákat, amelyek a használat során ritkán, de előfordulhatnak.

A teszt vektorokat úgy állítjuk be, hogy a tartományhatártól nagyon kis távolságban legyenek a teszt pontjaink, mind a helyes és helytelen tartományban. Azért azt gondoljuk meg, hogy egy vo-

nal is elméletileg végtelen pontból áll.

Persze ez a kvantálás miatt véges számosságot jelent a gyakorlatban, de ez a szám is hatalmas.

Újabb kutatások során a mesterséges intelligencia néhány ágát is felhasználják a vektorok előállítására [5]

Rendszerintegrációs teszt

A rendszerintegrációs teszt során a már előre legyártott, letesztelt és hibamentesnek talált programegységek együttműködését teszteljük, amikor ezekből teljes rendszert építünk fel. A teszt célja, hogy egy helyesen működő rendszert kapjunk.

A rendszerintegráció során célszerű a unitokat inkrementális módon, tehát lépésről – lépésre, unitról – unitra összeépíteni. Így kellő mértékű verzió követés segítségével jó esélyünk van a hibák felderítésére.

A kész rendszerben a programegységek interfészeken keresztül kommunikálnak.

Tapasztalataink azt mutatják, hogy a hibák legnagyobb része ezen interfészek helytelen működésből fakad. A hibás működés másik forrása az, hogy programelemek egymást is zavarhatják.

A leggyakoribb interfész hibák típusai:

- specifikációs hiba;
- konstrukciós hiba;
- az adott funkció nem a megfelelő helyen van;
- az eredeti specifikációhoz képest a kérdéses funkció változott;
- új funkció került a rendszerbe;
- rosszul értelmezett az interfész működése;
- az interfészt rosszul kezelik;
- rossz hibakezelés;
- inicializációs hiba.

A rendszerintegrációs teszt szintjei:

1. belső (intra) teszt, ahol a modulok egyszerű együttműködését vizsgáljuk,
2. külső (inter) teszt, ahol a rendszert vizsgáljuk, úgy, hogy rendszerfüggetlen interfész felületeket biztosítunk a tesztelés számára,
3. pár teszt, ahol már több rendszert kapcsolunk össze azért, hogy ezek együttműködését vizsgáljuk.

A rendszerintegrációs technikák közül csak az inkrementális módszerrel foglalkozunk.

Az inkrementális módszer szükséges lépései:

1. minden egyes integrációs lépés előtt a rendszer állapotáról egy mentést kell eszközölni a verziókövető rendszerben. Ez azért fontos, mert probléma esetén vissza tudunk “a még helyes” szintre térni;
2. az aktuális inkrementális lépést hajtsuk végre;
3. futtassuk le a kívánt teszt ciklusokat;

4. ellenőrizzük az eredményeket;
5. dokumentáljuk a lépést, mind helyes működés, mind hiba esetén;
6. hiba esetén "rendeljük meg" a javítást;
7. végezzük a tesztelést a javítás után addig, amíg az adott inkrementális lépés eredményei nem helyesek;
8. ha az adott állapot helyes működést mutat lépjünk a következő szintre.

A felsorolást elolvasva ez teljesen természetesnek tűnik. Azonban gyakorlati tapasztalataink azt mutatják, hogy ezek lépések vagy nincsenek végrehajtva, vagy az inkremens túl nagy, nevezetesen egy lépésbe túl sok programegységet vonnak be, illetve a verzió követés nem történik meg a lépések között.

Az inkrementális módszer nagy előnye a módszeresség, nagy hangsúlyt fektet a kis részletekre. Ha kell a rég tesztek is újrafuttathatók a rendszeren és ösztönzi a hibák gyors javítását.

Hátránya az, hogy az aprólékossága miatt lassú.

A rendszerintegrációs teszt kritikus pontja a hardver – szoftver integráció tesztelése. A legnagyobb probléma az, hogy egy nem megfelelő működés honnan ered, hardver, vagy szoftver problémával állunk-e szemben. Ezért a következő lépéssort célszerű betartani:

- a szoftver telepítése a prototípus hardverre eleinte legyen inkrementális;
- eleinte a cél hardveren kis számú teszt lépést hajtunk végre;
- esetleg a hardver ellenőrzésére készítsünk speciális unitokat (de ne felejtsük a rendszerben),

a unitok és az integrációs lépések egy része mehet a fejlesztő rendszeren, csak később tegyük át a cél hardverre.

A későbbiekben felmerülő problémák elkerülésére a szoftver – hardver integrációra vonatkozó információkat egy kompatibilitási mátrixban kell rögzíteni. Ez minden követelményt mindkét oldalról rögzít.

A rendszerintegrációs teszt kategóriái:

- interfész integritási teszt, amelyben minden egyes modul interfészeit teszteljük;
- funkcionális validáció, amelyben az integrációra került modulok esetleges hibáit vizsgáljuk;
- elejétől – végéig (end-to-end) teszt, amelyben a teljesen felépített rendszer adott bemenetre adott kimeneteit vizsgáljuk (ez igen bonyolult lehet véges automaták reprezentációja esetén);
- páros validáció, amelyben két összekötött rendszer együttes működését vizsgáljuk;
- interfész stressz teszt, amelyben a kérdéses interfészt maximális terheléssel vizsgáljuk;
- állóképességi teszt, amelyben a rendszert előre meghatározott, de hosszabb ideig folyamatos átlagos terhelés alatt vizsgáljuk.

Számos esetben az integrációs eljárásban úgynevezett idegen komponensek is felhasználásra kerülnek, például BSD TCP/IP interfész. Ezek beépítésére a szakirodalom három módszert említ, ezek:

1. burkolás (wrapping), amelyben egy köztes szoftver interfész kerül a rendszer és az idegen komponens között (Pl. HAL);
2. ragasztás, olyan funkciók megvalósítása, amely az idegen komponenseket elérhetővé teszi (p. htons);

3. betűzés (tailoring), az adott idegen komponens képességeinek növelése, hogy az a rendszerünkhöz illeszkedjen.

Biztonságkritikus esetben ezek az idegen komponensek sem kerülhetik el a tesztelést, az alkalmazott módszerek:

- fekete doboz teszt, csak a funkcionalitást nézzük, a komponens felépítését nem;
- rendszer szintű hiba beviteli teszt, ekkor az adott komponenssel felépített rendszer hibatűrését vizsgáljuk;
- működési tesz, a rendszer helyes működését vizsgálja.

Rendszer tesztelés

A rendszer tesztelés a következő vizsgálatokat végzi el:

- alap tesztelés, amely a rendszer installálhatóságát, konfigurálhatóságát és működési állapotba hozhatóságát vizsgálja;
- funkcionalitás, amely szerepe a rendszer működését vizsgálja a követelmények és a specifikáció szempontjából;
- robusztussági tesztelés, amely a rendszer hibatűrését teszteli;
- inter operábilitási teszt, amely a rendszer más rendszerekkel történő együttműködési tulajdonságait vizsgálja;
- teljesítmény teszt, a válaszidőket és az erőforrásokat teszteli;
- stressz tűrési teszt, amely a rendszert olyan körülmények közé helyezi, amelyek a rendszer működésére jelentős befolyással vannak, pl. erőforrások csökkenése;
- terhelési és stabilitási teszt, a rendszert hosszabb időtartamig folyamatos terhelés alatt vizsgálja;
- regressziós teszt, a rendszer hogyan viselkedik új elemek installálásakor;
- dokumentációs teszt, amely a dokumentáció teljességét ellenőrzi.

A felhasználó szempontjából a legfontosabb teszt típus a funkcionalitási teszt, amely a következő tulajdonságokat teszteli:

- boot teszt, a rendszer indulását és az inicializálási lépéseit vizsgálja;
- diagnosztika, a hardver modulok megfelelő állapotait vizsgálja. Hibásan működő modul esetén a rendszer nem működhet teljesen helyesen;
- kommunikációs teszt, a rendszeren belüli és a külvilág felé történő kommunikációt vizsgálja, úgymint az alapvető kapcsolat tesztelése, képességi teszt, viselkedési teszt, kapcsolat bontási teszt;
- naplózási és követési teszt, amely a rendszer nyomkövetési képességeit vizsgálja;
- EMS a rendszer elemeinek főbb funkcióit vizsgálja kezelési szempontból;
- GUI teszt, ha van GUI ennek áttekinthetőségét és kezelhetőségét teszteli;
- biztonsági teszt, teszteli a rendszer titkosságát (ha ez követelmény), az integritást, a rendelkezésre állást;
- tulajdonság teszt, ez a teszt olyan tulajdonságokat vizsgál, amelyek nincsenek (esetlege-

sen nem részletezve) a specifikációban lefektetve, de a rendszer működését, vagy működtetését érinthetik.

Biztonságkritikus rendszerek esetén a robusztussági teszt is rendkívül fontos, mivel rendszer meghibásodás esetén, vagy funkció kiesés esetén a rendszernek működőképesnek kell maradnia.

Ebbe a teszt csoportba a következő eljárások tartoznak:

- határérték vizsgálat, amely határértékhez közeli, illetve hibás adataik által okozott rendszer reakciókat teszteli;
- teljesítmény kimaradás teszt, mi történik, ha pillanatnyi teljesítmény kimaradás történik,
- modul ki-be mozgatás, hogyan reagál a rendszer egyes modulok eltávolítására, vagy behelyezésére;
- megbízhatósági teszt, statisztikailag a rendszer mennyire biztonságos működésű, illetve milyen mértékben képes egy meghibásodás után a normál működési állapotba kerülni;
- csökkentett mód teszt, vizsgálja, hogy a rendszer a specifikáción kívüli állapotban milyen működési jellemzőket mutat.

A teljesség igénye nélkül – mint láthatjuk néhány tesztelési lépést nem ismertetünk részletesen – a következő fontos kérdés a stressz teszt.

Ebben a tesztelési fázisban a rendszert a specifikációhoz képest túlterheljük és vizsgáljuk, hogy a rendszer válaszidői, pontossága milyen mértékben változik. Ezekben az esetekben a real-time tulajdonságok és hálózati jellemzők jelentik a kritikus pontokat.

Real-time esetben kritikussá válhat a rendszer által felhalmozott overhead mennyisége, amely akár a rendszer teljes leállásához is vezethet. Vizsgálni kell a ciklus vesztés, a memória szivárgás és a memória határátlépés problémáit.

A memória szivárgás és határátlépés számos esetben későbbi rendszerhibákhoz vezethet. Ezeket a problémákat a megbízhatósági teszt is felderítheti.

ELEMZÉSEK

A cikk ezen részében a bevezetőben felsorolt események rövid elemzését végezzük el tesztelési szempontból és kísérletet teszünk arra, hogy kiderítsük mi vezetett az adott esemény bekövetkezéséhez, továbbá, hogyan lehet az ilyen eseteket elkerülni.

Hangsúlyozzuk, hogy nem célunk a tervező és tesztelő kollégákkal kapcsolatban semmilyen minősítő kijelentést tenni, csak és kizárólag az okokat és a megoldásokat keressük.

Patriot esemény

A Patriot vezérlő számítógépének feladata a bejövő célobjektumok követése és elfogása. Ha egy célobjektum adott tulajdonságokkal rendelkezik – ebben az esetben a egy Scud rakéta jellemzőivel – akkor a rendszer ezt követi és amennyibe az indítási ablakba esik az üteg a célra rakétát indít.

A szoftver hiba akkor adódott, mikor a követő algoritmus a Scud következő pozícióját számította ki az eddigi mozgásjellemzőiből.

A vezérlő számítógép 24 bites fixpontos számábrázolással dolgozik, a számítási időalap 0,1 másodperc. A 0,1 egy végtelen ciklikus bináris tört, amelyet a bináris pont után 24 bit hosszúságban levágtak. A pontatlanság időpillanatról időpillanatra nőtt. Nyolc óra elteltével a Patriot célzási pontatlansága 20%-ra nőtt. Ez elegendő volt a célpont elhibázásához.

Az ok egyértelmű szoftverhiba. Mi vezethetett ehhez a hibához. Azt egyértelműen leszögezhetjük, hogy ez egy egyértelmű tervezési hiba, amit egy állóképességi teszt jó eséllyel felderíthetett volna. A szoftver tervezői úgy oldották meg a problémát, hogy az számábrázolást megnövelték, így a levágásból eredő pontatlanság nem növekedett a kritikus mérték felé.

Araine 5 esemény

Az eseményt egy olyan szoftver modul okozta, amelynek a repülés ezen szakaszán semmilyen szerepe nem lett volna. Feladata az volt, hogy az indítás első másodperceiben a szerelvényt függőlegesen tartsa, ha tolóerő aszimmetria lépne fel.

A hiba közvetlen oka egy újra nem tesztelt **double → short int** típuskonverzió során bekövetkezett tartományon kívül kerülés volt – vagyis a double változó értéke nagyobb volt, mint a short int típus számábrázolási tartománya, így az átadott érték teljesen rossz volt.

Ezt a modult évek óta azonos módon az Ariane 4 rakétákban problémák nélkül alkalmazták. Azonban az Ariane 5 kezdeti gyorsulása nagyobb volt mint elődjéé, ezért fordulhatott az adott indításnál elő a fent említett konverziós hiba.

Kérdés mi vezethetett ide. Leszögezhetjük, hogy alapvető szoftver tervezési hiba az, hogy egy már funkcióját vesztett szoftver elem beleszólhat a repülés menetébe.

Ennek a problémának már a tervező asztalon a statikus tesztelés alatt ki kellett volna derülni, sajnos ez nem történt meg. Miután számos probléma mentes Ariane 4 indítás történt és a modul semmilyen hatást nem gyakorolt a repülésekre később sem figyeltek fel rá.

Másik közrejátszó faktor a nagyobb horizontális sebesség, amely végül is az előző tervezési hibára mintegy „szuperponálódva” okozta a problémát. Szintén a tesztelés elégtelenségére vezethető vissza.

Valószínűsítjük, hogy ez egy egyszerű interfész teszttel kiszűrhető probléma volt.

Bírálatként – nyomatékosan nem a tervezőkre és a tesztelőkre - szeretnénk megjegyezni, hogy sajnos szoftverfejlesztőként nagyon jól ismerjük a projektek életciklusát és elmondhatjuk, hogy bizonyos menedzser körök még biztonságkritikus rendszerek esetén a biztonsági szempontokat háttérbe szorítva a termék indokolatlan kiadását erőltetik, lásd Challenger katasztrófa.

Quantas QF72 esemény

Ennek az esetnek a leírását jelentősen leegyszerűsítjük, mert a részletes magyarázathoz az A330 irányítási rendszerének részletes ismertetése szükséges, ezért a teljesség igénye nélkül emeljük ki

azt a szoftver problémát, amely az adott eseményhez vezetett.

A repülési adatokat kezelő ADIRU (Air Data and Inertial Reference Unit) berendezések közül az első meghibásodott. A meghibásodás eredményeként az ADIRU1 adott időpontokban az általa közölt adatok közé kiugróan hibás adatokat helyezett.

Mivel a gép repülési jellemzői létfontosságú adatok, ezért a kérdéses adatok hihetőségét az elsődleges repülésirányító számítógép is ellenőrzi, például hihetőségi vizsgálatot végez állásszög változásra.

Az ellenőrzést végző program azonban pontosan azokban a mintavételi időpontokban végzett ellenőrzést, mikor a hibás adatok közlésre kerültek, így ezeket helyesnek vette.

A hibás adatok kritikus állásszög adatokat jeleztek, így az elsődleges repülésirányító számítógép elkerülendő az átesést radikálisan beavatkozott. [3]

Kérdésünk a szokásos, hogy lehetséges egy ilyen program hiba és hogyan lehet a jövőben elkerülni. Sajnos azt kell mondanunk, hogy az ilyen hibák előre szinte felderíthetetlenek. Ennek oka, hogy számos véletlen paraméternek kell együttesen fellépni, hogy az esemény bekövetkezzen.

Hozzá kell tennünk, hogy az eseménynek voltak előjelei a pilóták kijelzőin igen sok zavaró információ jelent meg, amelyet nem tudtak értelmezni.

A probléma megoldására az EADS tett lépéseket, illetve figyelmeztette a pilótákat, hogy felismerjék az ilyen események előjeleit. Néhány hónappal később hasonló eset majdnem bekövetkezett, de a tények ismeretében a pilóták időben be tudtak avatkozni. [6]

ÖSSZEFOGLALÁS

Egyértelműen kijelenthetjük, hogy a szoftver mindennapi életünk része, nem kerülhetjük ki, nem hagyhatjuk figyelmen kívül. Ezért minden rendszer tervezőnek, programozónak és tesztelőnek, de a gazdasági vezetőnek és menedzsernek is tudnia kell, hogy a szoftverkészítés és tesztelés rendkívül aprólékos, felelősségteljes és végtelenül bonyolult munka, különösen biztonságkritikus alkalmazások fejlesztése esetén.

FELHASZNÁLT IRODALOM

[1] Arnold, Douglas, Two Disasters caused by Computer Arithmetic Errors,
[2] Prof. J. L. LIONS, ARIANE 5 Flight 501 Failure Report by the Inquiry Board,
[3] In-flight upset, 154 km west of Learm3"onth, WA, 7 Oct 2008, VH-QPA, Airbus, 2008. 11. 14
[4] Kshirasagar Naik, Priyadarshi Tripathy, Softvware Testing and Quality Assurance,
[5] Gergely Takács, Novel Fuzzy Approach for Ranking Test Vectors, 2012
[6] Háy György, Megbokrosodott számítógépek, 2012