Schuster György, Ady László

BIZTONSÁGKRITIKUS SZOFTVER FEJLESZTÉS

A szoftver kritikus sikertényező. Manapság nincs olyan bonyolultabb műszaki alkotás, amelyben nincs szoftver támogatás. Ez a kijelentés a közlekedésben résztvevő járművekre fokozottan igaz, akármelyik területet is nézzük. Ezek a szoftver elemek megjelennek minden területen a szórakoztató rendszerektől az intelligens forgalomirányítások keresztül a kritikus fedélzeti rendszerekig. Minél több biztonságkritikus feladatot bízunk ezekre a fedélzeti rendszerekre, annál kevesebb feladat hárul a folyamatokban résztvevő emberekre, illetve olyan funkciókat is meg tudunk valósítani, amelyek korábban elképzelhetetlenek voltak. Sajnos a rendszerek növekvő bonyolultsága növeli a hibák előfordulásának lehetőségét. Ebben a cikkben azt mutatjuk be, hogy milyen szempontokat és szabályokat kell betartani, ha olyan beágyazott rendszerek szoftvereit kell előállítani, amelyek kritikus biztonsági feladatokat látnak el.

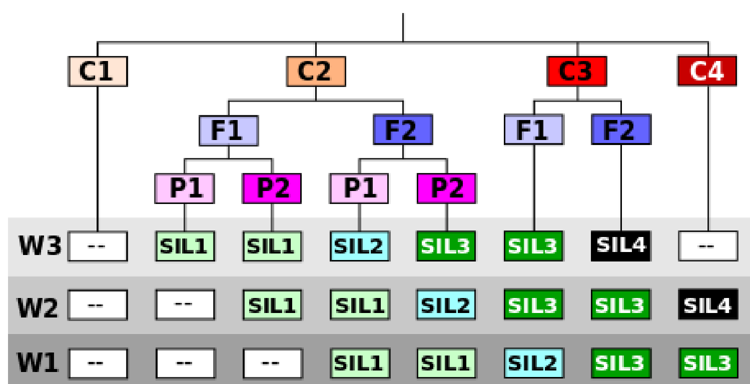
***Kulcsszavak**: szoftver, biztonság, fejlesztés*

BIZTONSÁGI SZINTEK

A fentiek alapján számos olyan terület van, ahol a felhasználás, illetve a működés biztonsága rendkívül lényeges. Minél nagyobb a működési biztonság, annál ritkábban következik be káresemény. A műszaki eszközök biztonsági szempontból négy osztályba sorolhatók. Ezek a SIL (Safety Integrity Level) szintek. A SIL1 a legalacsonyabb a SIL4 a legmagasabb szint. A meghibásodások megengedett valószínűsége az 1. táblázatban látható. A táblázatban látható valószínűségi értékek a vizsgált termék teljes életciklusára vonatkoznak.

SIL szint	A hiba bekövetkezési valószínűsége
SIL1	$10^{-2} \leqslant p < 10^{-1}$
SIL2	$10^{-3} \leqslant p < 10^{-2}$
SIL3	$10^{-4} \leqslant p < 10^{-3}$
SIL4	$10^{-5} \leqslant p < 10^{-4}$

1. táblázat SIL szintek IEC61508 szerint [1]



1. ábra Kockázati gráf [saját szerkesztés]

A megbízhatósági igényt sok tényező befolyásolja. Nem mindegy milyen jellegű a kezelőszemélyzet képzettsége, milyen jellegű a meghibásodás mértéke és előfordulásának jellege és végül, de nem utolsósorban milyen a keletkezett kár és a hiba elháríthatósága.

Az IEC 61 508 [1] és az IEC 61 511 [2] szabványok megadnak egy kockázati gráfot, amely a fenti szempontokat és megbízhatósági igényt mutatja be (1. ábra).

A jelölések magyarázata:

- a káresemény besorolása:
  - **C1** egy, vagy néhány személy könnyű sérülése, illetve a környezet kismértékű terhelése;
  - **C2** egy, vagy több személy súlyos nem gyógyítható sérülése, esetleg egy személy halála, illetve a környezet átmeneti súlyosabb terhelése;
  - **C3** több személy halála, illetve a környezet súlyos terhelése;
  - **C4** tömeges halálos baleset és egyéb katasztrofális hatások;
- az események gyakorisága:
  - **F1** ritkán, szórványosan bekövetkező esemény;
  - **F2** gyakran bekövetkező és tartós hatású esemény;
- a kár elháríthatósága:
  - **P1** elhárítható káresemény;
  - **P2** nem elhárítható káresemény;
- a személyzet képzettsége:
  - **W1** teljesen felkészült személyzet;
  - **W2** felkészült, de nem a „legjobb” személyzet;
  - **W3** teljesen járatlan személyzet.

A kockázatot a lehetőségekhez mérten minimalizálni kell, ennek érdekében a következő intézkedéseket és szabályokat kell betartani:

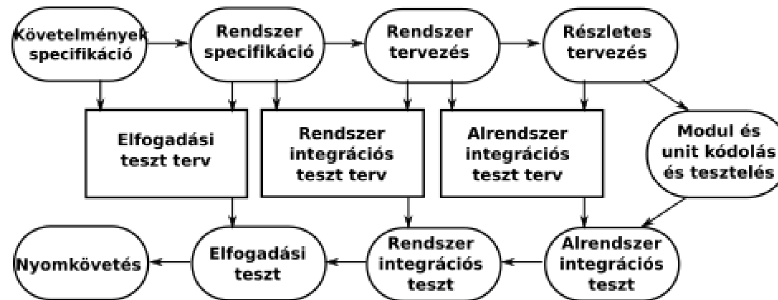
1. a megvalósítandó rendszer minden elemére el kell végezni a kockázat elemzését, tehát összetett rendszer esetén a mechanikai elemekre, az elektronikára és szoftver elemekre is;
2. meg kell vizsgálni az elkészült (elkészülendő) rendszert és meg kell valósítani a maradék kockázatot minimalizáló intézkedéseket;
3. minősíteni kell azokat a felszereléseket is, amelyekkel a fenti intézkedéseket végzik;
4. ismétlődő funkcionális teszteket kell végrehajtani, hogy a biztonsági előírások végrehajtása korrekt módon történt.

Ha biztosítani akarjuk a rendszer fokozott megbízhatóságát, akkor a rendszernek hibatűrő képességgel kell rendelkeznie ez a HFT (Hardware Fault Tolerance/Hardver Hiba Tolerancia) mutató. A HFT szintek a következők:

- **HFT0** nincs biztonsági funkció az adott elem meghibásodására,
- **HFT1** az elem egy működési hibát „elvisel”, legalább két hibának kell bekövetkeznie a működési funkció elvesztéséhez,
- **HFT2** az elem két működési hibát „elvisel”, legalább három hibának kell bekövetkeznie a működési funkció elvesztéséhez.

ÉLETCIKLUS ÉS FEJLESZTÉSI MODELL

Biztonságkritikus fejlesztéseknél a legelterjedtebb életciklus modell az úgynevezett V-modell. A szakirodalomban a modellnek számos ábrázolása található, de ezek az ábrázolások filozófiájukat tekintve nem, csak lépéseik felbontását tekintve különböznek. A 2. ábra a V életciklus modell egy ábrázolást mutatja.



2. ábra A V életciklus modell [saját szerkesztés]

Az életciklus modell elemei:

- **követelmények és specifikáció**, a szoftver tervezés kiinduló lépése ebben a fázisban kerülnek meghatározásra az alapvető funkciók és rögzítésre kerülnek a részletes specifikációk. Ebben a fázisban történik az átfogó kockázatelemzés is;
- **rendszer specifikáció**, ez a lépés a rendszer logikai szintű specifikálása, meghatározásra kerülnek a szoftver összetevői, ezek kapcsolata;
- **rendszer tervezés**, az életciklus ezen fázisában az előző specifikációk alapján kerül sor a rendszer logikai szintű tervezésére;
- **részletes tervezés**, modul és unit szintű tervezés. Itt már vannak algoritmusok, adatszerkezetek és kommunikációs protokollok;
- **modul és unit kódolás és tesztelés**, a modulok és az unit-ok kódolása és ezek tesztelése;
- **alrendszer integrációs teszt**, az elkészült unit-ok és a részmodulok összeépítésének tesztelése;
- **rendszer integrációs teszt**, a rendszer integrálása utáni tesztelés;
- **elfogadási teszt**, a szoftver utolsó tesztelési lépése, ahol a fejlesztő(k) a tesztelési terv alapján lépésről lépésre bizonyítja a megrendelőnek a szoftver működőképességét;
- **elfogadási teszt terv**, forrása a követelmény specifikáció és a rendszer specifikáció, ez alapján történik az elfogadási teszt;
- **integrációs teszt terv**, a rendszer összeépítésének tesztelési terve, ez biztosítja a rendszer elemeinek hibátlan együttműködését. Forrása a rendszer specifikáció és a rendszer terv;
- **alintegrációs teszt terv**, a kisebb elemek összeépítésének teszt terve. Forrása a rendszer terv és a részletes terv.

Biztonságkritikus rendszerek esetén a tesztelésről a hangsúly nagyrészt áttevődött a tervezésre. A tervezési és tesztelési szempontok és konvenciók az életciklus alapján kerülnek alkalmazásra.

Tervezési szempontok az életciklus fázisai alapján

Mielőtt egy szoftver projekt elindul egy részletes funkcionális specifikációra van szükség. Első közelítésben ez egy viszonylag egyszerű probléma, azonban ez egyáltalán nincs így. Többéves tapasztalatunk azt mutatja, hogy a megrendelő által megadott funkció terv, amely a funkcionális specifikációt tartalmazza szinte minden esetben hiányos, vagy erősen hiányos. Ezért ebben a lépésben a tapasztalt fejlesztőknek mindenképpen részt kell venni ebben a fázisban. Ezek a résztvevők tapasztalataik alapján számos szempontot kiemelhetnek, illetve figyelmeztethetik a megrendelőt, hogy egy kérdéses funkció nincs teljesen definiálva. Ezért a kiindulási funkcionális specifikáció elkészítése egy iterációs folyamat.

Az első fázisban megtörténik a kockázatelemzés is. Itt ne a gazdasági kockázatra gondoljunk, hanem szigorúan műszakira. Szerencsés esetben a szoftveren kívüli alkotó elemek már olyan készültségi szinten vannak, hogy számos tulajdonság előre látható. Azonban többször az ilyen projektek minden alkotó elemének fejlesztése egy időben kezdődik[1]. Ilyen a műszaki kockázatelemzése lényegesen bonyolultabb és előre meg nem jósolható tulajdonságok és hatások befolyásolhatja.

A funkcionális specifikáció és kockázatelemzés után a rendszer-specifikáció kerül sorra. Ekkor készítik el a rendszer felépítésének az előző lépés alapján eldöntik a megfelelő SIL szintet, meghatározzák a megfelelő szoftver eszközöket. Ebbe a fázisba be lehet vonni a megrendelőt, de nem okvetlenül szükséges. A fázis kimenetén a fejlesztők megkapják a rendszer tervezéshez szükséges irányelveket, szabványokat és a fejlesztői csoportra jellemző eljárásokat és a fejlesztő eszközök körét.

A rendszer tervezése már a rendszer logikai felépítésének és főbb részeinek tervezését jelenti. Ebben a fázisban már konkrétan a szoftver tervezése történik. Ekkor már meghatározzák a modulokat és a feladatok is szétosztásra kerülnek a különböző csoportok között. Ezek a csoportok léphetnek át a következő fázisba, ahol megtörténik a részletes tervezés.

Ebben fázisában már részletekbe menően tervezik meg a megfelelő szoftver egységeket. Itt kerülnek elő a megfelelő adatszerkezetek, függvények, algoritmusok és egyéb részletekbe menő architekturális elemek.

A unit kódolás és tesztelés fázisban a szoftver elemek programkódjainak előállítása történik a megfelelő programozási nyelvvel, a meghatározott programozási környezetben és a meghatározott kódolási konvenciókkal, ilyen például a megfelelő MISRA szabályok alkalmazása.

A kiválasztott programozási nyelv az esetek többségében a C és az ADA. Kisebb biztonsági követelmények esetén (egyelőre SIL2-ig) kezdenek a script nyelvek terjedni, mint például a MicroPython.

Ehhez a fázishoz rendeltük az unit tesztelést is. Ez első ránézésre egyszerűnek tűnik, de a szoftver helyességének a 80%-a körülbelül itt történik meg, hiszen itt kerülnek megírásra a kódok. Ez a tesztelési fázis nagyon alapos, mind statikus, mind dinamikus tesztelési módszereket tekintve. Ebben a fázisban gyakran alkalmaznak teszt automatákat a minél kiterjedtebb teszt esetek vizsgálatára.

1 Sokkal egyszerűbb egy már meglévő repülőszerkezethez avionikát tervezni, mint egy most kezdődő fejlesztéshez.

A következő – az alrendszer integrációs – fázisban a már megírt tesztelt és helyesnek talált szoftver egységeket integrálják és az integráció minőségét tesztelik. Ez a lépés sem tűnik bonyolultnak, azonban több, mint 20 hibalehetőséget szoktak erre a fázisra felsorolni. A hibás integrációra kiváló mintapélda a Mars Climate Orbiter és a Mars Polar Lander emlékezetes esete [4][5][6]. Mars Climate Orbiter a Mars időjárásának feltérképezésére 1998-ban indított szonda sikeresen elért a vörös bolygóhoz, ahol azonban leszállás közben darabokra szakadt. A problémát az okozta, hogy a földi vezérlőrendszerek imperiális mértékegységekkel számoltak, míg a szonda fedélzeti számítógépei metrikus egységekben várták volna az adatokat [16].

Mars Polar Lander Szintén a Naprendszer negyedik bolygójára indult 1999-ben ez a leszállóegység, ahova meg is érkezett, de sajnos nem a szándékolt módon. Sima leszállást helyett ugyanis nagy sebességgel a Mars felszínébe csapódott, minekután a vezérlését végző szoftver a leszállólábak süllyedés közbeni rezgését - nyilván valamilyen a jeltüskéket kiszűrő algoritmus hiányában - a felszínre érkezés jeleként értelmezte, és kikapcsolta a lassító rakétákat [16].

A továbbiakban ezt az integrációs folyamatot addig ismétlik a tesztelésekkel együtt, amíg a teljes rendszer nem készül el.

Utolsó előtti fázis az elfogadási teszt, amely legalább két szintre bontható. Az első szint a fejlesztők által elvégzett részletes funkcionális teszt, amely a lehetőségekhez mérten a két első lépés minden követelménye szerinti megfelelőséget vizsgálja. A második szint, ahol a fejlesztők csapat a követelmény specifikációnak megfelelően a „megrendelő” felé bizonyítja a szoftver működőképességét és a specifikáció maradéktalan teljesítését. Ez egy komolyabb szoftver esetén akár hónapokat is igénybe vehet.

Az utolsó fázis a nyomkövetés. A szoftver további viselkedését is érdemes követni, illetőleg a fejlesztőnek vannak garanciális kötelezettségei. Volt már arra példa, hogy egy későbbi projektben modul újra felhasználáskor derült ki olyan programhiba, ami már egy előző szoftverben is szerepelt. Ekkor az első dolog, hogy a „megrendelőt” értesíteni kell és ki kell kérdezni szándékáról[2].

FEJLESZTŐI KÖRNYEZETEK

Ha biztonságos rendszer építése a cél, akkor azt minősített eszközökkel kell megtenni. Ez a szoftverek esetében fokozottan igaz. Ha az alkalmazott eszközök nem megfelelőek az előállított bináris kódok hibásak lehetnek.

A tervezés során minden lépésben informatikai támogatást vesznek igénybe. Ez részben kockázatot jelent, részben növeli a biztonságot. A kockázat az informatikai segédeszközök lehetséges hibáiban rejlenek, viszont ezen eszközök használata lehetővé teszi a rendszer biztonságos tervezését, előállítását és tesztelését.

Minden mérnök ismeri azt az érzést, amikor egy műszaki alkotás elindul „Mit felejthettem még el?”. Ennek a hatásnak a csökkentésére az integrált tervezői és fejlesztő rendszerek alkalmazása egy jó megoldása. Azonban biztonságkritikus fejlesztésben a rendszernek, vagy minimum a rendszer kritikus részeinek minősítettnek kell lenni.

2 Lényeges, hogy ne kezdjünk kapkodni. Ha javítást kell végrehajtani a már leszállított rendszeren. A javított szoftvert újra validálni kell. Az is előfordulhat, hogy a javítás túl nagy kockázatot jelent és nem engedélyezik.

A teljesség igénye nélkül milyen elemei lehetnek egy ilyen integrált rendszernek:

- ✈ verzió követő;
- ✈ dokumentáció készítő;
- ✈ grafikus tervező felület;
- ✈ fordító és szerkesztő programok;
- ✈ teszt generátorok;
- ✈ szimulátorok és emulátorok;
- ✈ nyomkövető eszközök.

Adott feladattól függően az alkalmazott eszközök listája még bővülhet.

KORLÁTOZÁSOK ÉS SZABVÁNYOK

A korszerű fejlesztések nagy többsége igen terjedelmes, ami azt jelenti, hogy számos szakember dolgozik az adott feladaton. Az ilyen fejlesztések nagy része C programozási nyelven történik. Sajnos a C túl szabad programozási nyelv, ami félreértéseket és hibákat okozhat. Ezért nem csak C nyelvre bevezettek egy korlátozás csomagot a C90 és C99 szabványra, amit a MISRA (Motor Industry Software Reliability Association) bocsájtott ki, amely irányelveket és szabályokat fektet le. Egy irányelv lehet kötelező, ekkor ezt maradéktalanul be kell tartani, lehet megkövetelt, ekkor kismértékű formális eltérések megengedettek és lehetnek javasoltak, ekkor célszerű ezeknek a betartása, de nem kötelező.

Amennyiben a fejlesztők ezeket az előírásokat betartják, kisebb valószínűséggel hibáznak, vagy írnak félreérthető kódokat. Azt viszont nem biztosítja, hogy az előállított kód helyes lesz.

Az előzőekben már említettük az IEC 61 508 szabványt. Az IEC 61 508 egy „általános” szabvány, amely gyakorlatilag minden iparágban használható. Hét részből áll és egy „alap szabványként” kezelendő. Adott ágazatokban közvetlenül használható, illetve ebből kiindulva ágazat és alkalmazás specifikus szabványok forrásaként használható. A részei:

1. meghatározza az elvégzendő tevékenységeket a teljes biztonsági életciklus minden szakaszában, valamint a dokumentáció követelményei, a szabványnak való megfelelés, irányítási és biztonsági értékelés szempontjából;
2. elektromos, elektronikus, programozható berendezése követelményeket foglalja össze;
3. a szoftver követelmények értelmezi az 1. rész általános követelményeit figyelembe véve elemzi a hardver és a szoftver összefüggéseit;
4. definíciók és használt rövidítések magyarázata;
5. példák a biztonsági szint meghatározásához szükséges módszerekről, kockázat elemzési példák és SIL szintek bemutatása;
6. iránymutatások a 2. és 3. szintek alkalmazásáról;
7. biztonságtechnikai és szoftverfejlesztési technikák leírása és a hivatkozások a forrásokra.

A szabvány rendkívül részletes, terjedelme közel 400 oldal.

BIZTONSÁGKRITIKUS SZOFTVER FEJLESZTÉS KOCKÁZATAI

A biztonságkritikus szoftver fejlesztés nagy mértékben meg van támogatva szabványokkal és eszközökkel, módszertanokkal. Ezek betartása ugyanakkor nem garantálja a megfelelő biztonsági szint elérését.

A kockázatok az alábbiak szerint sorolhatók be [3]:

- program hibák (dokumentált szoftver hibák)
  - belső hibák;
  - interfész hibák (Mars Climate Orbiter [6]);
  - funkcionális hibák;
- emberi hibák
  - kódolási és szerkesztési hibák (Mariner 1 [7][8]);
  - kommunikációs hibák a csapaton belül;
  - kommunikációs hibák a csapatok között (Mars Climate Orbiter [6], Mars Polar Lander [4][5]);
  - hiba a követelmények felismerésében;
  - hiba a követelmények feldolgozása során;
- folyamat hibák
  - nem megfelelő kód vizsgálat és tesztelési módszereket (Therac-25 [9]);
  - nem megfelelő interfész specifikációk, nem megfelelő kommunikáció a szoftver fejlesztés lépései között;
  - nem megfelelő interfész specifikációk, nem megfelelő kommunikáció a szoftver/hardver fejlesztés között;
  - nem azonosított, vagy megértett követelmények, hiányos dokumentáció;
  - nem azonosított, vagy megértett követelmények, nem megfelelő tervezés.

A biztonságkritikus szoftver funkció gazdagságának a növekedésével és a piaci igények által megkövetelt kiadási gyakoriság mellet a biztonságkritikus szoftver fejlesztés járulékos eszköz parkjának mérete és komplexitása nagyra nőtt. Eközben a fejlesztő szervezetek nagysága és összetettsége is jelentős méretűvé vált. A biztonságkritikus fejlesztés egy komplex rendszerré fejlődött, aminek a fejlesztő csoportok csak kis részére vannak rálátással. Jelentősen mennyiségű az elérhető, megvásárolható eszközök, elemek száma. Ugyan akkor az igények kielégítéséhez munkaerőhiány alakult ki. Így biztonságkritikus fejlesztésben nem jártas szakembereket is be kell vonni ilyen projektekbe. Ezek a tendenciák szükségessé teszik az emberi viselkedés és az IT rendszerek üzemeltetésének vizsgálatát a biztonságkritikus fejlesztés folyamatában.

A hamis biztonság definíciója: az, hogy valami felől meg vagyunk győződve, hogy biztonságos, miközben veszélyben vagyunk.

„Zóna” állapot definíciója: „arról a rendkívüli összpontosítást eredményező, elmélyült tudatállapotról van szó, amelybe a programozók például akkor kerülnek, amikor kódot írnak. Ebben az állapotban érezzük hatékonynak a munkánkat. Ebben az állapotban hisszük azt, hogy tévedhetetlenek vagyunk. Ezért törekszünk mindig ennek az állapotnak az elérésére és értékeljük magunkat aszerint, hogy mennyit időt vagyunk képesek az áramlási zónában eltölteni.” [11]

A hamis biztonságérzet kialakulása és a „zóna” állapot jelentős mértékű jelenleg alacsonyan kezelt kockázat.

A biztonságkritikus szoftver fejlesztést megkönnyítik és felgyorsítják az automatizált folyamatok, eszközök. Ezek segítségével rendszeresen ellenőrzött és egyenletes szoftver minőség tartható, azonban ezek használata könnyen hamis biztonságérzet kialakulásához vezet. Ennek a megelőzése történhet rendszeres egyedileg injektált hibával. Ebben az esetben egy erre a célra megbízott személy vagy csoport titokban hibát helyez el a fejlesztési folyamat különböző részeiben és vizsgálják, hogy a hiba a tervezett módon az automata rendszerek detektálják-e. Ennek a folyamatnak a kritikus része az injektált hiba termékbe kerülésének korlátozása. Ez elérhető amennyiben a IT környezet fel van készítve erre. Az éles fejlesztői környezet automatizált klónozásával egy elszeparált környezetbe injektált hibákkal tesztelhető a fejlesztői környezet. Így az automatizált folyamatok működő képessége az éles forrás kódok érintése nélkül minősíthetőek.

Hamis biztonság érzetek:

- a(z) X szabványt betartjuk;
- a(z) X modell szerint dolgozunk;
- van automata tesztünk;
- van tesztelőnk;
- egyszer már működött használjuk fel újra;
- tartunk rendszeres review-t.

A „zóna” állapot elkerülésére használható a páros programozás.

A **páros programozás** definíciója szerint a programozók párban dolgoznak a szoftverfejlesztés alatt. A munkatársak együtt dolgoznak egy közös munkaállomás mellett [12][13]. A párban való programozásnak a következő előnyei vannak:

- megnövekedett fegyelemérzet;
- rugalmas munkafolyamatok, megszakítástűrés;
- jobb kódminőséget eredményez;
- mentorszellem, tudás és tapasztalatterjesztés;
- a munkatársak gyorsabban megismerik egymást;
- „zóna” állapot elkerülése.

Az IT rendszer üzemeltetése során jellemző a hamis biztonságérzet kialakulása. Ez jellemzően a helyes működés, a rendelkezésre állás és a sértetlenség körül alakul ki. Miniszterelnöki Hivatal Informatikai Koordinációs Iroda szerint egy kiadott rendelete alapján „Olyan előírások, szabványok betartásának eredménye, amelyek az információk elérhetőségét, sérthetetlenségét és megbízhatóságát érintik és amelyeket az informatikai rendszerekben vagy komponenseikben, valamint az informatikai rendszerek vagy komponenseik alkalmazása során megelőző biztonsági intézkedésekkel lehet elérni.” Az informatikai rendszerek hamis biztonság érzetének kockázata ellen fontos, hogy szervezeteknél olyan munkatárs, aki kidolgozza és rendszeresen végrehajtja a IT rendszer minősítését. Ez a minősítés magába kell, hogy foglalja az alapnak tekinthető C.I.A. és a biztonság kritikus fejlesztés támogató rendszerek kifogásolhatatlan működőképességét. Szükséges, mint az IT mind a biztonság kritikus fejlesztés monitorozása, pillanatnyi adatok megjelenítése, a tendenciák követése és a folyamatok minőségi ellőre jelzése. A monitorozás elengedhetetlen feltétele továbbá a CMMI szinteknek.

CMMI (**C**apability **M**aturity **M**odel **I**ntegration) egy szoftverfolyamat-fejlesztési modell, mely két megközelítésben (lépcsős és folytonos) mutatja meg az IT folyamatokat. A Carnegie Mellon

University fejlesztette ki, az Amerikai Védelmi Minisztérium és az USA kormányzati szerződései kötelezően elvárják [15].

FELHASZNÁLT IRODALOM

[1] Wikipédia: IEC 61 508 (e-dok), url: https://en.wikipedia.org/wiki/IEC_61508
[2] Wikipédia: IEC 61 511 (e-dok), url: https://en.wikipedia.org/wiki/IEC_61511
[3] Robyn R. Lutz: Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems ISBN:0-8186-3120-1
[4] JPL: Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions2000.03.22
[5] SG: A Polar Lander sztori (e-dok), url: https://sg.hu/cikkek/it-tech/10313/a-polar-lander-sztori
[6] NASA: Mars Climate Orbiter Mishap Investigation Board Phase I Report 1999.11.10
[7] Wikipedia: Mariner 1(e-dok), url: https://en.wikipedia.org/wiki/Mariner_1
[8] NASA: Mariner 1 / NSSDCA ID: MARIN1 /
[9] Nancy Leveson: Medical Devices The Therac-25
[10] MISRA Compliance:2016 Achieving compliance with MISRA Coding Guidelines
[11] Robert C. Martin: Túlélőkönyv programozóknak ISBN:9789639637863
[12] Puskás Béla: A hamis biztonságérzet kialakulásának megelőzése az informatikai rendszerek üzemeltetése során 2012.11.15 ÓE Tézis füzet 2017
[13] Dr. Ulbert Zsolt: Szoftverfejlesztési folyamatok és szoftver minőségbiztosítás 2014
[14] Tóth Géza: (e-dok), url: http://astro.u-szeged.hu/szakdolg/tothgeza/fokep.html
[15] Wikipédia: CMMI (e-dok), url: https://en.wikipedia.org/wiki/Capability_Maturity_Model_Integration
[16] Prog.hu: 9 híres programozási hiba, ami katasztrófát okozott az űrben (e-dok), url: https://prog.hu/hirek/3995/9-hires-programozasi-hiba-ami-katasztrofat-okozott-az-urben

SAFETY CRITICAL SOFTWARE DEVELOPMENT

Software is a critical success factor. Nowadays, there is no more complicated technical work without hard software support. This statement is true in case of vehicles involved in transport independent of the examined area. These software elements appeared in all areas from board entertainment systems through intelligent traffic management to critical board control systems. If we apply larger number of onboard intelligent and smart systems, the crew will have less tasks and we will be able to implement such functions that were previously impossible. Unfortunately, the increasing complexity of systems increases the possibility of errors occurrence. This article describes the aspects and rules that must be followed when producing embedded systems software that performs safety critical tasks.

Keywords: *software, safety critical, development*

Dr. Schuster György (PhD) Egyetemi docens, főiskolai tanár Óbudai Egyetem Kandó Kálmán Villamosmérnöki Kar Műszertechnikai és Automatizálási Intézet schuster.gyorgy@kvk.uni-obuda.hu orcid.org/0000-0002-8573-3670	György Schuster (PhD) College professor Óbuda University Kandó Kálmán Faculty of Electrical Engineering Institute of Instrumentation and automation schuster.gyorgy@kvk.uni-obuda.hu orcid.org/0000-0002-8573-3670
Ady László Hallgató Óbudai Egyetem Kandó Kálmán Villamosmérnöki Kar Műszertechnikai és Automatizálási Intézet ady.laszlo@kvk.uni-obuda.hu orcid.org/0000-0001-6702-6000	László Ady Student Óbuda University Kandó Kálmán Faculty of Electrical Engineering Institute of Instrumentation and automation ady.laszlo@kvk.uni-obuda.hu orcid.org/0000-0001-6702-6000

http://www.repulestudomany.hu/folyoirat/2018_1/2018-1-11-0453_Schuster_Gyorgy-Ady_Laszlo.pdf