BUDAPESTI GAZDASÁGI EGYETEM
PÉNZÜGYI ÉS SZÁMVITELI KAR

SZAKDOLGOZAT

Pálfi Andrea
nappali
Gazdaságinformatikus
Logisztikai informatikus

2018.

BUDAPESTI GAZDASÁGI EGYETEM
PÉNZÜGYI ÉS SZÁMVITELI KAR

A Titan compiler fejlesztésének bemutatása

Belső konzulens:
Dr. Fauszt Tibor

Külső konzulens:
Szabados Kristóf

Pálfi Andrea
nappali
Gazdaságinformatikus
Logisztikai informatikus

2018.

NYILATKOZAT

Alulírott ...PÁLFI ANDREA..... büntetőjogi felelősségem tudatában nyilatkozom, hogy a szakdolgozatomban foglalt tények és adatok a valóságnak megfelelnek, és az abban leírtak a saját, önálló munkám eredményei.

A szakdolgozatban felhasznált adatokat a szerzői jogvédelem figyelembevételével alkalmaztam.

Ezen szakdolgozat semmilyen része nem került felhasználásra korábban oktatási intézmény más képzésén diplomaszerzés során.

Tudomásul veszem, hogy a szakdolgozatomat az intézmény plágiumellenőrzésnek veti alá.

Budapest, 2018. év ...május... hónap ...9... nap

Pálfi Andrea

hallgató aláírása

Tartalomjegyzék

1 Titan, a compiler

1.1 Bevezetés

Amikor szoftverfejlesztő gyakornokként bekerültem az Ericsson Test Competence Center osztályára, hallgatótársaim a feladataimról érdeklődtek. Szakdolgozatom elolvasásával választ kaphatnak kérdéseikre. Szakmai gyakorlatom során abban a folyamatban veszek részt, amelyben amellé a fordító mellé, amely TTCN-3-ról C++-ra fordít készül egy olyan fordító, amely TTCN-3-ról JAVA-ra tud fordítani. A Titan 2015 óta opensource[1], ezért dolgozatom nyilvános, bárki számára elérhető.

Szakdolgozatom kulcsfejezetét a „Loggolást" keretbe foglalja a Titanról és a tesztelésről szóló fejezet. Azért alakítottam ki ezt a logikai felépítést, mert a loggolás fejezethez szükség van némi Titánnal és TTCN-3-mal[2] kapcsolatos alapismeretre is.
Miért ez a fejezet a legfontosabb számomra? Amikor az alapvető osztályok alapműveletei közé megírtuk a megfelelő függvényeket, három nagyobb feladatból lehetett választani. A többi fejlesztő az openspaceben[3] folyamatosan a logfileokról beszélt, a logokat tanulmányozta, így nem volt kérdés számomra, hogy a nélkülözhetetlennek tűnő TtcnLogger osztály implementálását fogom választani.

Amikor kollégáimnak meséltem új feladatomról, az egyikük megkérdezte, hogy tudom-e, hogy honnan ered a „loggolás"? Tőle tudtam meg, hogy régen a hajósok használtak először hajónaplót, amelyet angolul LogBooknak hívnak és a beleírt sorok voltak a log bejegyzések. A dátumokat is ugyanúgy rögzítették, mint a manapság használt logfileokban. Ekkor döntöttem el, hogy erről írom a szakdolgozatomat. Annyira felkeltette az érdeklődésemet, hogy egyből rákerestem a „LogBookra". A loggolást a hajózásnál, repülésnél mind a mai napig használják. Sőt az elektronikus logolás a logisztikában is megjelenik.

---

[1] Opensource=nyílt forráskódú szoftver, azaz a felhasználók szabadon futtathatják, tanulmányozhatják, másolhatják és tökéletesíthetik a szoftvert (Forrás: https://prestashop.hu/open-source-nyilt-forraskod-nem-egyenlo-ingyenes-fejlesztes/ letöltve: 2018. május)
[2] Testing and Test Control Notation Version 3
[3] nyitott terű iroda

Szakdolgozatom néhány alfejezetcímében egy-egy fontos kérdést fogalmaztam meg, ilyen például a „Miért loggoljunk?” alfejezet.
A későbbiekben rátérek a konkrét feladatom részletes ismertetésére, bemutatom, hogy milyen lépések vezetnek a megoldásig, TTCN3-kód értelmezéssel, a logikai felépítés megértésével, és a TtcnLogger osztály fontosabb függvényeinek bemutatásával.

A többi osztály közül az egész számok és a karakterláncok logolását emelem ki. Néhány java-ban írt példakóddal és magyarázattal fogom bemutatni a részfeladatok megoldását.
A fejezet végén pedig TTCN3-ban írt teszteseteket is bemutatok. A szakdolgozatomban ismertetett feladataim közé tartozik a konzolra és fileba logolás összehasonlítása. A legérdekesebb fejezet felhasználói szemszögből pedig a „mit lát a felhasználó” képekkel illusztrált alfejezet lesz.

A fejlesztésnek része a tervezés és a logikus gondolkodás. Ezeken kívül a tesztelés is egyre nagyobb szerepet játszik. Szakdolgozatomban a TDD-ről [4] és az agilis gondolkodásmódról is írok Dawn Haynes HUSTEF[5]-en elhangzott előadása alapján.

A kezdő fejezetben a Titan és a compilerek felépítéséről csak lényegileg számolok be, a fontosabb megértéshez elengedhetetlen fogalmak magyarázatával. A fordítóprogramok reguláris kifejezéseiről, gramatikákról és konkrét szabályokról szakdolgozatomban nem lesz szó. A formális nyelvek és automaták témakört sem érintem.

Az első fejezetben előjön az IOT[6], M2M[7] kommunikáció, amelyben bemutatom az Ericsson és a Sigma Technology által fejlesztett LegoTruckot, amely egy legóból készült autónak tűnik, de valójában sokkal több annál. Nem véletlen képezte az IOT technológia bemutatásának tárgyát a HUSTEFen, a Kutatók Éjszakáján[8] és különböző Ericssonos rendezvényeken.

[4] Test Driven Development
[5] Hungarian Software Testing Forum
[6] Internet Of Things
[7] Machine to Machine
[8] Ismeretterjesztő rendezvény Európa szerte

A „Titan, a compiler” című fejezetnek a kulcs alfejezetét a Titan előnyei című alfejezet fedi le. Itt utalok a TTCN-3 nyelvre, amelynek lényegi elemei a 2. fejezetben bontakoznak ki. Az olvasó választ kaphat a két leggyakrabban feltett kérdésre: mi az és hol használják. Ezután az első két fejezet összekapcsolódik és az „Alapvető típusok” fejezet elolvasásával az olvasó egy képet kaphat a kezdeti feladataim logikai gondolatmenetéről. A konkrét kódok githubon megtalálhatók[9].

Célom, hogy felkeltsem hallgatótársaim érdeklődését a szoftverfejlesztés iránt és az érdeklődőknek bemutassam, hogy mit is csinál valójában egy szoftverfejlesztő.

1.2 Az Eclipse Titan története

A Titan 2003 óta hivatalosan elfogadott és támogatott teszt eszköz az Ericssonban. Napjainkban már a legnagyobb német kutatóintézet, a Fraunhofer Fokus[10] is Titant és TTCN-3at használ az IOT biztonsági tesztelésre. Erről a Bosch Connected World 2018 IOT konferenciáján[11] volt szó.

A Titan Szabó János Zoltán Msc-s diplomamunkájaként indult a 2000-es évek elején, 15 év fejlesztés után 2015 elején volt az első open source release.

A Titan egy TTCN-3[12] és ASN.1[13] fordítóprogram[14]. Az Msc-s diplomamunka célja egy olyan futtatókörnyezet létrehozása volt, amely protokoll- és alkalmazásfüggetlen és a teljesítménytesztek végrehajtására is alkalmas. Ha a bemenet a szintaktikai és szemantikus ellenőrzés szerint is hibátlannak bizonyul, akkor a fordító C++ nyelvű programmodulokat generál, amelyek részei lesznek a végrehajtandó tesztsorozatnak. (SZABÓ J.Z., CSÖNDES T.[15])

---

[9] https://github.com/eclipse/titan.EclipsePlug-ins

[10] Frauenhofer Fokus – IOT szórólap: https://cdn0.scrvt.com/fokus/8b97fcd0ae224ca2/9694adadb60c/IoT_Flyer_NEU_140218_EN.pdf , letöltve: 2018. május

[11] Bosch IOT konferencia weboldala: http://bcw.bosch-si.com/berlin/, letöltve 2018. április

[12] Testing and Test Control Notation Version 3

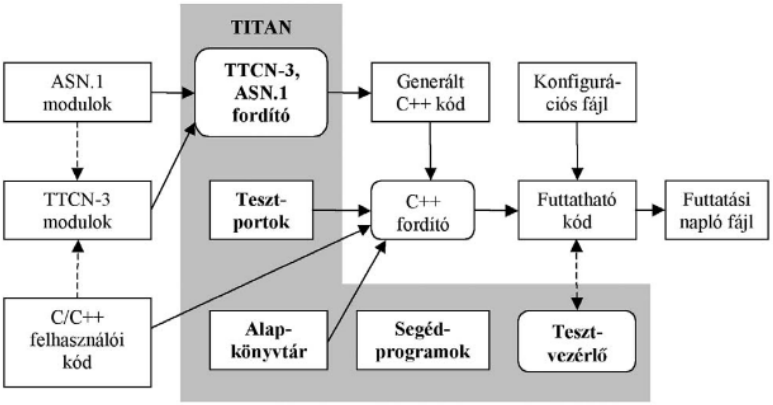
[13] ASN.1. = Abstract Syntax Notation, alkalmas adattípusok formális leírására (Forrás: http://www.hiradastechnika.hu/data/upload/file/2004/2004_08/HT0408-5.pdf, letöltve: 2018. május)

[14] Compiler, más néven fordítóprogram egy magasabb szintű nyelvet átkonvertál egy alacsonyabb szintűre (Forrás: https://www.techopedia.com/definition/3912/compiler, letöltve 2018. május)

[15] http://docplayer.hu/3597870-Titan-ttcn-3-tesztvegrehajto-kornyezet.html, letölve: 2018. február

A Titan részei

1. ÁBRA - TITAN BLOKKDIAGRAMJA



(FORRÁS: HÍRADÁSTECHNIKA CIKKJE, SZABÓ J.Z., CSÖNDES T.)

Alapkönyvtár: kézzel megírt, előre lefordított C++ kódok, TTCN-3 alaptípusait megvalósító osztályok

Tesztportok: külső világgal tartja a kapcsolatot, feladatuk az összeköttetés és a kommunikáció biztosítása a futtatható tesztsorozat és a tesztelendő rendszer között (a tesztelendő rendszer a külső világ eleme)

Segédprogramok: tesztfuttatást automatizáló script, formázó segédprogram

Tesztvezérlő: több párhuzamosan futó tesztkomponens összehangolása, a felhasználó láthatja a pillanatnyi állapotokat, végrehajtott műveleteket, szükség esetén beavatkozhat: új teszteset indítása, teszteset leállítása[16]

[16] Forrás: Szabó J. Z, Csöndes T. – Titan, TTCN-3 test execution environment cikkje alapján (http://www.hiradastechnika.hu/data/upload/file/2007/2007_1a/HT_0701a-6.pdf, letöltve: 2018. február)

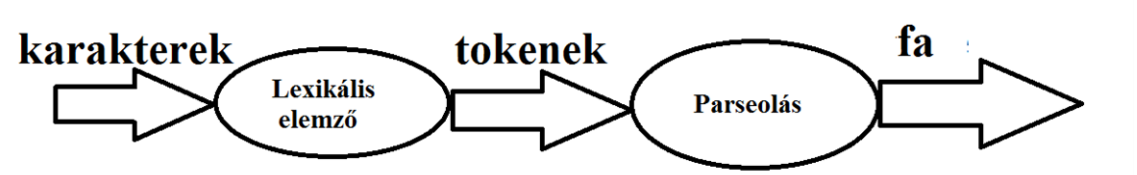
1.3 A Titan, mint compiler működési folyamata

A compilerek az adott programozási nyelv kódját fordítják át gépi kódba. A Titan esetében a TTCN-3 kódot C++-ra fordítják át, innen egy C++ fordító fejezi be a folyamatot. Szakmai gyakorlatom során abban a folyamatban veszek részt, amelyben amellé a fordító mellé, amely TTCN-3-ról C++-ra fordít, készül egy olyan fordító, amely TTCN-3-ról JAVA-ra tud fordítani.

A következőkben a compilerek működési folyamatát szemléltetem.

1.3.1 Lexikális elemzés és parseolás

2. ÁBRA – COMPILER FELÉPÍTÉSE 1.RÉSZ



Saját szerkesztés, (Sethi, R., Aho, A.V., Ullman,J.D., Lam, M.S., 2013 alapján)

A Lexikális elemző feladata, hogy a szöveget tokenek listájára bontsa fel. Példaként vegyük az osztályzatok tömböt, amelynek az "indexedik" elemét egy összeadással adjuk meg:

osztalyzatok[index] = 4+1

Ez összesen 12 (nem üres) karaktert tartalmaz, a lexikális elemző pedig 8 tokenre fogja felbontani az alábbi táblázat alapján:

1. TÁBLÁZAT - A LEXIKÁLIS ELEMZÉS SORÁN LÉTREJÖTT TOKENEK

osztalyzatok	azonosító
[	bal szögleteszárójel
index	azonosító
]	jobb szögleteszárójel
=	művelet,hozzárendelés
4	szám
+	pluszjel
1	szám

(SAJÁT SZERKESZTÉS, FORRÁS: LOUDEN, K.C. ÉS *SETHI, R., AHO, A.V., ULLMAN,J.D., LAM, M.S., 2013*)

A következőkben egy egyszerű TTCN-3 specifikus példát is bemutatok:

const integer x:=5

A fenti példát az alábbiak szerint bontja fel:

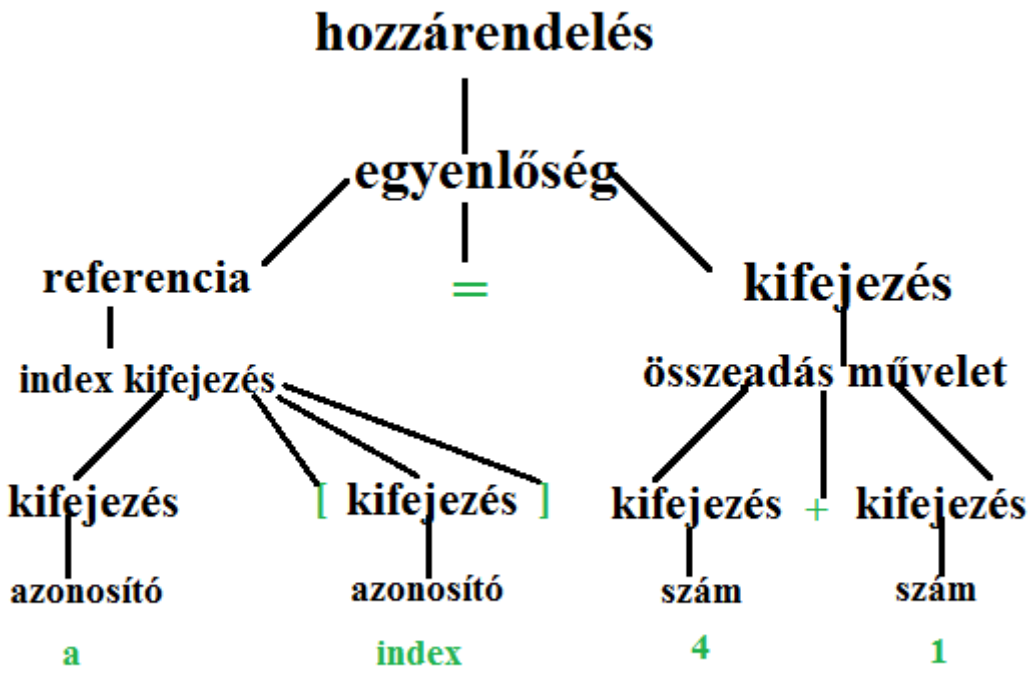
2. TÁBLÁZAT - TTCN-3 SPECIFIKUS PÉLDA A LEXIKÁLIS ELEMZÉSRŐL

const	integer	identifier	assign	number
keyword		x	keyword	5

SAJÁT SZERKESZTÉS (TTCN-3 COURSE PRESENTATION MATERIAL ALAPJÁN)

A lexer felismeri a kulcsszavakat. A token objektumokkal feltölt egy listát, ezt olvassa a parser. A parseolás során a szintaktikai elemző mintákat keres a tokenek között és felépít egy fát (AST-t[17]). A fa felépítése különböző alszabályok alapján történik.

3. ÁBRA - LEEGYSZERŰSÍTETT SZINTAKTIKAI FA AZ ELSŐ PÉLDÁHOZ



SAJÁT SZERKESZTÉS (LOUDEN, K.C. ÉS SETHI, R., AHO, A.V., ULLMAN, J.D., LAM, M.S., 2013 ALAPJÁN)

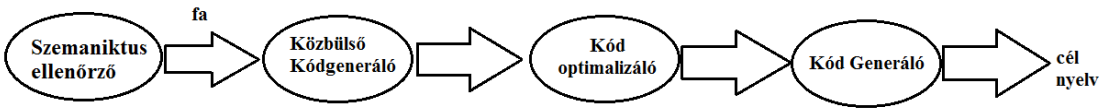
TTCN-3ban modul szinten definíciók vannak, pl. konstansok, module paraméterek, függvények. A JAVA oldalon használt ANTLR_LL elemzés szerint a modulban lévő elemek listát alkotnak, a „const” kulcsszó észlelésekor a konstans definíció szabályát

[17] AST=Abstract Syntax Tree

ráilleszti a konstans kulcsszóra. Vagy például amikor a „for” kulcsszó következik, az alszabályban rögzítve van, hogy a for(…) ciklus pontosan hogy fog kinézni. A for token alapján dönti el, hogy melyik szabályt kell alkalmazni a következő tokenekre.
A szintaktikai elemzés során egy absztrakt szintaxisfa készül el, amely a következő fordítási lépések alapja.

1.3.2 A szemantikus ellenőrzés

4. ÁBRA - COMPILER FELÉPÍTÉSE 2. RÉSZ



SAJÁT SZERKESZTÉS (SETHI, R., AHO, A.V., ULLMAN,J.D., LAM, M.S., 2013 ALAPJÁN)

A szemantikus ellenőrzésben például az adattípus ütközéseket szűri ki, megnézi, hogy egy tömb indexelésére tényleg egész számot írtak-e be. (SETHI, R., AHO, A.V., ULLMAN,J.D., LAM, M.S, 2013)
Egy másik példa, amikor a check függvényen keresztül ellenőriz, a checkUniqueness megnézni, hogy egyediek-e a nevek.

1.3.3 Kódgenerálás

Minden TTCN-3 és ASN.1 adattípushoz külön C++ típus (osztály) tartozik.

1.3.3.1 Futtatható tesztsorozat előállítása

make segédprogram segítségével
Makefile-t a Titan fordítóprogramja automatikusan előállítja
A Titan fordítója és a make segédprogram együtt inkrementális fordítást végez: csak azokat a szükséges fileokat fordítja le újra, amelyekben változás történt

1.3.3.2 Kódolás-dekódolás

A tesztelendő rendszernek küldött üzeneteket kódolni, az onnan fogadottakat dekódolni kell. (SZABÓ J. Z, CSÖNDES T.)[18]

[18] Forrás: Szabó J. Z, Csöndes T. – Titan, TTCN-3 test execution environment cikkje alapján (http://www.hiradastechnika.hu/data/upload/file/2007/2007_1a/HT_0701a-6.pdf, letöltve: 2018. február

1.4 Az Eclipse Titan projekt komponensei

3. TÁBLÁZAT - TITAN FŐ KOMPONENSEI

Titan Designer	szintaktikai, szemantikus ellenőrzés, maga a fejlesztőkörnyezet, kódkiegészítéssel, definícióra ugrással, a kinyert szemantikus információt hozzáférhetővé tudja tenni a ráépülő pluginok számára
Titan Executor	tesztfuttatás
Titan LogViewer	logolás megjelenítése
Titanium	kódminőségvizsgálat elvégzése (ehhez a Titan Designert használja)
Runtime library	olyan funkcionalitások gyűjteménye, amikre a futó kódnak szüksége lesz a működéshez és nem TTCN-3 kód függő
TTCN-3, ASN.1 compiler	szemantikus ellenőrző és C++ kódgenerátor

FORRÁS: SZABÓ,J.Z.,CSÖNDES,T.

Ezeken kívül fontos komponensek továbbá:

- xsd2ttcn: xsd dokumentumok fordítása TTCN-3 modulokká
- mctr_cli: a párhuzamos futtatás központi vezérlője
- makefilegen
- logmerge: logolási események összevonása
- logformat: logfileok formázása
- logfilter: log események beállítása alapján történő szűrés
- documentáció

A logmerge, logformat, logfilter a legenerált logfileokkal végeznek műveleteket.

1.5 Titan Előnyei

5. ÁBRA - AZ ECLIPSE TITAN LOGOJA

FORRÁS: PROJECT.ECLIPSE.ORG

Miért érdemes a Titant választani a TTCN-3 nyelv tesztvégrehajtó környezetének?

- A Titan olyan gyors, hogy tudja teljesíteni a 4G hálózatok követelményeit. A tesztelt rendszerüzenetek megérkezése, feldolgozása és megválaszolása között kevesebb, mint 4 ezredmásodperc telik el. A Titan sok extrával került ki az opensource szoftverek közé.
- Az, hogy OpenSource hatalmas előnyt jelent
    - gyakran előírás az állami és egyetemi szervezetek számára, hogy open source eszközöket használjanak
    - az ügyfelek teljes kontrollal rendelkeznek
- A license EPL [19] engedi, hogy másik kereskedelmi célból felhasználják (értékesítsék magát a terméket is)
- A Titan követi a szabvány változásait
- Professzionális supportot tud adni a termékhez, a nagy céges környezet miatt ismeri a céges környezetek gondolkodásmódját
- A Titan sok extrával került ki, pl. számos protokoll támogatására készített belső termék is ingyen elérhetővé vált
- Az Ericssonnál a Titánnal foglalkozó csapat folyamatosan folytat szoftver technológiai kutatásokat is

[19] EPL=Eclipse Public Licence

1.6 IOT/M2M kommunikáció

Az M2M a Machine to machine kommunikáció rövidítése. Gépet géppel köt össze emberi közreműködés nélkül. Lehetővé teszi az adatáramlást, Big Data elemzést és más felhő alapú szolgáltatások létrejöttét. Bármennyi eszközt hozzáköthetünk egy alkalmazás szerverhez, amely egy másik eszközön fut, így a munkafolyamatokat optimalizálni, gyorsítani lehet.
A logisztikában ez hasznos lehet áruk szállítási hőmérsékletének ellenőrzésére, robotok távfelügyeletére vagy optimális útvonalak meghatározására.

Az egészségügyben betegségmegelőzésre és távgondozásra használják.
A pacemakerek vagy az okos órák egészségügyi szerveknek vagy az internetre továbbítják az adatokat, annak érdekében, hogy a páciensek saját egészségügyi állapotukat nyomon tudják követni. Ez fiataloknak és időseknek egyaránt hasznos lehet.
A fiatalabbakat segítheti egy hatékony edzésterv kialakításában, az időseknek pedig a kontroll vizsgálat miatt nem kell felesleges időt töltenének a rendelőbe utazással és a várakozással. Sürgős esetben az eszköz azonnal tud jelezni a kórházban található rendszernek.

Nem csak az egészségügyben, hanem mindennapjainkban is egy nagyon hasznos technológia az M2M, ugyanis a távfelügyeletet az otthonunk biztonságának megtartása érdekében is igénybe vehetjük. Tűz esetén a füstérzékelők képesek azonnal riasztani a tűzoltóságot, betörés esetén a rendőrséget. Sőt a betörő pontos útvonalát is meg tudja adni, valós időben, ezen kívül kiegészíthető arcfelismerő szoftverrel is. Távolról lehet figyelni a berendezések állapotát, egy autó üzemanyagtartalmát, áruk szállítási hőmérsékletét.

Szerepet játszik a vásárlási szokások monitorozásában, képes rögzíteni, hogy az emberek milyen termékeket vásárolnak együtt. Például rengeteg üzlet törzsvásárlói-kártyát alkalmaz, hogy ezeket az adatokat begyűjthesse és tudja mely termékek árát mikor kell felemelni, leakciózni. Az M2M technológia képes rögzíteni, hogy mennyi pénz van egy automatában. Későbbi újítás lehet, hogy bevezetik ezt a pénztárgépeknél is, így nem lesz szükséges az eladóknak megszámolniuk a kassza tartalmát.

Az M2M segíthet az energiagazdálkodásban is. Képes feltérképezni az energiapazarlási szokásokat, felismeri a sokat fogyasztó berendezéseket és otthonunkban optimalizálni tudja a fűtés és meleg víz használatát is.

A Machine to machine kommunikáció a közlekedésben egyfajta biztonságot nyújt. Nem véletlen döntött úgy az Európai Unió, hogy 2015-től minden autóba M2M alapú segélyhívó rendszert szerelnek[20]. Nem csak baleset, veszély, meghibásodás esetén képes jelezni a központnak, az optimalizálást, a szabálybetartatás és a tájékoztatást is a feladatai közé tartozik. Az optimalizálás például a rövidebb utak megtalálása. A balesetekről, útlezárásokról történő tájékoztatás fontos az utasok és a sofőr oldaláról is, mert rövidebb várakozási időt és előretervezési lehetőséget biztosítana. A teljesítményadatok begyűjtése során rögzítik, hogy mennyi időt töltött egy adott busz a megállóban, hány utas volt, milyen gyorsan ment. Ez a flottamenedzsment fontos lépéseként a sofőröket a szabályok betartására kényszeríti, betartatja velük az optimális útvonalat, így elkerülhetik a dugókat, útlezárásokat. Buszok esetén a kevés utassal rendelkező járatok megszűntetése, átütemezése történhet meg. Ez a személy autóknál is nagy előrelépést jelent, segíti a tájékozódást, valós idejű GPS jelekkel látja el a sofőrt és bármilyen működési hiba esetén értesít. (Forrás: www.telekom.hu)

Az IOT technológia lényege: az internet of things kifejezést először Kevin Ashton használta 2009ben. Lényegében valós, létező eszközök adatcseréjéről beszélünk. Ez az adatcsere történhet egymás között vagy úgy, hogy valaki irányítja az eszközt. Tegyük fel, hogy az ébresztőóra előre tudná, hogy hány percet fog késni a vonat és e szerint ébresztene, képes lenne értesíteni a kávéfőzőt, amely automatikusan 15 perccel később készítené el a kávét munkába indulás előtt. A jövő megoldásának tűnik azonban az IOT technológiákkal ez könnyen elérhetővé válik a közeljövőben. (Forrás: hivemq.com)
Az említett funkciók közül a standardizáltakhoz tartozó standardizált teszteket TTCN-3ban írják[21]

[20] eCall automata segélyhívó berendezés, Forrás: http://www.karrendezes.eu/kotelezove-tette-az-eu-az-ecall-automata-segelyhivo-beepiteset/ letöltve 2018.március
[21] Intelligens szállítmányozási rendszerek: http://www.ttcn-3.org/index.php/downloads/publicts/publicts-etsi/65-publicts-its

http://www.ttcn-3.org/index.php/downloads/publicts/publicts-onem2m/128-publicts-onem2m-core

http://www.ttcn-3.org/index.php/downloads/publicts/publicts-etsi/67-publicts-epassport

http://www.ttcn-3.org/index.php/downloads/publicts/publicts-etsi/132-smartm2m-test-suites

1.6.1 IOT LegoTruck

6. ÁBRA - IOT LEGOTRUCK

FORRÁS: ECLIPSE.ORG/FORUMS/INDEX.PHP/T/1091027/

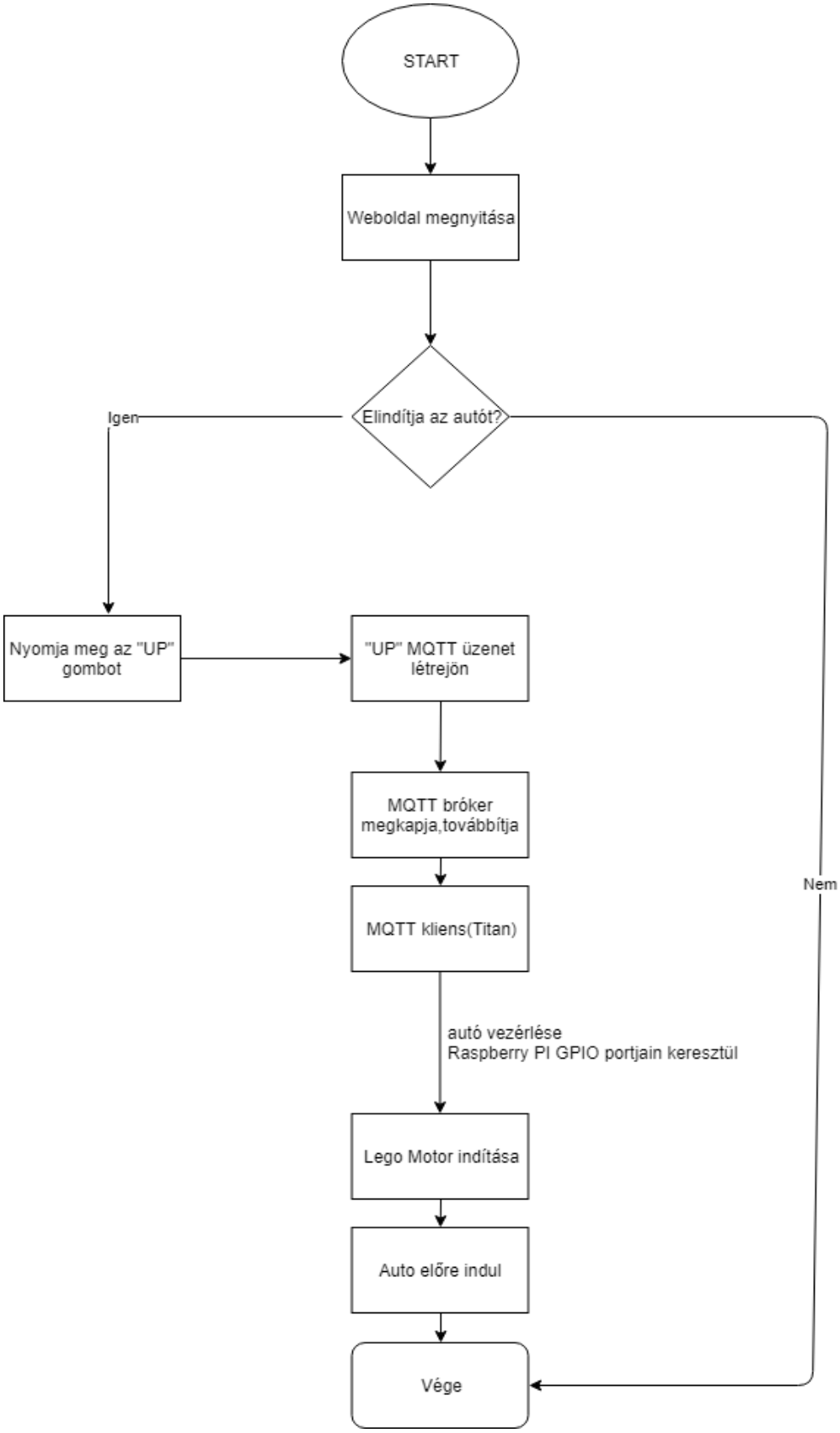
A Sigma Technology és az Ericsson közreműködése kapcsán született meg a Lego Autó vagy Legó Kamion, amelyet mobiltelefonról felhőn keresztül lehet vezérelni, az IOT technológián alapul. A mobiltelefonról egy weblapot érünk el, ahol az autó vezérléséhez szükséges gombok találhatók. A gombok megnyomására egy MQTT[22] üzenet keletkezik, amelyek az autó irányításában segítenek. Ez az üzenet fog eljutni az MQTT brókerhez. Az MQTT bróker felelős az üzenetek cseréjéért, hogy eljusson a feladótól a címzettig. A legó autó hátuljába Raspberry Pi 3 van építve egy powerbankkel együtt, amelyről az áramot kapja. Az eszközön Titan fut. Az MQTT brókerre a Titan fel van iratkozva, mint MQTT kliens. A Titan az MQTT üzenetek alapján vezérli az autó Lego motorjait a Raspberry GPIO portjain keresztül. A Raspberry Pi 3 számítógépen egy Raspbian nevű linux disztibúció fut.

7. ÁBRA - FÉNYKÉP AZ AUTÓ VEZÉRLÉSÉHEZ SZÜKSÉGES GOMBOKRÓL

FORRÁS: ECLIPSE.ORG/FORUMS/INDEX.PHP/T/1091027/

[22] MQTT=Message Queuing Telemetry Transport, melynek lényege: egy üzenetküldő protokoll, amellyel küldeni és fogadni lehet üzeneteket. Kis sávszélességet használ. Andy Stanford-Clark és Arlen Nipper fejlesztette ki 1999-ben.

LegoTruck elindításának flowchartja



2 TTCN-3

2.1 TTCN-3 nyelv ismertetése

A következőkben szeretném kifejteni, hogy mi az a TTCN-3 és mivel ad többet a hagyományos programnyelveknél és a script nyelveknél. [23]

- Erősen típusos nyelv, amely támogatja az altípusokat
- Mintaillesztési mechanizmusok (matchelés)
- Snapshotok kezelése (port, timeout)
- Eseménykezelés: verdictek (ítéletek) megjelenése, kiértékelése
- Timerek (időzítők indítása, időzítő lejárta)
- Párhuzamos futás
- Futás idejű teszt konfiguráció: futás alatt verdictek megjelenése, kiértékelése a teszt sikeres/sikertelen futásáról
- Dinamikus tesztek konfigurálása
    - egymástól független tesztkomponensekkel
    - külön tesztesetek kezelése: az adott részegység tesztelésre összpontosító tesztek

2.1.1 Mi az a TTCN3?

A TTCN-3 a Testing and Test Control Notation version 3 rövidítése. Ez az egyetlen szabványos tesztelési nyelv. A TTCN-3 programozási nyelvet az ETSI (European Telecommunication Standards Institute) szabványosította. Nem objektum orientált. Alapvetően protokollok tesztelésére találták ki. A protokoll olyan szabályok összessége, amely a különböző rendszerek közötti kommunikációt szabályozza. „A protokollok határozzák meg azokat a szabályokat, amelyek szerint a rendszer elemei egymással kommunikálnak, információt cserélnek. Tehát a megfelelő protokollokat minden rendszerelemnek ismernie kell.” [24] (DIBUZ, S, CSOPAKI, GY) A TTCN-3 tesztnyelv képes megvizsgálni, hogy rendszerünk hasonlít-e egy adott protokoll szabályrendszerből felépített modellre. Ez a konformancia tesztelés során nagyon hasznos, mivel ott pont azt vizsgálják, hogy a rendszer eleget tesz-e a specifikációnak, követelményeknek, szabványoknak. (DIBUZ, S, CSOPAKI, GY)

---

[23] http://www.ttcn-3.org/index.php/downloads

[24] http://epa.oszk.hu/00600/00691/00043/pdf/868-872.pdf

2.1.1.1 **Erősen típusos nyelv, amely támogatja az altípusokat**

Az alap típusok, pl. az integer, float, string bármely programozási nyelv építőkövei. Ezt terjeszti ki a TTCN-3 olyan beépített típusokkal, mint a bitstring, octetstring. Ezzel megkönnyíti a felhasználók munkáját, mert sokszor előfordul, hogy a hálózati kommunikáció nem bitek vagy karakterek szintjén történik. Sok protokol például octetekben írja le, hogy mit kell csinálni. A TTCN-3 közvetlen a nyelvből támogatja ezeket a típusokat. A verdict egy fontos nyelvi elem egy tesztelésre kitalált nyelvben. A konstruált típusokra (record, set, record of, set of, komponens, port) azért van szükség, mert így a felhasználók egy tetszőlegesen komplex típusú adatstruktúrát fel tudnak építeni és utána tudnak dolgozni velük.

A subtyping vagy „altípus” a típusok megszorítását jelenti. Nem minden értéket fogadunk el, hanem csak azt, ami a megszorításnak megfelel.
Például a 0 és 255 közé eső számokat a „type integer SzamHalmaz(0..255)”-tel érhetjük el, a minimum 4, maximum 8 karakter hosszú bitstringeket a „type bitstring MyBits length(4..8)” kódsorral.
Mintával is lehet megszorítani, pl.: type charstring MyString(pattern ”abc*xyz”);
A típusokat keverni is lehet: type charstring MyCharStr5 ("gr", "xyz") length (1..9);
A subtypeok komplex típusokra pl. recordokra is használhatók a következő módon:

```
type record Pelda{
integer field(1..10),
charstring field2 optional
}
```

Type Pelda Pelda2({1,omit},{2,”nev”}, {1,”szemelyiszam”}); esetben a record típus 3 féle értéket vehet fel.

2.1.1.2 **Mintaillesztési mechanizmusok**

A TTCN-3 olyan mintaillesztési mechanizmusokkal rendelkezik, amelyek segítségével üzenetküldés esetén meg tudja nézni, hogy az adott üzenet illeszkedik-e egy adott mintára például a match függvény segítségével. Ez alapesetben egy változó értékét hasonlítja össze egy template-tel, a zárójelbe összetettebb kód esetén reguláris kifejezések is kerülhetnek.

Jelölés: match(változó,template).

A képen egy matchelés eredménye látható:

```
15:44:06.769000 "szakdolgozat" with "egyetem" unmatched
15:44:06.769000 "szakdolgozat" with "szakdolgozat" matched
15:44:06.769000  matched
15:44:06.769000 .charrec := "a" with "b" unmatched
15:44:06.769000 Execution of control part in module LoggingTitanChar finished.
```

2.1.1.3 **Verdictek megjelenítése, kiértékelése**

A tesztelés után ha tesztcase-ekben egy verdictet, vagy magyarul ítéletet kapunk aszerint, hogy a tesztünk az elvárt módon viselkedett-e. (ERŐS, L PUBLIKÁCIÓJA ALAPJÁN)

```
Verdict Statistics: 0 none (0 %), 3 pass (75 %), 0 inconc (0 %), 1 fail (25 %), 0 error (0 %)
Test execution summary: 4 test cases were executed. Overall verdict: fail
```

Pass: elvárt módon viselkedett

Fail: Tesztünk valamelyik követelménye nem teljesül

Error: Futás idejű hiba

None: az ítéletnek nincs értéke

Inconclusive: a teszt célja nem teljesült.

2.1.1.4 **Snapshotok kezelése**

A snapshotokat a jobb megértés érdekében úgy magyarázzák el, hogy képernyőkép készül az adott pillanatban a rendszer konfigurációjáról. Valójában nem készül kép, csak így könnyebb elképzelni. A rendszer konfigurációjának adott értéke nem változik meg ténylegesen, amíg el nem döntjük hogy melyik ágon kell továbbhaladni.

A szerveren futó adatok rögzítésre kerülnek a konfigurációs fileban. Az alábbi konfigurációs file nem kapcsolódik [25] a Titanhoz. Azért illesztettem be belőle egy

8. ÁBRA - HOGY NÉZ KI EGY CONFIG FILE (NEM TITAN SPECIFIKUS)


```
{
    "fileVersion": "1.0",
    "requestId": "asudf8ow-4e34-4f32-afeb-0ace5bf3trye",
    "configurationItems": [
        {
            "configurationItemVersion": "1.0",
            "resourceId": "vol-ce676ccc",
            "arn": "arn:aws:us-west-2b:123456789012:volume/vol-ce676ccc",
            "accountId": "12345678910",
            "configurationItemCaptureTime": "2014-03-07T23:47:08.918Z",
            "configurationStateID": "3e660fdf-4e34-4f32-afeb-0ace5bf3d63a",
            "configurationItemStatus": "OK",
            "relatedEvents": [
                "06c12a39-eb35-11de-ae07-adb69edbb1e4",
                "c376e30d-71a2-4694-89b7-a5a04ad92281"
            ],
            "availibilityZone": "us-west-2b",
            "resourceType": "AWS::EC2::Volume",
            "resourceCreationTime": "2014-02-27T21:43:53.885Z",
            "tags": {},
            "relationships": [
                {
                    "resourceId": "i-344c463d",
                    "resourceType": "AWS::EC2::Instance",
                    "name": "Attached to Instance"
                }
```


FORRÁS: DOCS.AWS.AMAZON.COM

részletet, hogy el lehessen képzelni körülbelül hogy néz ki egy konfigurációs file.

A Titánnál konkrétan ez hogyan működik abba nem láttam bele, főként portokat és timereket lehet könnyen kezelni a TTCN-3 nyelv segítségével, úgyhogy valószínűleg ezeknek az adatai szerepelhetnek a rendszer konfigurációjában.

[25] https://docs.aws.amazon.com/config/latest/developerguide/example-s3-config-history.html

A TTCN-3 nyelvben konkrétan az „alt”-ot és az „alt step”-et használják a snapshotok kezelésére. Párhuzamos folyamatként definiálhatóak, az összes kommunikációs portnak és az időzítőknek is rögzítik az állapotát (úgy kell elképzelni, mintha képernyőkép készülne róluk). Az altot a beérkezett üzenetek illeszkedésének ellenőrzésére is használják, például ha az alábbi sor ([] Port4.receive(charstring:?)….) egy alt belsejében van, annyit jelent, hogy ellenőrzi, hogy a 4es porton karakterlánc típusú üzenet érkezett-e be. Az alt ágaiban különböző feltételeket lehet definiálni.

2.1.1.5 **Párhuzamosság**

Több egymástól független folyamat is dolgozik a rendszerben. Például ha a tesztemben elindítok 10 párhuzamos komponenst, akkor a Titan minden párhuzamos komponenst egy vagy több különböző gépen indít el. Ezáltal sokkal nagyobb adatmennyiséget lehet fogadni és küldeni, több erőforrást lehet bevonni a tesztelésbe vagy ellenőrizni lehet, hogy 1-1 szimulált kapcsolat lassulása befolyásolja-e a többi kapcsolat minőségét. A párhuzamosan futó komponensek egymástól függetlenül is tudják végezni a feladatukat. Ha egy eszköznek különböző felületeket kell tesztelnie, akkor sokkal egyszerűbb mindegyikre csinálni egy komponenst ami csak azzal az adott feladattal foglalkozik, mint egy olyat amely mindenhez ért.

2.1.1.6 **Timerek**

Timert akkor használunk, amikor szeretnénk eldönteni, hogy a tesztelt rendszer megfelelően működik-e. A tesztelő üzenetet küld a tesztelt rendszernek és a kapott válaszból eldönti, hogy jól vagy rosszul működik a tesztelt rendszer.
Ha nem várt választ kapunk, akkor az ítélet fail lesz.
Ha nem kapunk az adott időn belül választ, timeout esemény keletkezik és az ítélet inconclusive lesz, azaz nem tudjuk eldönteni, hogy jól vagy rosszul működik a tesztelt rendszer. (ERŐS, L)

Timeout esemény: az üzenet elküldése előtt elindítjuk a timert(Timer.start(3)) és az adott időn belül (3 másodperc) nem kapunk választ a tesztelt rendszertől, akkor timeout esemény keletkezik. Ezt egy alt szerkezetben kezeljük le, itt állítjuk be az inconc ítéletet. A timer időbeállítását megadhatja a felhasználó a config fileban modulparaméterként, és ezt adjuk a timer értékéül vagy a tesztelő is beállíthat neki értéket. (ERŐS, L)

2.2 Hol használják a TTCN-3 nyelvet?

A legnagyobb német kutatóintézet, a Fraunhofer Fokus is TTCN-3at és Titánt használ az IOT biztonsági tesztelésre.
A Bosch Connected World 2018 IOT konferencián is kielemelték a tesztelés fontosságát mind üzleti, mind biztonságkritikus területeken.

A TTCN-3 nyelvet használják a teljes 4G-5G hálózatok tesztelésére. Ezen kívül használják még például biztonsági tesztelésekre is. Konkrétabban:

- Az armour (EU szintű) project IOT eszközök biztonságának tesztelésére[26]
- Mobil és vezetékes kommunikáció (LTE, WiMAX, 3G, GSM, ISDN, SS7)
- TETRA biztonsági rendszer (amely a katasztrófavédelmet, rendőrséget segíti) tesztelése
- szélessávú technológiák (ATM, DSL)
- Middleware platformok (WebServices, CORBA, CCM, EJB)
- Internet protokolok, IP alapú hálózatok és alkalmazások (SIP, IMS, IPv6, SIGTRAN, XMPP, SOAP és REST alapú szervizek stb.)
- Okos kártyák, ePassport
- Autóiparban a jármű biztonságának tesztelésére kapcsán a fedélzeti szoftverek tesztelése (AUTOSAR, MOST, CAN)
- oneM2M kapcsán – IOT eszközökhöz írt tesztek
- Intelligens szállítmányozasi rendszerek (ITS) tesztelésére[27]

[26] https://www.armour-project.eu/wp-content/uploads/2016/08/D22-Test-generation-strategies-for-large-scale-IoT-security-testing-v1.pdf
[27] http://www.ttcn-3.org/index.php/component/taxonomy/ITS

2.3 TTCN-3 előnyei

- Újrafelhasználhatóság
- Könnyen megtanulható
- Egyértelmű jelölések
- Különböző teszt toolok használják (pl.Titan)
- Open source közösség kialakulása (eszközök, hasznos modulok vizsgálata)
- Szabványosítási tesztek meghatározására használható
- Rugalmas (skálázható – engedi a tesztrendszerek növekedését)

A szabványosítás egyértelműsíti az elvárásokat, a tesztelők a specifikusan nekik készített eszközökkel tudnak tesztelni. Az újrafelhasználhatóság vagy újrafelhasználás üzleti értéke abban rejlik, hogy egy nemzetközileg elfogadott tesztsorozatot futtatnak le az eszközökre. Egy telekom szolgáltató mielőtt megvesz egy új eszközt minden lehetséges módon tesztelnie kell azt. Ha a kapcsolat során 10 eszköznek kell együttműködnie, akkor 10! tesztet kellene végrehajtani. Ez fizikailag lehetetlen, főleg, hogy mindennek együtt kell működnie egymással. A globális tesztekkel ezt nagyságrendileg csökkenteni lehet. Ezen kívül beszállító választásánál is hasznos egy adott teszthalmaz alkalmazása, mivel ugyanazt a tesztsorozatot lefuttatják az eszközökre, így könnyen össze tudják hasonlítani őket. A tesztelés költsége nem nő egy új beszállító megjelenésével. Ez ugyanúgy a beszállítóknak is hasznos, mert tudják előre, hogy az eszközöknek milyen teszteken kell átmenniük. Itt ecosystem szinten történik az újrafelhasználás a különböző cégek között. A szabványos teszteknek nagyobb az előállítási költsége, mint a házon belüli teszteknek, viszont ez a költség szétoszlik az összes résztvevő között. A teszteket ugyan az összes érdekelt félnek el kell fogadnia, de még mindig hatékonyabb a szabványos teszthalmazt alkalmazni, minthogy minden résztvevő külön-külön kifejlessze a tesztrendszereket.

2.4 Szintaktika és egyszerű példák

A TTCN-3ban modulokban kell kódolni és a file végén szükség van egy control{} részre. A testcase-eket a control törzsében a következőképpen tudjuk futtatni: execute(testcaseneve());

2.4.1 Változó deklarálás

A TTCN-3 tesztnyelvben változót a következőképpen deklarálunk:

var típus változónév := változóértéke

var boolean Dontes := false;

const[28] integer PeldaKonstans := 7;

Az értékadás = helyett :=-vel történik, az összehasonlítás a JAVAhoz hasonlóan ==-vel. Fontos, hogy TTCN-3-ban változónak értéket csak **control, testcase, function**, vagy **altstepen** belül adhatok.

2.4.2 Ciklusok

Ugyanúgy, mint a többi programozási nyelvben a TTCN-3ban is szerepelnek az alap ciklusok (for, while, switch, select, if).

2.4.3 Adattípusok

A TTCN-3 nyelvben megjelenik az integer, boolean,string(charstring,octetstring,hexstring). A charstring például csak karaktereket, míg a hexstring 0-Fig 4 biten fogad értékeket. A tesztelésnél fontos, hogy a küldő port mit fogad.

A **record és set** vizsgálatánál egy táblázatot kell elképzelni, közöttük a különbség, hogy a recordnál fix a sorrend, míg a setnél tetszőleges. Amikor a recordnál felsorolom milyen típusúak lehetnek a mezők, azt a sorrendet kell tartanom. Ha az első mező int, akkor oda mindenképpen egy egész számot fog várni. A **record of** és a **set of** lista jellegű típusok. Az **union** lényege, hogy csak az egyik van jelen a felsorolt típusokból.

A **type component** egy fizikai számítógépet reprezentál. A **testcase**-t kívülről is lehet hívni, ez jelzi hogy itt egy teszt fog következni, amelynek lesz egy saját verdictje, azaz eldől, hogy helyesen futott-e le (pass) vagy nem (fail). Testcase-eken kívül például a control is futtatható kívülről.

---

[28] Konstansok esetén a var kulcsszó elé a const kerül.

3 Alapvető típusok

A runtimeban előforduló típus és belőlük példányosodó érték vagy template struktúrák a kompozit tervezési mintát[29] követik. Minden TTCN-3 típusból lesz egy osztály, amik így hierarchiát alkotnak.

Mindegyik osztályban egyenként meg kell írni a műveleteket, például a TitanBoolean osztályban a true és false logikai műveletek között létrejöhet „és” kapcsolat. Az „and” függvényben megvizsgáljuk, hogy a változó kapott-e értéket és ha igen,akkor &&-t fog visszaadni, a „vagy” függvény esetében || -t a két érték között.

3.1 Alapműveletek kódolása TitanInteger osztályban

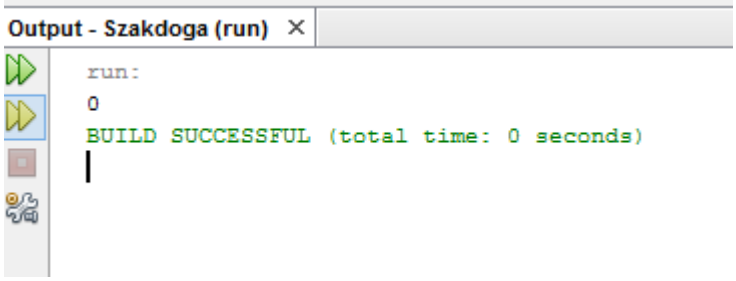
A TitanInteger osztályban megjelenik az összeadás, kivonás, szorzás, osztás és a különböző operátorok: <, <=,>,>=,=,!=.

Első feladataink egyike ezek implementálása volt. Integerek esetén vizsgálni kellett, hogy NativeInteger vagy BigInteger. Ha a minimum (Integer.MIN_VALUE) és a maximum (Integer.MAX_VALUE) intervallumon kívül esik akkor BigInteger. A műveleteket külön implementálni kellett int, TitanInteger és BigInteger bemenetre.

Az első nagyobb feladatom a MOD elkészítése volt. A titan.core core mappájából kikerestem a hozzá tartozó C kódot. Az implementáláshoz először át kellett gondolnom, hogy mi is az a mod. Maradékos osztást jelent.

Egy egyszerű példán szemléltetve, 10et ha 5tel elosztom 0-t kell kapnom eredményként, ugyanis ennyi a maradék. Ezt az egyszerű példát a könnyebb megértés segítése érdekében NetBeansben írtam meg és valóban 0-t kaptam megoldásként.

9. ÁBRA - NETBEANSBEN MEGÍRT EGYSZERŰ PÉLDA



[29] https://sourcemaking.com/design_patterns/composite

A kód a következő volt:

```
public static int modm(){
   int osztando=10;
   int oszto=5;
   int eredmeny=osztando%oszto;
   return eredmeny;
}
```

A Titánban létrehozott mod függvény implementálásához a következőeket kellett átgondolni: A moduláris osztás függvénynek TitanInteger típusa lesz és fog kelleni két változó, amellyel az osztást szemléltetem. A példámmal összehasonlítva a hivatalos kódot az osztandó lesz a leftValue és az osztó pedig a rightValue.

Először a rightValuet azaz az osztót vizsgáljuk egy IF(ha) függvény segítségével. **if(rightValue<0)** Hogyha ez kisebb, mint nulla, tehát a feltétel értéke igaz, akkor negatív számmal kell tovább számolnia a programnak. Így az if függvény törzsébe a **rightValue=rightValue*(-1);** kerül.

Nullával nem lehet osztani, emiatt került be az else if feltételvizsgálatához az, hogyha az osztó nulla, akkor hibaüzenetet írjon ki a program. Ez egy úgynevezett Dinamic Testcase Error: **throw new TtcnError("The right operand of mod operator is zero.");**

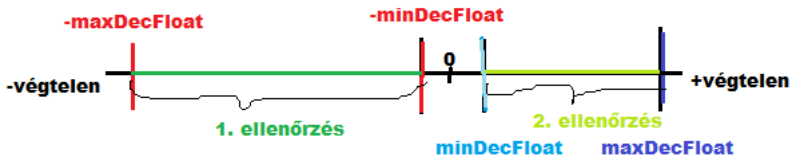
Értékek összehasonlítását javaban mindig dupla egyenlőségjellel vizsgáljuk. Miután megvizsgáltuk és lekezeltük azokat az eseteket, amikor az osztó kisebb vagy egyenlő, mint 0, következhet az osztandó vizsgálata, ahol hasonlóképp jártunk el. Ha az osztó és az osztandó is nagyobb nullánál, elvégezzük a maradékos osztást. Ha kisebb venni kell az értékek „nativeInt”-jét. Integerek esetén NativeIntbe tároljuk a számot, ha a minimum (Integer.MIN_VALUE) és a maximum (Integer.MAX_VALUE) érték közé esik. Ha az intervallumon kívül esik, akkor BigInteger.

Ha az osztandó pontosan 0, akkor a hányadoshoz hozzá kell adni az osztót, hogy megkapjuk a maradékot.

Hasonló módon kellett átgondolni és megírni a többi függvényt is.

3.2 TitanFloat logolásának intervallumellenőrzése

10. ÁBRA - INTERVALLUMELLENŐRZÉS



A TitanFloat logolása közben előjött intervallumellenőrzést a log_float() függvénnyel végezzük. Az ábrán látható ennek a logikai gondolatmenete. Az összes lehetőséget vizsgálni kell. A vizsgálat célja, hogy a végtelen és a „nem szám” eseteket jól tudjuk kiírni. Először a két zöld intervallumot kell figyelni, az 1. intervallumba azok az értékek kerülnek amelyek a -MAX_DECIMAL_FLOAT és a -MIN_DECIMAL_FLOAT közé esnek, a 2. ellenőrzésnél vizsgálni kell, hogy a pozitív min_dec_float és max közé essenek. Majd megnézzük azt az esetet amikor a float_val a 0-t veszi fel. Ha egyik variáció sem teljesült, akkor a float szám lehet mínusz vagy plusz végtelen. Egy érdekes vizsgálat, hogy a double típusú float értéket saját magával hasonlítjuk össze annak érdekében, hogy megtudjuk, hogy tényleg float érték-e. A min decimal float értékét 1.0E-4-re tároljuk. A maximumét pedig 1.0E+10 -re. Az említett vizsgálatokat IF és elseIF szerkezettel végeztük.

3.3 TitanCharString

- TitanCharString()
- TitanCharString(String)
- TitanCharString(StringBuilder)
- TitanCharString(TitanCharString)
- append(String) : TitanCharString
- append(TitanCharString_Element) : TitanCharString
- assign(Base_Type) : TitanCharString
- assign(String) : TitanCharString
- assign(TitanCharString) : TitanCharString
- assign(TitanCharString_Element) : TitanCharString
- assign(TitanUniversalCharString) : TitanCharString
- cleanUp() : void
- concatenate(String) : TitanCharString
- concatenate(TitanCharString) : TitanCharString
- concatenate(TitanCharString_Element) : TitanCharString
- concatenate(TitanUniversalCharString) : TitanUniversalCharString
- constGetAt(int) : TitanCharString_Element
- constGetAt(TitanInteger) : TitanCharString_Element
- copyValue(String) : void
- copyValue(StringBuilder) : void
- decode_text(Text_Buf) : void
- encode_text(Text_Buf) : void
- getAt(int) : TitanCharString_Element
- getAt(TitanInteger) : TitanCharString_Element
- getValue() : StringBuilder
- isBound() : boolean
- isPresent() : boolean
- isValue() : boolean
- lengthOf() : TitanInteger

A TitanCharString osztály lényegében a képen található függvényekből épül fel. Az operator+ a concatenate, az operator += az append.

Az osztályban a val_ptr StringBuilder típusú változót használjuk. A StringBuilder egyfajta módosítható karakterlánc, könnyen lehet módosítani tartalmát vagy hozzáírni ahhoz. Az append metódussal bármit hozzá lehet fűzni a StringBuilderhez. Ezt fogjuk használni, a concatenate esetén a string végéhez fűz hozzá, az append esetén szintén hozzáfűz a string végéhez, de felülírja (updateeli) az objektumot az összefűzött értékre.

4 Loggolás

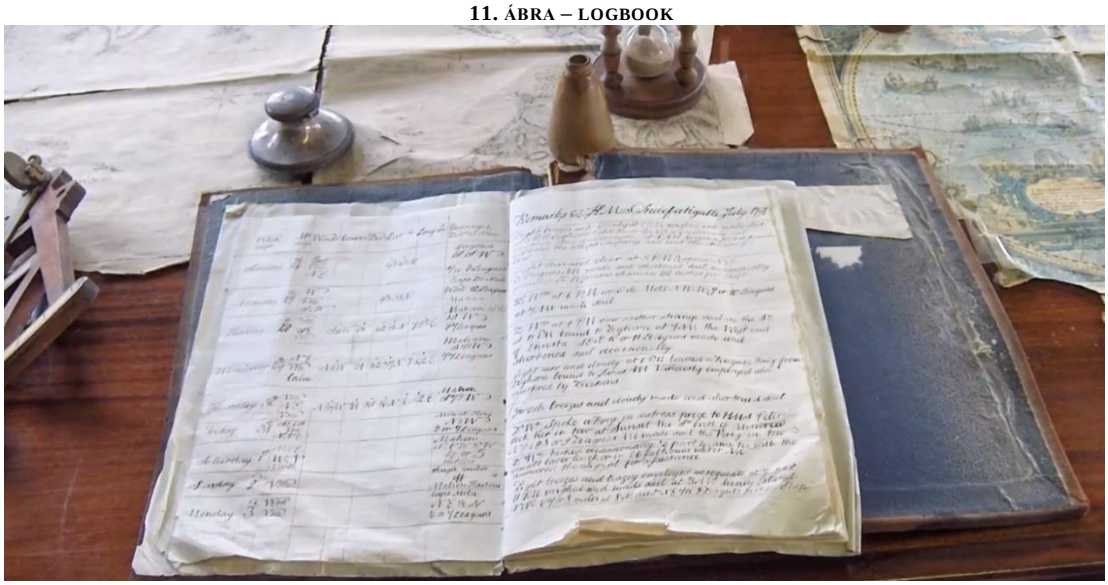
4.1 A loggolás alapjai

Szakdolgozatom ezen fejezetében szeretném bemutatni, hogy egy adott teszt futtatásánál létrejött lognak milyen elemei vannak, hogyan épül fel és hogy egyáltalán miért kell loggolni. A TtcnLogger osztály implementálási lépéseiről, teszteléséről is beszámolok. A logolás történelmi hátterével kezdem, amely a való életből vett példákkal szemlélteti azt, hogy honnan ered a logolás, régen milyen esetekben használták és a mindennapi életünkben hogyan jelenik meg.

4.1.1 Történelmi háttér

A logolást először hajósok használták. Egy könyvet vezettek, amelybe időrendben feljegyezték a különböző fontos eseményeket, minimum naponta egyszer. Feljegyezték, többek között azt, hogy egyenlő időközök elteltével mekkora távolságot tett meg a hajó. Manapság már egyre több mindent beleírnak, például az időjárást is.

11. ÁBRA – LOGBOOK



FORRÁS: HTTPS://APIRATESPORTAL.COM/2012/10/14/THE-CAPTAINS-LOG/

Az úgynevezett „LogBook” fontosabb elemei:

- Események, vészhelyzetek leírása, kezelése
- A hajó pontos pozíciója
  (a szélességi és a hosszúsági körök kiszámolásával, átváltásával)
- A pontos dátum, az eltelt idő
- Időjárás, adatok a tenger állapotáról
- A hajó mozgásának módja

- Rendellenességek a hajón
- Sebesség (csomókban)
- Balesetek esetén feljegyzett különböző intézkedések

4.1.2 A loggolás napjainkban

A loggolás a hajózáson kívül más területeken is megjelenik a mindennapi életünkben. Az alábbi alfejezetben ezeket a különböző területeket mutatom be.

4.1.2.1 **LogBook a légi közlekedésben**

A repülőgépen a pilóták is használnak „LogBook”-ot. Mind a mai napig léteznek különböző hivatalos kitöltési útmutatók[30] és szabványok[31], amelyekben azt ajánlják, hogy a rengeteg applikáció ellenére, mindenképp érdemes tartani egy saját papír alapú logot is. Az alapvető személyes adatok mellett feljegyzik például a dátumot, a felszállás és a landolás idejét, a repülőgép típusát, az utazás időtartamát, valamint megjegyzéseket is fűzhetnek az adott repülőúthoz. A lap alján mindegyik oszlopot összegzik, így könnyen lehet látni a teljes repülési időt is. Ez azért is fontos, mert a repülési időre gyakran rákérdeznek a pilóták állásinterjúján.

A kézi log megjelenése óta már rengeteg elektronikus loggolási metódust, programot létrehoztak a logok rögzítésére. Könnyebben kereshetővé, átláthatóbbá tették az események vizsgálatát. Gyorsabban lehet rögzíteni a bejegyzéseket és később különböző feltételek alapján rendezni őket. A manapság használt szoftverek közül néhány, amelyet a loggolás vezetésére használnak: LogBook Suite, Spot2Log, Vlog, AirplaneManager, eDriver (logisztikában).

---

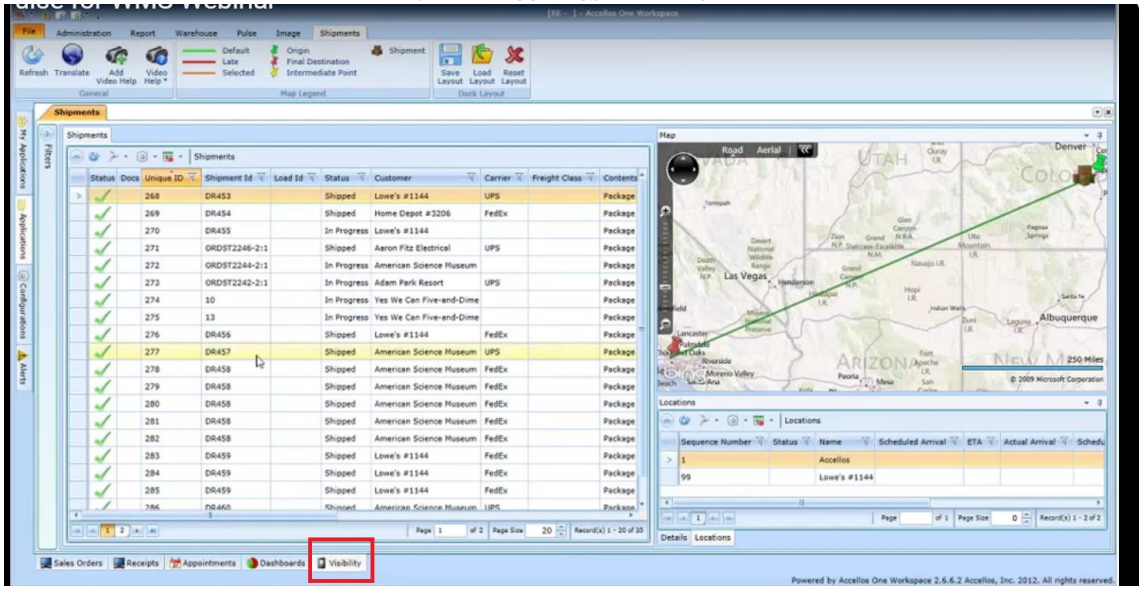
[30] https://dutchpilotgirl.com/pilot-logbook/

[31] http://rgl.faa.gov/regulatory_and_guidance_library/rgfar.nsf/daa4c54debeb6dca86256f3400626ab0/0367625a797466aa8625793b0068f34d!OpenDocument
http://clayviation.com/2017/02/22/paper-vs-electronic-flight-logbook/

4.1.2.2 Az Elektronikus Log előnyei és hátrányai a logisztikában

Nem csak a pilóták és a hajósok használnak logolást, hanem például a logisztikában vagy egy egyszerű beléptető rendszernél is megjelenik. A logisztikában különböző készletnyilvántartó, raktár menedzsment rendszereket használnak, amelyek például a sofőr útvonalát is képesek lekövetni.

12. ÁBRA - ACCELLOS KÉPERNYŐKÉP



FORRÁS: ACCELLOS.COM

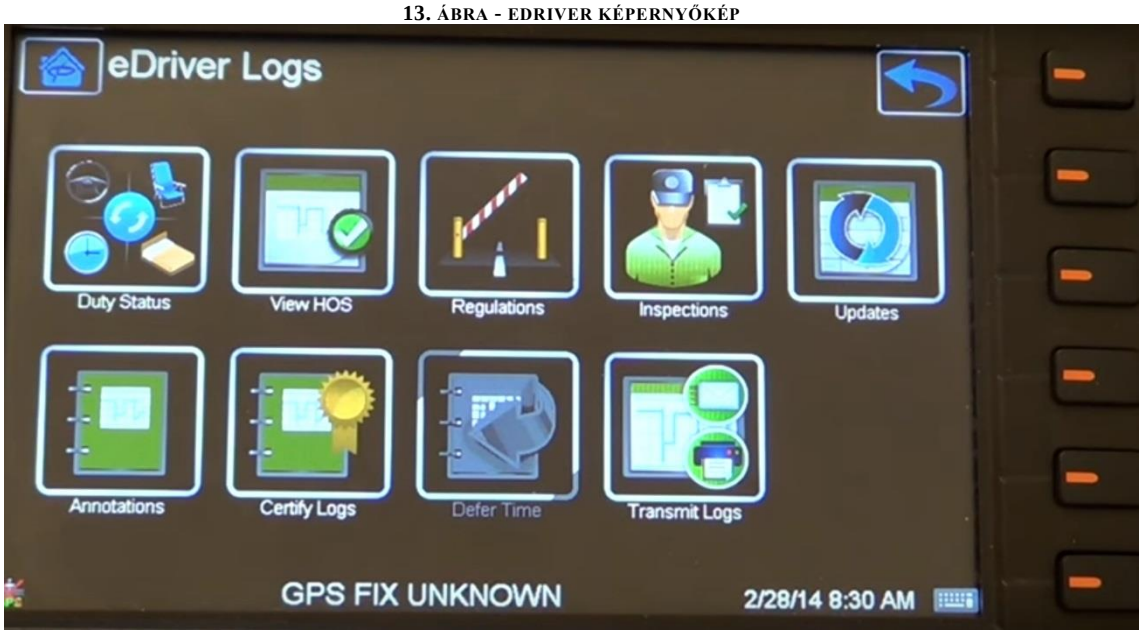
2009 decemberétől egyre többen mondtak búcsút a papír alapú loggolásnak (truckinginfo.com)[32]. A sofőröknek saját elmondásuk szerint több idejük maradt és jobban meg tudták tervezni a napjukat. A logoláson kívül sok egyéb funkciót be lehet építeni a rendszerbe: ilyen például a vezetők közötti kommunikáció, térképek, GPS-ek kezelése, szállítólevelek küldése, fék vagy motorhibák jelentése.

Az E-Logbookokat lehetetlen átverni, figyeli az útvonalat, ezáltal a kitérőket is, csak a tényleges munkáért fizetik a kamionsofőröket. A sofőröknek is hasznos, mert a teljes napi menetrendjüket látják előre és a teljesítményükön is javíthatnak, mert a rendszer tárolja az előző hónap teljesítményét. Rolf Lockwood a trunkinginfo.com „The Dark Side of Electronic Logging Devices" cikkjében[33] felhívja a figyelmet az E-log árnyoldalára is. Véleménye szerint mindenkinek megvan a saját életritmusa, de a rendszer bevezetésével

[32] Electronic Logs: The End of the Comic Book?
http://www.truckinginfo.com/channel/fleet-management/article/story/2009/12/electronic-logs-the-end-of-the-comic-book.aspx
[33] http://www.truckinginfo.com/channel/drivers/article/story/2016/05/commentary-the-dark-side-of-electronic-logging-devices.aspx

mindenkit belekényszerítenek egy adott sémába, sokan törvényt szegnek (pl. a parkolási szabályokat nem veszik figyelembe) hogy a megadott időben végezzenek egy adott feladattal. A cikk arra utal, hogy egyénre szabott ütemtervet lehetne beállítani a sofőröknek. Az alábbi eDriver szoftverben(2014) már be lehet állítani az „Off Duty”-t is pihenés céljából. Létrehoztak külön alvás menüpontot is.

13. ÁBRA - EDRIVER KÉPERNYŐKÉP



4.1.2.3 **Beléptető rendszerek**

Biztosan mindenki találkozott már különböző beléptető rendszerekkel. Például a munkahelyeken egy RFID chippel ellátott kártya RFID olvasóhoz történő érintesekor rögzítik, hogy ki és mikor lépett be az épületbe. Ezt egy adott rendszerben tárolják, így tudják monitorozni, hogy ki hány órát dolgozott. Bizonyos cégeknél figyelik azt is, hogy beléptek-e olyan területre ahova a munkavédelmi előírások betartása kötelező és betartották-e ezeket, illetve mennyi időt töltöttek ott.

Ezekre különböző log bejegyzések készülnek, amelyek beleírják egy fileba, hogy a kártya tulajdonosa mikor és hol volt.

Az említett rendszerekben közös, hogy eseményt rögzít, dátumot (adott időpontot) és a logok sorrendben vannak.

A hajósok azért hozták létre a hajónaplót, hogy mikor megérkeztek visszakövethető legyen, hogy mikor mi történt. Problémák (pl. a hajó sérülése, balesetek) esetén ez nagyon hasznos. A hajón a kapitány egyedül dönt, viszont a szárazföldre éréskor felelnie kell a döntéseiért. Programok esetén futás közben a rendszer viselkedését nehezen lehet nyomon követni, de a futás közben keletkezett logok segítségével könnyen fel tudják tárni, hogy mi volt a probléma oka, ezáltal ki is tudják azt javítani.

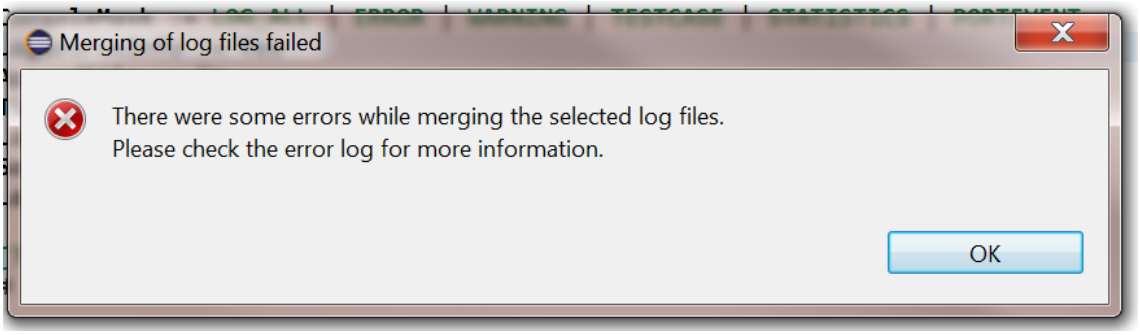
4.2 Miért loggoljunk?

Probléma esetén gyakran hangzik el a “Küldd el a logfile-t” kérés. Nem véletlen, hogy a logolást viccesen a fejlesztők “szemének és fülének”(developer’s eyes and ears) is nevezik. A logfilenak kereshetőnek és jól összeszedettnek kell lennie.

Főként akkor van rájuk szükség, amikor egy alkalmazásban nem várt módon hiba keletkezett és kíváncsiak vagyunk rá, hogy pontosan mi a hiba és mikor romlott el. A logolás dátumokra lebontva tárolja a múltbeli eseményeket is, segítségül szolgál arra, hogy meg lehet keresni, hogy hol romlott el, meddig működött jól az adott teszt. A gyakorlatban azonban nem mindig kaphatunk a loggolás segítségével egyértelmű, tiszta és teljes képet a hibák pontos helyéről és időpontjáról. A loggolásnak hatalmas költsége van, fogy vele a tárhely és a kapacitás. Ha nem történt hiba akkor feleslegesnek tűnik. Mint minden cégnél, itt is a kiadások csökkentésére és a bevételek növelésére törekszenek. A cél a kigyűjtött információ mennyiségének és a feldolgozás költségének optimalizálása.

Ajánlott, hogy egy logfileban sorrendben legyenek tárolva az események, azonban több gépes környezetnél rossz beállítások esetén eltérhetnek a formátumok és a gépek órabeállításai is. Ilyen esetekben nem lehet a logokat mergelni (összevonni egy fileba).

Az alábbi hibaüzenet például erre vonatkozik.



A kód tesztelésénél egy log file megléte nagy előnyt jelent, mert így nem kell átnézni az egész kódot, kiírja, hogy melyik függvényben volt a hiba és a konkrét hibaüzenetekből is sokat megtudhatunk, ha részletesen vannak logolva az adatok. Például a nullpointer exceptionről tudjuk, hogy nem foglaltunk az objektumnak helyet. Sok esetben előfordul, azonban, hogy a változók értékeit nem logolják ki, vagy olyan hibaüzenetet hoznak létre, amely nem utal arra, hogy mi volt a hiba oka. Esetleg a fejlesztők egyáltalán nem foglalkoznak a logolással. Nagyon fontos, hogyha mi magunk logolunk mindenképpen írjuk ki azoknak a változóknak az értékét, amelyeket a hibához lehet kötni a vizsgált rendszerben. A hiba okának felderítésében az internetes fórumok nem nyújtanak nagy segítséget, mert sokszor olyan technológiákat tesztelnek a telekommunikációs területen, amely még meg sem jelent, pl. 5G-s rendszerek logolásával kapcsolatban.

Összefoglalva: Hiba esetén a log üzenetek hasznosak és részletes logolás esetén áttekinthetőek. Ha a hibára fény derül, gyorsabban kijavíthatóvá válik, mert a múltbeli eseményeket is lehet látni. Viszont számolni kell a logolás költségével is. Ha például a hajó kapitánya csak a hajónapló kitöltésével foglalkozna, nem haladna előre a hajó. Ugyanez a helyzet a programozóknál, a fő feladatuk, hogy logikusan átgondolt, letesztelt kódot írjanak. Emellett sokan nem fordítanak elég időt a logolásra, mert a logokat átnézni időigényes, a logfileok tárolásához pedig rengeteg tárhely szükséges. Ha mi loggolunk, hiba esetén írjuk ki a hibához köthető változók értékét, függvények nevét, hogy log üzenetünk könnyebben értelmezhető legyen!

4.3 Miért hasznos a loggolás?

- Hibakeresés
- Tesztelés
- Hibamegelőzés esetén, és még időt is megtakaríthatunk vele.

A loggolás futásidőben történik meg. Kezelhetünk configurációs fileokat és használhatjuk üzenetküldésre is.

4.4 A log file felépítése

A teszt program futása alatt "logot" generál. A loggolás események sorozata, amelyeknek az időpontját is rögzítjük, ezáltal sorrendben vannak tárolva. Egy log különböző logolási szinteket tartalmaz, például: ERROR, DEBUG, INFO, ezzel felhívva a log olvasójának a figyelmét arra, hogy mikor kell közbelépni. Például ERROR esetén az adott file-t nem lehet megnyitni vagy egy kódrészlet nullával szeretne osztani.
A kilogolt események a teszt futtatása során létrejött események, amelyeket pontos időponttal, dátummal és hellyel ír ki. A hely alatt a kódban lévő elhelyezkedését, tartalmazó függvényét kell érteni. Ha valami többször szerepel a logfileban, mindig a legutolsó felvett értéket fogja kiírni. Nem kötelező mindent kiiratni, ezeket egyénileg be lehet állítani.

4.4.1 A loggolás működésének megértése

Az Eclipse Lunában a Project Explorer fülre kattintva a titan.core alatt a core mappában a Logger.cc C++-ra fordult kód és a Logger.hh header file alapján kell vizsgálni a logikai szerkezetet. Az első feladatom az volt, hogy jobban átlássam mi történik a loggolás során, hogy a TTCN3-ban írt egyszerű loggolási példa alapján megnézni a generált C kódban történt változásokat.

4.4.1.1 A TTCN-3 kódból történő C kód generálása:

1. Át kell váltani a TITAN EDITING nézetre
2. Titan Project-et kell létrehozni, jobb klikk/new/Titan Project
3. Az src mappába kell létrehozni a TTCN-3 filet, amibe fogjuk írni a tesztet
4. Létre kell hozni egy config file-t is az src mappába, amelynek az [EXECUTE] szekciójába meg kell adni létrehozott TTCN3 file nevét, ami jelen esetben a LogFirst lesz.

A TTCN-3 nyelven írt példakódom a következő volt:

```
module LogFirst {
type component CT{
}
testcase logger() runs on CT{
  var integer x1;
  var charstring x2;
  x1:=5;
  log(x1);
  x2:="Probaszoveg";
  log(x2);
 log("barmi","masodik","abc");
}
testcase logger2() runs on CT{
  var template integer x3;
  x3:=5;
 log(x3);
 } control{
  execute (logger());
}
}
```

A TTCN-3 fájlban a következő példa eseteket kel vizsgálni:

- Egy integer változó értékül kapja az 5öt

  var integer x1;

  x1:=5;

  log(x1);

- A log változón keresztül egy CharString-et, karakterláncot kap

  var charstring x2;

  x2:="Probaszoveg";

  log(x2);

- A log változóban kapja a string szöveget

  x2:="SzakdolgozatLog";

  log(x2);

- Templateként kap egy számot

  var template integer x3;

  x3:=5;

  log(x3);

4.5 A loggolás módja - Értelmezés

A példa esetekre a következő C kódot generálta:

```
verdicttype testcase_logger(boolean has_timer, double timer_value)
{
TTCN_Runtime::check_begin_testcase(has_timer, timer_value);
try {
TTCN_Runtime::begin_testcase("LogFirst", "logger", "LogFirst", "CT",
"LogFirst", "CT", has_timer, timer_value);
/* ../src/LogFirst.ttcn, line 7 */
INTEGER x1;
/* ../src/LogFirst.ttcn, line 9 */
x1 = 5;
/* ../src/LogFirst.ttcn, line 10 */
try {
TTCN_Logger::begin_event(TTCN_USER);
x1.log();
TTCN_Logger::end_event();
} catch (...) {
TTCN_Logger::finish_event();
throw;
}
} catch (const TC_Error& tc_error) {
} catch (const TC_End& tc_end) {
TTCN_Logger::log_str(TTCN_FUNCTION, "Test case logger was
stopped.");
}
```

A vizsgált try catch blokk előtt felismeri a változó típusát, értékét. Ez a 2. és a 3. esetben CHARSTRING az INTEGER helyett. A 3. esetben az x1=5 helyett TTCN_Logger::log_str(TTCN_USER, "SzakdolgozatLog"); fog szerepelni.

A logolás konkrétan a begin_event és az end_event között történik. A begin_testcase után felismeri, hogy integer/charstring típusú a változó, aztán az x1 vagy x2 változó értékül kapja az 5öt vagy a szöveget és ez fog kiloggolódni. A try catch blokk segítségével azt

érjük el, hogy bármi ami a függvénytörzsön belül történik nem hat ki a többi tesztre. Azaz gond esetén csak 1 tesztet ront el.

4.6 A TtcnLogger osztály felépítése és implementálása

A logger osztály egy mondatban összefoglalva: események sorozata, amelyeknek az időpontját is rögzítjük. A loggolás 3 részből áll:

- a Logger egy statikus belépési pont, biztosítja, hogy minden egy helyen menjen keresztül, továbbítja az információkat a LoggerPluginManagernek
- LoggerPluginManager tartja nyilván, hogy hány backend logger van beregisztrálva hozzá és küldi tovább mindnek a kezelendő üzenetet
- LegacyLogger, ami ténylegesen kiírja majd valahova a logolt üzenetet

Ezek közül a továbbiakban a LegacyLoggerrel fogok foglalkozni, amelyet két részre lehet osztani: loggolhatunk fileba vagy consolera.

A következőkben szeretném bemutatni TtcnLogger felépítését, implementálási lépéseit és a consolera logolás lényegi elemeit. Bemutatom, hogy a grafikus felület segítségével hogyan eszközölhet logolási beállításokat a felhasználó és ezek mit jelentenek.

4.7 Bevezetés a TtcnLogger osztály logikai szerkezetébe

A logolás kezdeti áttekintése kapcsán megbeszéltük, hogy a logban mindenképp szerepelni fog egy szöveg, amely egy logolási üzenet lesz. Ha csak ennyi lenne benne, akkor elég lenne egy függvény, ami stringet fogad és ír ki. Viszont jelen esetben összetett struktúrákat kell kiloggolni, ezeket összetettségük miatt nem lehet egy Stringként kezelni. Valamint figyelni kell arra is, hogy amikor egy függvény visszatérési értékét loggoljuk, lehet függvényt is hívni loggolás közben.

Log(„előtte”,fv(),”utána”) módon, és ennek kapcsán ha a meghívott függvény is logolni szeretne valamit azt külön kell választani a külső log függvény visszatérési értékétől.

Az üzeneteket bufferelni kell, amíg összerakjuk őket kisebb memóriát foglal, kiírásnál pedig egy logolási esemény minden adata egyben be tud kerülni a fileba vagy ki tud íródni a consolera. A logolás tehát a begin_event és az end_event között zajlik le, a current_event változó fogja gyűjteni a loggolandó üzenet részeit, új logolási egységet a log_event_struct osztállyal hozunk létre. Az end_event kiírja a buffer tartalmát a log_line függvény segítségével. Minden begin_event és end_event párost külön egységként kell

kezelni. A log hátterében a begin_event és az end_event egy even verment kezel, a tetején lévő elemhez íródik hozzá az új üzenet „darab”. Amikor a végrehajtás az end_eventig ér, akkor kiíródik az adott szinthet tartozó üzenet és lekerül a veremről, hogy a megelőző szint tudjon tovább íródni.

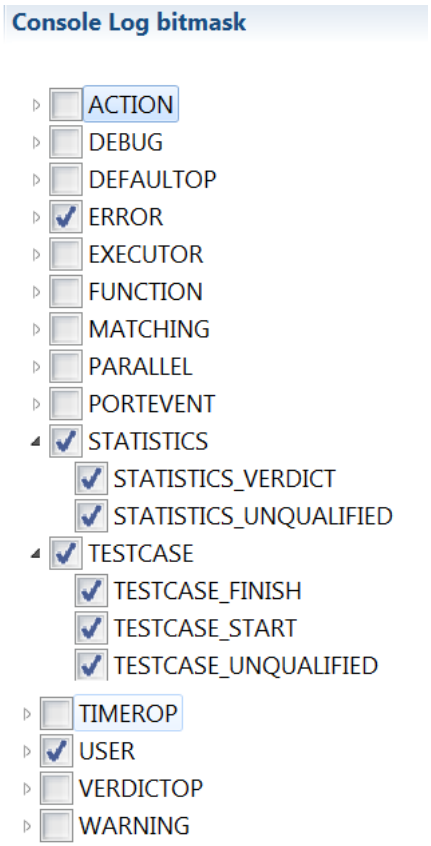
4.8 TtcnLogger implementálási lépései

A megvalósítás kapcsán fontos volt, hogy:

- a log eseményekhez Severityt tudjunk rendelni, azaz a felhasználó beállításai alapján el lehessen dönteni, hogy miket szeretne kiírni és miket nem
- le kell kezelni a például log_str-t, amely egy stringet fog kiírni és a log_chart, amely egyetlen karaktert ír majd ki.
- az eseményeknek kell, hogy legyen időbélyegük
- az eseményeknek tetszőlegesen komplex üzenetet és adatstruktúrát kell tudni kezelnie

A függvényeket JAVA nyelven az org.eclipse.titan.runtime\src\TtcnLogger.java osztályába és a különböző típusok osztályaiba implementáltam. A loggolás a Titanban a következőképpen épül fel: mindenképpen tartalmaz egy timestampet, egy filemaskot, consolemaskot.

4.8.1 *Log_this_event()*



A log_this_event függvény lényegi része, hogy a severityk alapján vizsgálja, hogy a fileos vagy a console-os loggolás történik meg. A severityk a loggolási szinteket jelölik. Ezt a Severity enum segítségével hoztuk létre. A logolási szintekből a felhasználó választhat az alapján, hogy mit szeretne a consolelogon vagy a filelogban látni. A console log és a file log bitmaszkja fogja ezt tartalmazni. Ez látható a képen. Ha fileba logolunk a log_this_event() meghívja a should_log_to_file függvényt, ha consolera akkor a should_log_to_console függvényt (nekünk itt azt kell vizsgálni, hogy a severity benne van-e a console log bitmaszkjában.)

Ezen kívül a log_this_event függvényben megjelenik az emergency logging, amely egy egyensúlyozó mechanizmus a fontos és a kevésbé fontos logok között. Gyűjti az utolsó X db üzenetet ha rendben van, akkor nem írja ki, viszont ha például Dynamic Testcase Errort dob valamelyik üzenet, akkor nem csak az utolsót, hanem az utolsó x-et írja ki. Hogy pontosan mennyi lesz az x, azt be lehet állítani. Egyfajta ringbuffert használ, amelynek a lényege, hogy az első üzeneteket, ahol nem volt hiba, felülírja az újakkal.

Működését a biztonsági kamerákéhoz lehet hasonlítani. A legtöbb ilyen eszköz az utolsó x nap videót tartja meg és utána felülírja a 24-25 órával megelőző anyagot. Ezáltal nem foglal sok helyet. Nem kell valakinek ott ülnie és folyamatosan néznie a kamerát, mert probléma esetén elővehető az előző nap felvétele.

4.8.2 Consolera logolás

A console-ra (terminálra) logolás a should_log_to_console() függvény segítségével zajlik. A C++ kód az alábbi volt:

```
boolean TTCN_Logger::should_log_to_console(TTCN_Logger::Severity sev)
{
  if (sev == TTCN_Logger::EXECUTOR_EXTCOMMAND) return TRUE;
  if (sev > 0 && sev < TTCN_Logger::NUMBER_OF_LOGSEVERITIES) {
    return console_log_mask.mask.bits[sev];
  }
  return FALSE;
}
```

Ennek jelentése: ha a severity típusú sev változó része az enumban megadott Severityknek, akkor írjuk ki a consolera. Azaz azt kell vizsgálni, hogy a severity benne van-e a console logmaszkjában.

Erre első ötletként létrehoztam egy Severity típusú Hashmap-et, amelyet feltöltöttem a severitykkel és a sorszámukkal, de ezáltal a should_log_to_console függvénynek 2 paramétert kellett volna fogadnia (először a severityt, aztán a severity nevét és sorszámát), ezért más megoldást kellett kitalálni. A containskey segítségével néztem meg, hogy a severity enum tartalmazza-e a severityt.

```
if(map.containsKey(sev)){
        return console_log_mask.mask.bits[map.get(sev)];
}
```

Ennél van sokkal egyszerűbb és logikusabb megoldás is. A második ötlet a bits elnevezésű Hashset létrehozása volt. A contains segítségével néztük meg, hogy szerepel-e az adott Severity az enumban.

Részlet a kódból:

```
final public HashSet<Severity> bits = new HashSet<Severity>();
public static boolean should_log_to_console(final Severity sev) {
        if (sev == Severity.EXECUTOR_EXTCOMMAND) {
                return true;
        }
        return console_log_mask.mask.bits.contains(sev);
}
}
```

A végső megoldás az lett, hogy létre kell hozni egy boolean típusú bits tömböt, amelynek a mérete a Severity enum hossza.

```
final public boolean bits[] = new boolean[Severity.values().length];
default_console_mask.bits[Severity.ACTION_UNQUALIFIED.ordinal()] = true;
```

A default console maskban a Severity értékeket kitölti true-val, ha a felhasználó csak olyan severitykből tud választani, amely létezik(benne van a Severity enumban). Ez a logger konstruktorában állítódik be, de ez később konfigurálható.

A .ordinal() veszi ki a különböző Severity típusoknak az enumban hozzárendelt szám értékeit (az elsőhöz 0. rendeltünk, a 2.-hoz 1et és így tovább).

4.8.3 A fileba és a consolera logolás összehasonlítása

Fileba vagy consolera logolnak a gyakorlatban?

A gyakorlatban az információ tárolásától, feldolgozásától és a szükséges információ mennyiségétől függ hogy fileba vagy consolera érdemes logolni. Ezt szemléltetem az alábbi táblázatban:

	fileba logolás	consolera logolás
információ tárolása	szeretnénk eltárolni a logokat	nem szeretnénk tárolni pl.: adatok feldolgozása hány százaléknál áll
információ feldolgozása	a jövőben pl. éjszakai teszteket másnap reggel kezdik el megnézni	csak az adott pillanatban van szükség az információra
szükséges információ mennyisége	sok	kevesebb

Nem szükséges a logolandó üzeneteket az adott gépen tárolni, átküldhetik azokat a hálózaton egy másik szerverre ahol nagyobb a tárhely.
Hálózat tesztelésénél elszigetelve a céges hálózattól tesztelnek, hogy hiba esetén ne terhelje azt túl. Menet közben úgy monitorozzák a logokat, hogy a logok egy olyan központi log gyűjtő rendszerre kerülnek, amit már el lehet érni kívülről.

Előfordul olyan is, hogy a console-ra kiiratott teszt futási eredményeit egy fileba tárolja és az 30 napig visszakövethető, de utána törlődik.

A fileba logolás tehát sok információ eltárolásában nyújt nem feltétlenül azonnali segítséget.

A console logolás pedig olyan, mint amikor a fedélzeten lévő kijelzőn az utasok tudják nézni, hogy éppen hol jár a repülő és mennyi idő van még hátra az utazásból. Nem fontos számukra, hogy 1 perccel előtte hol volt a gép.

4.9 Log() függvények a különböző osztályokban

A runtimeban előforduló típus és belőlük példányosodó érték vagy template struktúrák a kompozit tervezési mintát követik. Minden TTCN-3 típusból lesz egy osztály, amik így hierarchiát alkotnak. A hierarchia tetejét kérjük meg, hogy logolja magát, amihez minden csomópont rekurzívan az alatta lévő csomópontokat is megkéri, hogy logolják magukat. Ezáltal a nagy struktúra nem okoz gondot. A bizonyos osztályokon belül mindegyik tudja, hogyan kell kiírnia magát. Például egy integer tudja hogyan kell kiírnia magát, egy bitstring ettől teljesen eltérően fog kilogolódni. A „record of" szerkezet például elég ha csak annyit tud, hogy {} jelek között vesszővel elválasztva kell felsorolni az elemeit. Az elemeiről azonban semmit nem kell tudnia, mert azok saját magukat fogják logolni.
Néhány konkrét példával szeretném szemléltetni a log() függvények implementálását. Az alaposztályokban minden esetben vizsgáljuk az if függvény segítségével, hogy a változó amit a consolera szeretnénk kilogolni üres-e. Ha nem üres akkor kilogoljuk, ha üres, akkor pedig speciális állapotba kerül, azaz unbound. Ez azt jelenti, hogy létezik az adott változó, de nem kapott értéket.
A konkrét log() függvények az osztályokban az alábbiak, a következőkben ezekből fogok néhányat kiemelni és részletesen leírni:

- TitanInteger log()
- TitanInteger_template log()
- TitanBoolean log()
- TitanBoolean_template log()
- TitanFloat log()
- TitanFloat log_float()
- TitanFloat_template log()
- TitanBitString log()
- TitanHexString log()
- TitanOctetString log() és log_octet()
- TitanCharString log()
- TitanCharString_template log()
- TitanCharString_element log()
- TitanUniversalCharString log()

4.9.1 Egész számok logolása

Az egész számok logolását a TitanInteger osztály log() függvének rövid magyarázatával szeretném bemutatni.

```
public void log() {
        if (boundFlag) {
                if (nativeFlag) {
                        TtcnLogger.log_event("%d", nativeInt);
                } else {
                        TtcnLogger.log_event("%s", openSSL.toString());
                }
        } else {
                TtcnLogger.log_event_unbound();
        }
}
```

Integerek esetén NativeIntbe tároljuk a számot, ha a minimum (Integer.MIN_VALUE) és a maximum (Integer.MAX_VALUE) érték közé esik. Ha az intervallumon kívül esik, akkor BigInteger. A %s és a %d a formázásra utalnak, a %d a decimal integereket, az %s a stringeket jelöli.

A template osztályokban (például TitanInteger_template vagy TitanBoolean_template) 4 esetet vizsgálunk switch case segítségével:

- SPECIFIC_VALUE
- COMPLEMENTED_LIST
- VALUE_LIST
- VALUE_RANGE

Egész számok logolása esetén a kód a következő volt:

```
public void log() {
        switch (templateSelection) {
        case SPECIFIC_VALUE: {
                String tmp_str;
                if (single_value.isNative()) {
                        tmp_str = Integer.toString(single_value.getInt());
                } else {
                        tmp_str = single_value.getBigInteger().toString();
                }
                TtcnLogger.log_event("%s", tmp_str);
                break;
                }
        case COMPLEMENTED_LIST:
                TtcnLogger.log_event_str("complement");
        case VALUE_LIST:
                TtcnLogger.log_char('(');
                for (int i = 0; i < value_list.size(); i++) {
```

```
                        if (i > 0) {
                            TtcnLogger.log_event_str(", ");
                        }
                        value_list.get(i).log();
                    }
                    TtcnLogger.log_char(')');
                    break;
                case VALUE_RANGE:
                    TtcnLogger.log_char('(');
                    if (min_is_exclusive) {
                        TtcnLogger.log_char('!');
                    }
                    if (min_is_present) {
                        if (min_value.isNative()) {
                            TtcnLogger.log_event("%s",
Integer.toString(min_value.getInt()));
                        } else {
                            TtcnLogger.log_event("%s",
min_value.getBigInteger().toString());
                        }
                    } else {
                        TtcnLogger.log_event_str("-infinity");
                    }
                    TtcnLogger.log_event_str(" .. ");

                    if (max_is_exclusive) {
                        TtcnLogger.log_char('!');
                    }
                    if (max_is_present) {
                        if (max_value.isNative()) {
                            TtcnLogger.log_event("%s",
Integer.toString(max_value.getInt()));
                        } else {
                            TtcnLogger.log_event("%s",
max_value.getBigInteger().toString());
                        }
                    } else {
                        TtcnLogger.log_event_str("infinity");
                    }

                    TtcnLogger.log_char(')');
                    break;
                default:
                    log_generic();
                    break;
                }
                log_ifpresent();
        }
```

A kód értelmezése:
A Specific_value szó szerinti értékeket vizsgál. Például integernél egész számok, booleannél true, false és így tovább.

A complemented listnél egy egyszerű példával élve, ha a halmazunk 1,2,3 számokból áll és meghívjuk a complement(1,2)-t akkor a 3at fogja adni eredményül, vagy jelen esetben complement(1,2) fog kiloggolódni.

Value list alatt értékek vagy komplex adatstruktúrák listáját értjük, a példában számok listáját. Az értékek vesszővel vannak egymástól elválasztva.

Value range alatt pedig egy intervallumot, például számokat 1től 10ig. Vizsgálja, hogy a nativeInt vagy a BigInteger tartományba esik-e. Value Range-et TTCN-3ban a .. segítségével adhatunk meg, pl. (1..10)

A TitanInteger teszteléséhez a Titan Executing nézetébe átváltva, a project exploreren jobb egérgombbal kell kattintani. Ezután a New/Titan Java Projectet kell kiválasztani. Az src mappába kell létrehozni a TTCN-3 file-t, amelyet később Buildelünk majd Java Applicationként futtatunk.

4.10 Egész számok logolásának tesztelése

A TitanIntegerre írt alap tesztesetek a következők voltak:

```
module LoggingTitanInt {

control{

var integer szam1:=1256;
var template integer szam2:=(1,2,3,4,7,18);
var template integer cszam3:=complement(1,2);
var template integer valuerange:=(1..368);
var template integer nullateszt:=0;

log("Specific value: ", szam1);
log("Value List: ", szam2);
log("Complemented list: ", cszam3);
log("Value Range: ", valuerange);
log("Zero test: ", nullateszt);
log("Infinity test: ", infinity);
log("-Infinity test: ", -infinity);
}

}
```

Ezek outputja:

```
18:09:03.119000 Execution of control part in module LoggingTitanInt started.
18:09:03.119000 Specific value: 1256
18:09:03.119000 Value List: (1, 2, 3, 4, 7, 18)
18:09:03.119000 Complemented list: complement(1, 2)
18:09:03.119000 Value Range: (1 .. 368)
18:09:03.119000 Zero test: 0
18:09:03.135000 Infinity test: infinity
18:09:03.135000 -Infinity test: -infinity
18:09:03.135000 Execution of control part in module LoggingTitanInt finished.
18:09:03.135000 Verdict Statistics: 0 none (? %), 0 pass (? %), 0 inconc (? %), 0 fail (? %), 0 error (? %)
18:09:03.135000 Test execution summary: 0 test case was executed. Overall verdict: none
Total execution took 0.437204662 seconds to complete
```

A TitanBoolean osztály és a TitanFloat osztály logolása is hasonlóképp történik meg, a TitanBoolean osztályban .toString()-et, a TitanFloatban pedig a getValue()-t használjuk az értékek kinyerésére.

A TitanBoolean és a TitanFloat template ugyanazt a gondolatmenetet követi, mint az TitanInteger_template. Például a TitanBoolean osztály egy értékét a specific value esetben lehet megkapni, a többi része majdnem teljesen ugyanaz, mint a többi template osztály logolásánál:

```
public void log() {
	switch (templateSelection) {
	case SPECIFIC_VALUE:
		TtcnLogger.log_event_str(single_value.getValue() ? "true":"false");
		break;
	case COMPLEMENTED_LIST:
		TtcnLogger.log_event_str("complement");
	case VALUE_LIST:
		TtcnLogger.log_char('(');
		for (int i = 0; i < value_list.size(); i++) {
			if (i > 0) {
				TtcnLogger.log_event_str(", ");
			}
			value_list.get(i).log();
		}
		TtcnLogger.log_char(')');
		break;
	default:
		log_generic();
		break;
	}
	log_ifpresent();
}
```

4.11 Karakterlánc logolása

A TitanCharString logolásánál vizsgálni kellett az isPrintable() függvény segítségével, hogy nyomtatható-e az adott karakter. A logCharEscaped() függvény segítségével pedig elértük, hogy a felhasználók számára könnyen felismerhetően írja ki például a sortörést(\n) és tabulátort(\t) is.

Ezeket a TtcnLogger osztályban hoztuk létre.

```
public static boolean isPrintable(final char c) {
        if (c >= 32 && c <= 126) {
                // it includes all the printable characters in the ascii code table
                return true;
        }

        switch (c) {
        case '\b':
        case '\t':
        case '\n':
        case '\f':
        case '\r':
                return true;
        default:
                return false;
        }
}
```

A TitanCharString log() függvénye, mivel stringről van szó, máshogy fog felépülni, mint az integeré. Először megvizsgálja, hogy isPrintable()-e, tehát nyomtatható karakterről van-e szó, majd beállítja az állapotát: NPCHAR, INIT, és PCHAR lehet. Ezekre az állapotokra azért van szükség, hogy az aktuális karakternél el lehessen dönteni ki kell-e elé rakni a konkatenáló jelet. Ha nyomtatható karakterek kerülnek egymás után illetve ha nem nyomtatható karakterek után szintén nem nyomtatható karakterek következnek, akkor nem kell kirakni az összefűző jelet (pl. & vagy +).

A nyomtatható karaktereknél az egyértelmű, általunk ismert karakterekre kell gondolni, ezek az ASCII táblában 32 és 126 közé esnek, a nem nyomtatható ASCII karaktereket pedig az alábbi táblázat szemlélteti:

Non-printable ASCII Characters

128	Ç	143	Å	158	×	173	¡	188	╝	203	╦	218	┌	233	Ú	248	°
129	ü	144	É	159	ƒ	174	«	189	¢	204	╠	219	▉	234	Û	249	¨
130	é	145	æ	160	á	175	»	190	¥	205	═	220	▄	235	Ù	250	·
131	â	146	Æ	161	í	176	▒	191	┐	206	╬	221	¦	236	ý	251	¹
132	ä	147	ô	162	ó	177	▒	192	└	207	¤	222	Ì	237	Ý	252	³
133	à	148	ö	163	ú	178	▓	193	┴	208	ð	223	▀	238	¯	253	²
134	å	149	ò	164	ñ	179	│	194	┬	209	Ð	224	Ó	239	´	254	■
135	ç	150	û	165	Ñ	180	┤	195	├	210	Ê	225	ß	240	≡		
136	ê	151	ù	166	ª	181	Á	196	─	211	Ë	226	Ô	241	±		
137	ë	152	ÿ	167	º	182	Â	197	┼	212	È	227	Ò	242	‗		
138	è	153	Ö	168	¿	183	À	198	ã	213	ı	228	õ	243	¾		
139	ï	154	Ü	169	®	184	©	199	Ã	214	Í	229	Õ	244	¶		
140	î	155	ø	170	¬	185	╣	200	╚	215	Î	230	µ	245	§		
141	ì	156	£	171	½	186	║	201	╔	216	Ï	231	þ	246	÷		
142	Ä	157	Ø	172	¼	187	╗	202	╩	217	┘	232	Þ	247	¸		

4.12 Karakterlánc logolásának tesztelése

A TitanCharString osztály logolásának tesztelésére 3 példát mutatok be, az elsőben az isPrintable() függvény sortöréssel kapcsolatos logolását, a másodikban különböző karakterek összefűzését, a 3.ban a TitanCharStringek logolását a template matcheléssel együtt mutatom be, amelyhez a log(match()) függvényre lesz szükség.

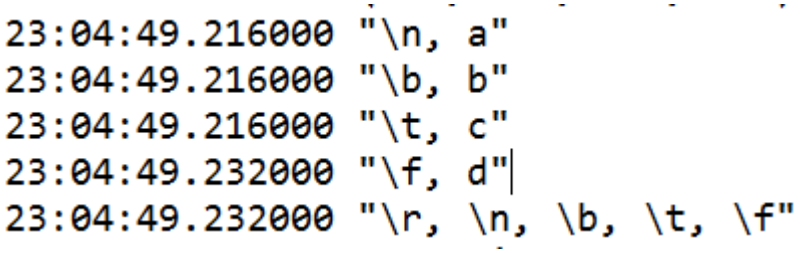
1. Az alábbi módon ellenőriztem, hogy a CharString isPrintable függvényébe beírt sortöréseket valóban kiírja-e

Input:

```
var charstring y1:="\n, a";
var charstring y2:="\b, b";
var charstring y3:="\t, c";
var charstring y4:="\f, d";
var charstring y5:="\r, \n, \b, \t, \f";

log(y1);
log(y2);
log(y3);
log(y4);
log(y5);
```

Output:



1. Egy másik példa különböző karakterek összefűzésére. Képes szöveget, egy karaktert(x), még egy szöveget és egy számot (2) összefűzni:

   var charstring x;

   x:= "pelda"&char(0,0,0,120)&"szakdolgozat"&char(0,0,0,50)

   Az output pedig „**probaxszakdolgozat2**” lesz.

Template Matchelés jelentése

Mivel matchelés csak template-ek és értékek között jöhet létre, a template osztályokban kellett lérehozni a log_match függvényeket. A logmatch azt vizsgálja, hogy 2 azonos típusú template és érték egyezik-e, tényleg azonos elemeket tartalmaz-e. Ehhez szükség van a matching verbosityra, amely egy 2 értékű enum, lehet compact vagy detailed. Ha a compact van kiválasztva, akkor az egyező struktúrákat, mezőket nem írja ki, csak azokat amelyek „unmatcheltek”. Ezt az alábbi példa is jól szemlélteti.

TitanCharString logolásának tesztelése (logmatch teszttel együtt)

```
//charstring logmatch
var CharType c1:={"a"};
var template CharType c2:={"a"};
var template CharType c3:={"b"};

var charstring char100:=("abc");
var template charstring char101:=("bca");
var template charstring char102:=("abc");

log("LOGMATCH FOR CHARSTRING");
log(match(char100,char101));
log(match(char100,char102));
log(match(c1,c2));
log(match(c1,c3));
```

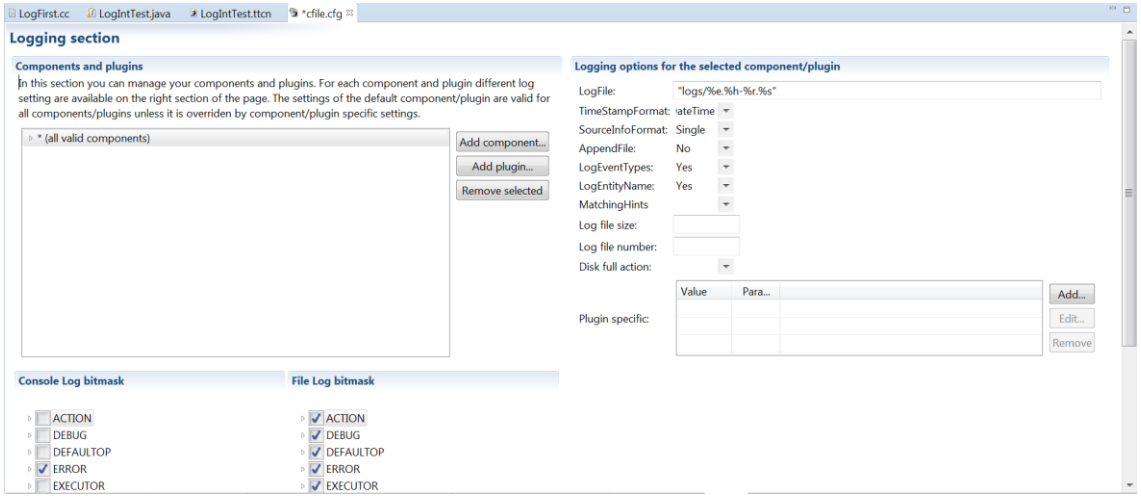
Output

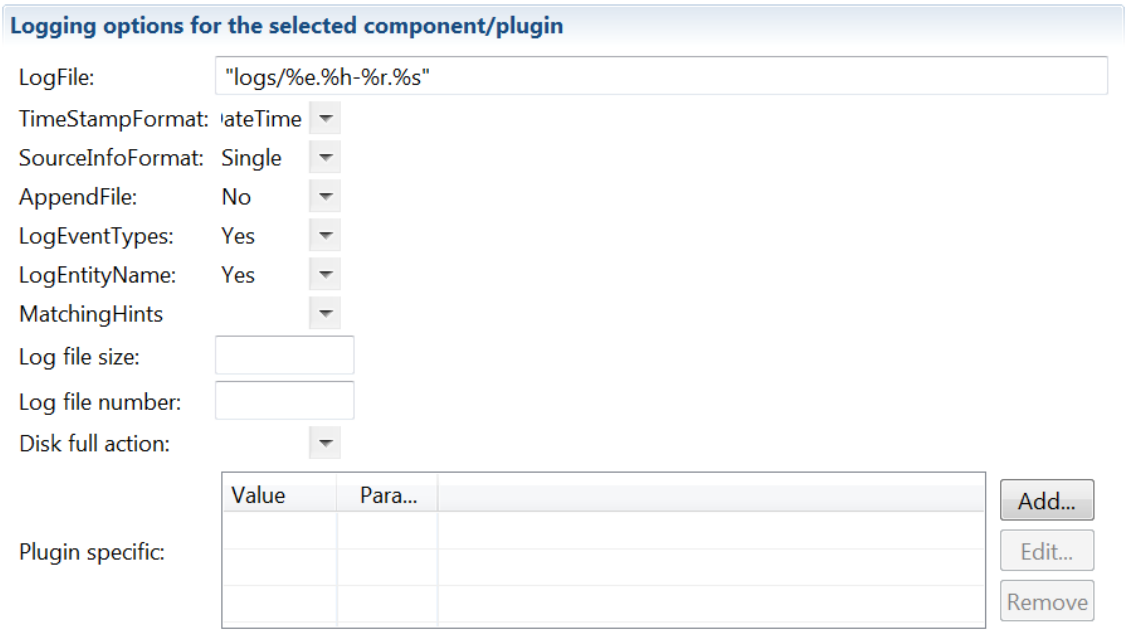
```
23:49:10.734000 Execution of control part in module LoggingTitanChar started.
23:49:10.750000 LOGMATCH FOR CHARSTRING
23:49:10.750000 "abc" with "bca" unmatched
23:49:10.750000 "abc" with "abc" matched
23:49:10.750000  matched
23:49:10.750000 .charrec := "a" with "b" unmatched
23:49:10.750000 Execution of control part in module LoggingTitanChar finished.
23:49:10.766000 Verdict Statistics: 0 none (? %), 0 pass (? %), 0 inconc (? %), 0 fail (? %), 0 error (? %)
23:49:10.766000 Test execution summary: 0 test case was executed. Overall verdict: none
Total execution took 0.402581632 seconds to complete
```

4.13 Logolási beállítások a felhasználó szemszögéből

Mit lát a felhasználó? A felhasználó az alábbi felületet látja, itt tudja beállítani miket szeretne kiloggolni.

Ezt a Titan Executing nézetben a létrehozott .cfg file-nál a Logging fülre kattintva értem el. A következőkben a különböző mezőket és beállításokat fogom részletezni.





A LogFile mező tartalmazza, %-kal jelölve, hogy melyik fileba logoljon. Jelen példában minden komponens a saját külön filejába ír. Ezt később majd össze lehet mergelni, hogy egyben lehessen látni a logokat.

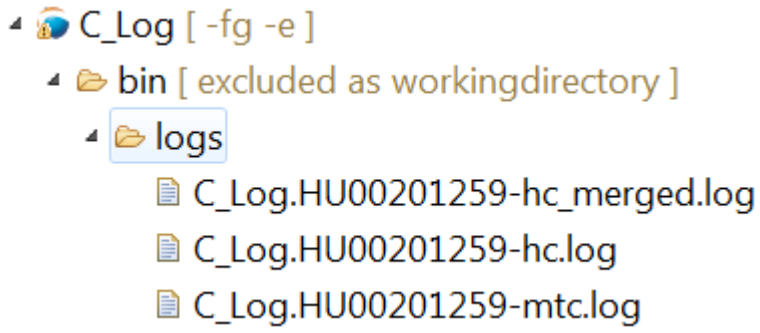
A %-os metakarakterek különböző fontos jelentéssel bírnak:

%l	login név
%h	számítógép hosztneve
%i	log szekvenciaszáma
%e	TTCN-3 executable, a végrehajtandó file neve
%n	A teszt komponens neve, ha a PTC adott neki nevet, HC ha a host controller hozta létre a logfile-t, MTC ha az MTC hozta létre a logfilet
%p	A Unix process Process ID-ja
%r	Komponens azonosító (PTC: egész szám>=3, MTC és HC esetén mtc,hc vagy single
%s	String log
%t	TTCN-3 component type
%%	% karakter

Nem történik fileba logolás:

- Ha ezt a mezőt üresen hagyjuk, akkor nem fileba fog logolni, hanem consolera, a FileMask beállításaitól függetlenül.
- Ha a FileMask LOG_NOTHING-ra van állítva

A cél, hogy minél több információt tartalmazzon a log fileneve és egyszerre ne akarjon több folyamat ugyanabba a logfileba írni. Mindenképpen egyedi nevet kell adni az elérési útvonalnak, erre szokták használni a %h, %r, %p kapcsolókat, így a számítógép hosztneve, a process ID és a komponens azonosító segít megkülönböztetni a logfileokat. A különböző komponensek különböző fileba logolnak.



Az beállításom tehát: "logs/%e.%h-%r.%s" volt, így a képen látható C_Log a TTCN-3 fileom neve, aztán következik a számítógépem hosztneve, majd a komponens azonosítója. A legvégén a %s írja oda, hogy log. Ebben az esetben minden komponens a saját külön filejába ír és létrejött egy merged log is, amelyben a kettő együttesen jelenik meg.

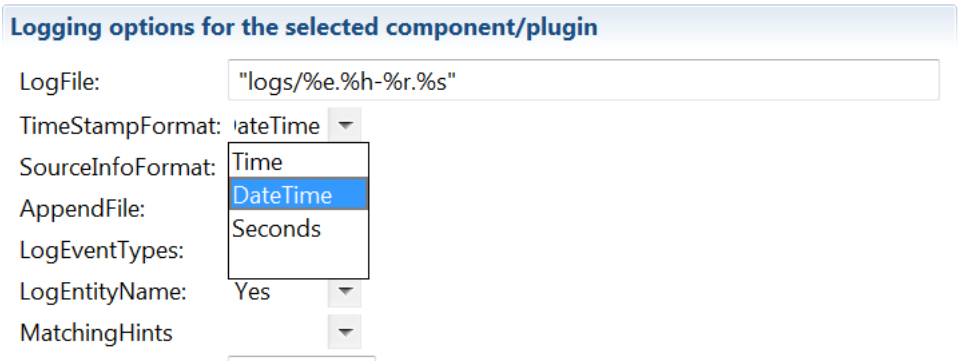
Megnyitottam a képen látható első file-t és ebből a példa log fileból illesztek be 2 sort, hogy lehessen látni hogyan néz ki egy egyszerűbb log:

```
2018/Mar/18 12:20:54.072000 hc EXECUTOR - MTC was created. Process id: 15368.
2018/Mar/18 12:20:54.091001 mtc EXECUTOR - TTCN-3 Main Test Component started on HU00201259. Version: CRL 113 200/6 R2A.
```

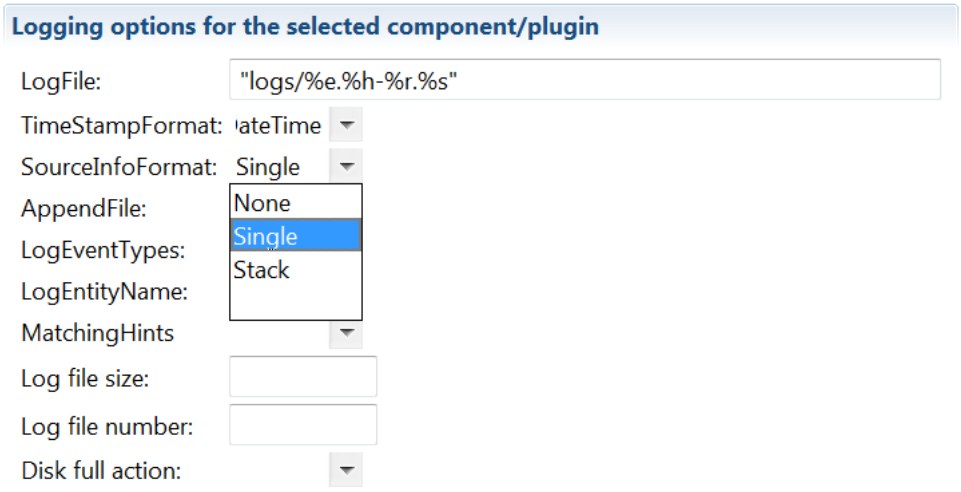
Tartalmaza a dátumot (DateTime formátumban), a következő „oszlopban” a komponens azonosítót, majd a logolási szintet (Severity) és végül a String üzenetet. A log fileok általában több ezer sorosak, jelen példában is 25 sor TTCN-3 kódból 40 sornyi log generálódott. A példával csak a szerkezeti felépítését szerettem volna szemléltetni.

A %i-t akkor célszerű használni, amikor meg van adva a logfileok maximális mérete és amikor az 1. betelik, újat szeretnénk nyitni, nem felülírni az előzőt. Így jön létre a pelda1.log, pelda2.log.

A következő opció a TimeStampFormat, amely alapvetően 3 választási lehetőséget kínál: Time, DateTime és Seconds. A DateTime kiírja a logolás teljes időpontját dátummal, órával, perccel, másodperccel. A Time óra:perc:másodperc formátumot, a Seconds pedig az eltelt másodperceket.



A **SourceInfoFormat**nál None-ra kattint a felhasználó ha nem kéri ezt a beállítást, Single-t akkor választ ha az érdekli, hogy hol volt a testcase/függvény hívás, Stacknél pedig kiírja, hogy futáskor melyik sor melyik függvényében hívták meg a logot. Ez akkor látszik, ha a hívott függvényen belül is logoltunk.

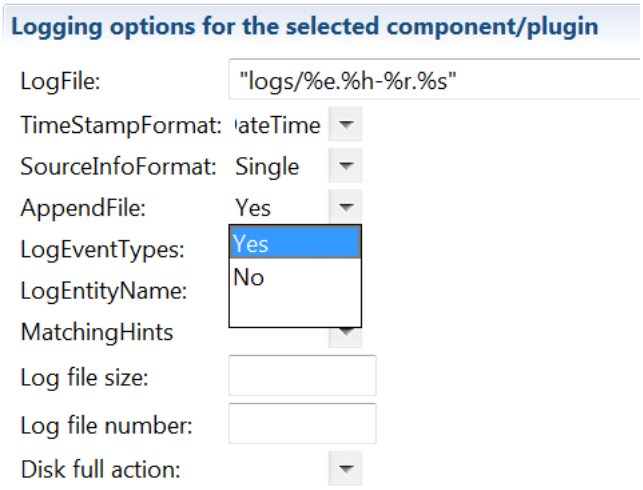


Példa log részlet és értelmezés Single beállításra:

SIPexample.ttcn:57(testcase:MakeCall)

TTCN3file neve:melyik sorban volt a log (függvény, teszt eset amely a logot tartalmazza)

AppendFile



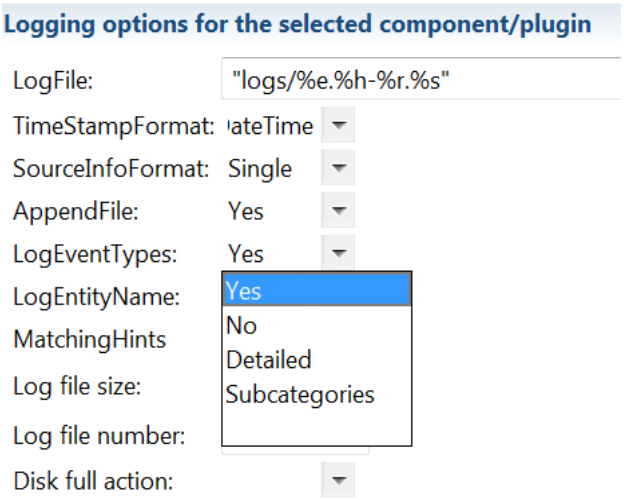
Ha a logfile nevében szeretnénk megjeleníteni a testcase vagy a komponens nevét, akkor célszerű az AppendFile-t Yes-re állítani, mert alapértelmezetten a fent említett sémába „part$_1$”, „part$_2$”, „part$_n$”-t fog beleírni a filenévbe. Például:

SzakdolgozatLog.HU00201260-hc-part1.log

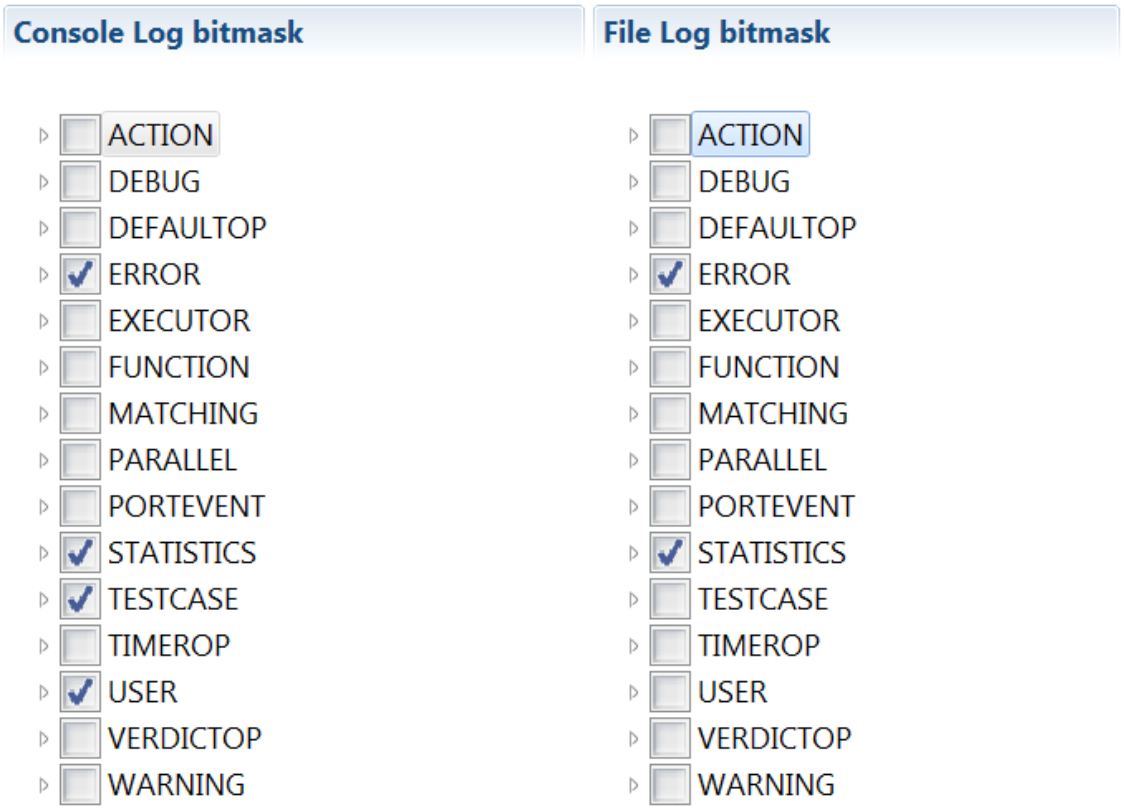
SzakdolgozatLog.HU00201260-hc-part2.log

A való életben nem csak 2 log file fog generálódni, hanem például 50 db és később már senki sem fogja tudni, hogy a part39 pontosan melyik teszt eseteket vizsgálta. Ezért erősen ajánlott, hogy az AppendFile:=Yes legyen beállítva.

LogEventTypes



A **LogEventTypes**ban állítható be, hogy a log üzenet tartalmazza-e a logolási szintek típusait, hogy pontosan melyikeket szeretné a felhasználó látni, azokat pedig a ConsoleLog és a FileLog Bitmaszkjában pipálhatja be:



A fenti példa szerint a fileba csak a Dynamic test case errorok és a test case-ek verdictjei kerülnek. DynamicTestCaseError például akkor keletkezik, ha a snapshot matchelés nem sikerül. Snapshotok úgy keletkeznek, hogy bizonyos időközönként „lefotózzák” a szerverek aktuális állapotát és ha nem egyezik az előző (helyes) állapottal, akkor fog hibát

dobni. A példában a consoleon az említetteken kívül a testcase indítását is megjelenítjük és a user (felhasználó) által kiírt logokat.

Ha fileba logolunk és a filemasknál semmit nem pipál be a felhasználó, akkor a LOG_ALL érvényesül. Ha a console_masknál nem jelöl semmit, akkor az alapértelmezett értéke a ERROR|WARNING|ACTION |TESTCASE|STATISTICS érvényesül.

A főbb kategóriák jelentése az alábbi táblázatban szerepel:

4. TÁBLÁZAT - A FŐBB KATEGÓRIÁK JELENTÉSE

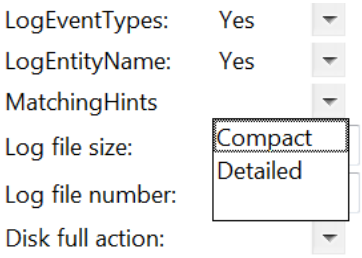
ACTION	Ttcn3 műveletek
DEBUG	Debuggoláshoz, nyomkövetéshez szükséges üzenetek a tesztportokról és a függvényekről
DEFAULTOP	Activate, deactivate, nem kategorizálható log függvények
ERROR	Dynamic testcase errorok
EXECUTOR	Belső Titan események, pl. MTC-k indítása
FUNCTION	Meghívott függvények
MATCHING	Mintaillesztési hibák vizsgálata a portokon létrejött műveletekből
PARALLEL	A párhuzamos teszt futtatáshoz szükséges műveletek, pl. létrehozás, portok összekötése (connect,map)
PORTEVENT	Az összes port művelet A send,receive-en kívül az alapvető állapotváltozások, ellenőrzések
STATISTICS	Verdictek végső statisztikái (fail,pass,error,inconclusive)
TESTCASE	Jelöli, amikor az adott tesztcaseket elkedi futtatni és amikor befejezi őket és kiírja minden egyes tesztcase verdictjét.
TIMEROP	Timer (időzítő) elindításának, megállításának időpontja
USER	A felhasználó log üzenetei

VERDICTOP	Ítélet meghatározása: pass,fail,error,inconclusive
WARNING	Figyelmeztetések futás időben történő eseményekről, pl. egy nem aktív időzítő leállt

A loggolási szinteket különböző alkategóriákra is lehet bontani. Például a MATCHING mintaillesztési kategóriát az alábbi alkategóriákra bontották:

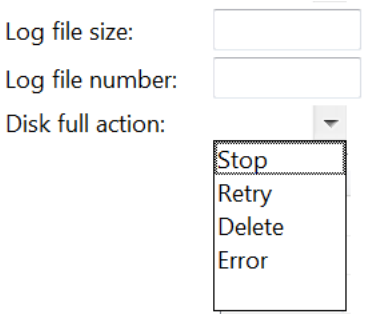
MATCHING_DONE	Sikeres matchelés
MATCHING_TIMEOUT	A timer nem timeout-tal lépett ki és nincs az expired (lejárt) timerek között sem
MATCHING_PMSUCCESS	Procedúra alapú mappelt portok között sikeres match
MATCHING_PMUNSUCC	Procedúra alapú mappelt portok között sikertelen match
MATCHING_PCSUCCESS	Procedúra alapú connected portok között sikeres match
MATCHING_PCUNSUCC	Procedúra alapú connected portok között sikertelen match
MATCHING_MMSUCCESS	Üzenet alapú mappelt portok sikeres mintaillesztése
MATCHING_MMUNSUCC	Üzenet alapú mappelt portok sikertelen mintaillesztése
MATCHING_MCSUCCESS	Üzenet alapú connectelt portok sikeres mintaillesztése
MATCHING_MCUNSUCC	Üzenet alapú connectelt portok sikertelen mintaillesztése
MATCHING_PROBLEM	Sikertelen mintaillesztés
MATCHING_UNQUALIFIED	Bármi más matcheléssel kapcsolatos log üzenet amely nem tartozik a fenti kategóriák egyikébe sem

A **LogEntityName** az alkategóriák megjelenítésére vonatkozik, például a fent említett MATCHINGnél kiírná, hogy az egy MATCHING_PROBLEM.



A **MatchingHints** a mintaillesztési hibák leírására vonatkozik, TTCN-3ban erre a log(match(...)) függvényt alkalmazzuk.

Compact beállításnál csak a nem matchelt mezőket és struktúrákat írja ki, míg **Detailed** esetén kiírja a mintaillesztésnek megfelelő és a nem megfelelő párokat is. Például ha két mező értéke 2 és 2, akkor a „2 with 2 matched" lesz kiírva ha két mező értéke 4 és 3 és ezeket szeretnénk „matcheltetni", akkor „ 4 with 3 not matched" lesz kiírva a consolera.



A képen látható többi esetben a LogFileSize-nál beállítható a logfile mérete. A **Disk full action**-nél állítható be, hogy amikor a Titan nem tud a log fileba írni, a képen látható 4 esetből melyik történjen meg.

Stop: abbahagyja a logolást, de a testcase-ek végrehajtása folytatódjon

Retry: időnként próbálkozzon (alapértelmezetten 30 mp) újraindítani a logolást

Delete: a legrégebbit kitörli és létrehozza sorszám szerint a következő logfilet (például ha 4 db volt, akkor a logfile5-öt) és ebbe fog logolni

Error: hibaüzenetet írjon ki (DynamicTestCaseError).

LogFileNumber: Ezzel a paraméterrel állítható, hogy maximum hány logfile verziót szeretnénk tárolni, alapértelmezetten 1-re van beállítva.

A fent részletezett beállításokat ugyanúgy a Titan Executing nézetben az adott C projekt cfg filejának [LOGGING] szekciójában is be lehet állítani.

```
[LOGGING]
# In this section you can specify the name of the log file and the classes of events
# you want to log into the file or display on console (standard error).

LogFile := "logs/%e.%h-%r.%s"
FileMask := LOG_ALL | DEBUG | MATCHING
ConsoleMask := LOG_ALL | ERROR | WARNING | TESTCASE | STATISTICS | PORTEVENT
LogSourceInfo := Yes
AppendFile := No
TimeStampFormat := DateTime #DateTime #Time #Seconds
LogEventTypes := Yes
SourceInfoFormat := Single
LogEntityName := Yes
```

```
2018/Mar/23 21:43:05.056814 mtc ERROR - Dynamic test case error: Module enumteszt does not have control part. (No such file
2018/Mar/23 21:43:10.127104 mtc STATISTICS - Verdict statistics: 0 none, 0 pass, 0 inconc, 0 fail, 0 error.
2018/Mar/23 21:43:10.127104 mtc STATISTICS - Number of errors outside test cases: 1
2018/Mar/23 21:43:10.127104 mtc STATISTICS - Test execution summary: 0 test case was executed. Overall verdict: error
```

Itt egy újabb részlet egy log fileból, amely tartalmazza a dátumot(DateTime formátumban), a componens típust, a filemaskot és a log üzenetet. Ez egy példa Dynamic Test Case Error-ra, mert az enum teszt nevű fileban nem hoztam létre control{} részt, így nem tudta lefuttatni.

5 Tesztelés

5.1 Az agilis szoftverfejlesztés alapelvei és a tesztelés fontossága

A következőkben Dawn Haynes – Being More Agile without Being Agile című HUSTEF (2017) – en tartott előadásának üzenetét foglalom össze, amelyből megismerhetjük az agilis gondolkodásmódot, amelyet a szoftverfejlesztő csapatok alkalmaznak. Hivatkozok továbbá a Tiszta kódban olvasottakra és Bíró Szilárd Agilis szoftverfejlesztés című előadására is. A fejezetből kiderül, hogy gyakorlatilag ugyanolyan fontos tesztelni, mint fejleszteni. A Magyar Tudomány cikkje alapján fény derül a BlackBox és a WhiteBox tesztelés közötti különbségre. Ezen kívül szakdolgozatom ezen fejezete a TDD alapelvekbe is betekintést nyújt James W. Grenning könyve alapján.

A felhasználók magas szintű, hiba nélküli alkalmazásokat követelnek. Egy programot nem elég átgondolni és megírni, le is kell tesztelni. Legjobb, ha a fejlesztő és a tesztelő is külön-külön leteszteli. A tesztelésre fordított figyelem az idő múlásával egyre nő, a cégeknél növelik az erre fordított költségkeretet.
A legtöbb helyen a fejlesztőcsapatok az agilis szoftverfejlesztés irányelvei szerint fejlesztenek, amelynek a lényege a rövid idő alatt kiszállított jól működő szoftver. A szoftvert nem egyben szállítják, hanem kisebb részegységenként, működő verziókat szállítanak gyakori időközönként. A követelmények gyakran változnak, van, hogy egyik pillanatról a másikra. Ezeket el kell fogadni és alkalmazkodni kell a kialakult helyzethez. A fejlesztők hozzállása is fontos: egyenletes tempóban kell haladni, a fejlesztőknek együtt kell működniük csapattársaikkal, fontos az önszerveződő csapatmunka. (BÍRÓ, Sz[34])
Sok csapat úgy szeretne belevágni az agilis szoftverfejlesztésbe, hogy kijelenti „ a mai naptól agilisek vagyunk”.(HAYNES, D., HUSTEF, 2017) Az agilis szoftverfejlesztés nem csak a szabályok betartását jelenti, hanem egyfajta gondolkodásmódot is. A legfontosabb, hogy a célt mindig szem előtt kell tartani, mert „ha nem tudod merre haladsz, nem is fogsz ott kikötni.” (HAYNES, D.) Meg kell tanulni együtt dolgozni a többiekkel, és az emberektől függ, hogy hogyan lesz a csapat az agilis szoftverfejlesztés követője. Fontos szerepet játszik a minőség, mert a visszacsatolásnál nem csak azt kell tudnunk, hogy mi a rossz, hanem ki kell emelni azt is, hogy mi a jó.

[34] Bíró, Szilárd, Agilis szoftverfejlesztés c. előadása alapján, letöltve: 2018. május

A tesztelőknek egy más szemszögből ismerik a rendszert, információik vannak a felhasználókról és a fejlesztett szoftver felhasználásról is. A tesztelésnek fontos szerepe van a szoftver helyes működésének meghatározásában és a hibák feltárásában. A kommentek nagy jelentősségel bírnak a fejlesztés és a tesztelés szempontjából is. Egy komment esetében elengedhetetlen, hogy olyan dolgot tartalmazzanak, amit a kód nem tud, egyértelműen, lényegretörően kell fogalmazni, „de nem: i++ = növelem i-t szinten" kell kommentet írni, a kikommentelt kódrészleteket vagy túl régi magyarázatokat ki kell törölni (MARTIN,R.C.).

A kollégákkal való együttműködés is kulcsszerepet játszik, be kell vallani a hibákat, együtt kell működni és úgymond „megosztani" a minőséget.(HAYNES, D)

Ösztönözni a másikat a munkára és együtt megoldani a feladatokat. Az úgynevezett retrospective meetingeken lehet elmondani hogy min szeretnénk változtatni a jobb munkavégzés érdekében.

Az agile az agyban és a szívben kezdődik. Egy agile környezetben dolgozó fejlesztő vagy tesztelő jó, ha olyan tulajdonságokkal rendelkezik, amelyeket nem lehet tanítani. Elengedhetetlen a kíváncsiság, tudásvágy, alkalmazkodónak, kreatívnak és bátornak kell lenni. A rossz híreket is tudni kell közölni, változásokat kezelni kell és fontos a keresési készség, azaz mindenképpen válaszokat kell találni a kérdésekre.

Robert C. Martin könyve és Dawn Haynes előadása alapján fontos, hogy tesztelésnél minden lehetséges esetet le kell tesztelni. A szoftverfejlesztésnél pedig a hozzáállás, a tudásvágy és a másokkal való együttműködés éppen annyira fontos, mint a kód megírása.[35]

[35] Dawn Haynes – Being More Agile without doing Agile előadása alapján

5.2 Teszttervezési technikák

A kommunikációs protokolloknak és szoftvereknek különböző tesztelési folyamatokon kell átesniük, amelyeknek nagy a költsége.

„Fehér doboz tesztelés esetén látni lehet a kódot és ismert a felépítése, struktúrája.”

„Fekete doboz tesztelés esetén csak a tesztelt szoftver külső viselkedése ismert”.

„A távközlésben használt szoftverek fejlesztése modulonként történik” (DIBUZ,S., CSOPAKI, GY[36])

- Modul vagy komponens teszt az elkészült modulok tesztje, általában a kódoló végzi, fehér doboz.
- Teszterek ellenőrzik, hogy a specifikációnak megfelelően működik-e, ez a fekete doboz tesztelés, először helyes, majd helytelen adatokkal vizsgálódnak
- Rendszer tesztelése valóságos környezetben: terhelési viszonyok vizsgálata, tud-e más rendszerekkel együtt működni
- Konformancia teszt: specifikációnak megfelel-e
- Együttműködési tesztek során több szállítóval történő együttműködést vizsgálják
- Regresszió teszt: új funkcionalitás hozzáadása után ellenőrizzük hogy nem rontottunk-e el valamit a ‚régi”fejlesztés előtt jól működő részben

A tesztelés automatizálásával sok időt takaríthatunk meg, ehhez szükségünk van egy tesztsorozatok leírására szolgáló programozási nyelvre.

Például ilyen a TTCN-3 nyelv. A gyakorlatban a tesztelők inkább egy úgynevezett graybox testinget használnak, amely a fekete doboz és a fehér doboz teszteléséből áll össze.[37]

[36] Magyar Tudomány Infokommunikációs Protokollok című cikkje alapján (Dibuz Sarolta és Csopaki Gyula)

[37] Magyar Tudomány Infokommunikációs Protokollok című cikkje alapján (Dibuz Sarolta és Csopaki Gyula)

5.3 Tesztvezérelt fejlesztés (TDD) áttekintése

A TDD[38]-ben először a teszteket írjuk meg, ezzel elkerülve a hetekig, hónapokig tartó debuggolást és így könnyebben, gyorsabban kideríthető a probléma oka. A TDD lényege a megelőzés, segítségével azonnal fény derül a hibás kódrészletre.

A TDD menete:

1. Tesztek elkészítése
2. Tesztek futtatása
3. Kis változtatásokat eszközölni, hogy fusson a teszt
4. Tesztek újrafuttatása
5. Hatékonyság növelése, kód javítása: a kódnak nem csak jól működőnek, hanem jól struktúráltnak, könnyen olvashatónak is kell lennie

A TDD előnyei:

- Kevesebb hiba
- Gyorsabb javítás
- Ok-okozati javítás: érintett osztályok szintén hibát jeleznek ha egy teszt failel
- dokumentációt színesíthetik a tesztelt példák
- előrehaladás monitorozása

Összefoglalva a TDD-vel időt lehet spórolni és bizonyos esetekben kezdő fejlesztőknél egy hatékonyabb, gyorsabb programozási módszert valósít meg. Az úgynevezett DLP[39] egy másfajta gondolkodásmódot igényel, és sok időt vesz el, hogy megkeressük pontosan mi okozhatta a hibát. (GRENNING, J.W.)

5.4 TDD elv megjelenése a Titánban

A TDD elnevezést Kent Beckhez szokták visszavezetni, 2003ra.
A Test Driven Development betűhármast sokféleképpen lehet értelmezni. Például lehet egy konkrét logika, amely azt írja elő, hogy 1-1 függvényt, osztályt minden más elemtől teljesen függetlenül teszteljünk le. Ilyen vonatkozásban egyáltalán nincs jelen a Titanban. Az összes teszt onnan indul, hogy TTCN-3, ASN.1 fileokat adunk a fordítónak és megnézzük, hogy milyen hibákat talál vagy hogyan fut le a kód.

Más szempontból ha a módszer elvét, szellemiségét nézzük, akkor teljes mértékben megvan az elemek aprólékos tesztelése.

---

[38]Test Driven Development
[39]Debug Later Programming

C++ oldalon:

- a regression könyvtár típusonként teszteli végig a nyelv elemeit
    - minden műveletre külön vannak tesztek, pl. összeadásf
- a Function teszt könyvtár alatt különböző kódolások vannak külön-külön tesztelve, pl. BER kódolás (Basic Encoding Rules, ASN.1 bináris megjelenítése) esetén a lebegőpontos számok kódolásának tesztelésénél a 0 tesztelésére 14 különböző teszt van
    - b1-28 BERPDU28 ::= 0
    - b2-28 BERPDU28 ::= 0.0
    - b3-28 BERPDU28 ::= 0E0
    - b4-28 BERPDU28 ::= 0.0E0
    - b5-28 BERPDU28 ::= 0e0
    - b6-28 BERPDU28 ::= 0.0e0
    - b7-28 BERPDU28 ::= 0E+0
    - b8-28 BERPDU28 ::= 0.0E+0
    - b9-28 BERPDU28 ::= 0e+0
    - b10-28 BERPDU28 ::= 0.0e+0
    - b11-28 BERPDU28 ::= 0E-0
    - b12-28 BERPDU28 ::= 0.0E-0
    - b13-28 BERPDU28 ::= 0e-0
    - b14-28 BERPDU28 ::= 0.0e-0
- Különböző hibaüzenetek kezelésére is több tízezer vizsgált eset

Eclipse oldalon:

- Szemantikus ellenőrzés C++ oldal képességeinek ellenőrzése
- Több tízezer elvárt hibajelzés ellenőrzése
- Hibajelzések hiányának jelzése helyes kód esetén
- 1-1 file tartalmaz nagyon sok típushoz kódot, amikor valaki bővíti a teszteket leellenőrzi, hogy jó helyre jó hibaüzenetek kerülnek-e, utána a rendszer automatikusan ellenőriz
- a Javas eszköz felhasználja a C++hoz készített teszteket, mert ugyanarra a bemetetre ugyanolyan érzékelhető kimenetet kell előállítani

Robert C. Martin Clean Code könyve alapján a teszteknek a FIRST szabályt kell követniük, azaz magyarra fordítva:

- a teszteknek gyorsan kell futniuk
- a teszteknek egymástól függetlennek kell lenniük, mert ha az egyik failel ne rontsa el a többit
- bármely környezetben megismételhetőnek kell lenniük, azaz platformfüggetlen teszteket kell kialakítani, hogy bárhol bárki tudja a programokat gond nélkül tesztelni
- Boolean (logikai) kimenetű teszteket kell írni: pass vagy fail, legyen egyértelmű és ne kelljen hozzá az egész log filet elolvasni
- Figyelni kell az időzítésre, éppen a kódrészlet megírása előtt kell megírni a unit testeket, különben utólag nehezebb lesz tesztelni

(MARTIN, R.C.)

6 Összefoglalás

Szakmai gyakorlatom során rengeteget fejlődtem. Nem csak szakmailag, de emberileg is. A fejlesztés során nem csak a kód számit, hanem a hozzáállás. Az együttműködés, a hibák beismerése és a megfelelő kérdések feltétele. Nagy kihívás volt részt venni egy ilyen egy ilyen összetett rendszer fejlesztésében. Véleményem szerint minél részletesebben megismer az ember valamit annál jobban megszereti azt.
Érdekes volt betekintés szintjén látni a Titan és a compilerek felépitését és különböző kutatásokról hallani, résztvenni a TTCN-3 tanfolyamon és a különböző Ericssonos rendezvényeken.

Lehetőségem volt jobban megismerni a Sigma Technology és az Ericsson által fejlesztett Legóautó és ezáltal az IOT eszközök működését is. Az IOT internet of things eszközök és a M2M (machine to machine) kommunikáció megkönnyítheti az ember életét és javíthatja az élet minőségét is. Különböző területeken hasznos lehet: például a logisztikában az optimális útvonal meghatározásában. Használhatják kórházak vagy emberek saját biztonságuk vagy egészségük megőrzése érdekében.

A TTCN-3 kurzuson megtanulhattam az egyetlen szabványos tesztelési nyelv alkalmazási módjait. A TTCN-3 nyelvet biztonsági tesztelésekre, a teljes 4G-5G hálózat tesztelésére használják. Használja a TETRA, az ARMOUR, különböző middleware platformok (ATM, DSL), mobil és vezetékes kommunikáció (LTE,wiMaX, 3G, GSM, ISDN,SS7), okos kártyák, ePassport. Szakmai gyakorlatom során amellé a fordito mellé amely TTCN3ról C++-ra fordít készül egy olyan fordito, amely TTCN-3ról JAVA-ra tud fordítani. A Titan fejlesztését 2000-ben Szabó János Zoltán indította el, 2003 óta hivatalosan elfogadott és támogatott teszt eszköz az Ericssonban. Napjainkban már a legnagyobb német kutatóintézet a Fraunhofen Fokus is Titánt és TTCN-3at használ az IOT biztonsági tesztelésre.

A compiler lényegi működése során a lexikális elemző a szöveget tokenek listájára bontja fel, majd elkésziti az AST-t, vagyis a fát. A szemantikus ellenőrzésnél kiszűri az adattípus ütközéseket. A TTCN-3 nyelv tesztvégrehajtó környezetének azért érdemes a Titánt választani, mert amellett, hogy gyors és tudja teljesíteni a 4G hálózatok követelményeit, követi a szabvány változásait. Sok extrával került ki az opensource szoftverek közé, például számos protokoll támogatására készített belső termék is ingyen elérhetővé vált.

Az opensource nyílt forráskodú szoftvert jelent, azaz az ügyfelek teljes kontrollal rendelkeznek. Másolhatják, tökéletesithetik, futtathatják és tanulmányozhatják. A Titanról szóló fejezetben bemutattam az Eclipse Titan komponenseit is.
A TTCN.-3 nyelvet különböző teszteléshez elengedhetetlen szempontok alapján vizsgáltam meg, kiemeltem mivel ad többet a hagyományos programnyelveknél. Az egyetlen szabványos, ETSI által szabványosított, nem objektumorientált alapvetően protokollok tesztelésére kitalált nyelv különböző mintaillesztési mechanizmusok használatára alkalmas. A match függvény használatával értéket hasonlítunk template-tel. Kezeli a verdicteket.
Támogatja az altípusokat. Az alap típusok mellett olyan beépített típusok jelennek meg, mint a bitstring, octetstring,hexstring. Ez azért hasznos, mert sok protokoll octetekben írja le, hogy mit kell csinálni. A snapshotkezelés a rendszerkonfigurációval kapcsolatos. A tesztelt rendszer megfelelő működésének vizsgálatához hozzájárul például a timer kezelés is.

TTCN-3 esetében fontos kiemelni az újrafelhasználhatóság kérdését. A szabványosítás egyértelműsíti az elvárásokat. Egy nemzetközileg elfogadott tesztsorozatot futtatnak le az eszközökre. Ennek hatalmas üzleti értéke van. Ugyanazon tesztsorozat lefuttatásával pedig könnyű az összehasonlítás is.
A szabványos teszteknek nagyobb az előállítási költsége, mint a házon belüli teszteknek, de ez a költség szétoszlik az összes résztvevő között. Még mindig sokkal hatékonyabb ezt a módszert alkalmazni, minthogy minden résztvevő külön-külön kifejlessze a teszteket.

Minden TTCN-3 típusból lesz egy osztály, amik így hierarchiát alkotnak. Mindegyik osztályban egyenként meg kell írni a műveleteket. Például a TitanInteger osztályban az összeadást, kivonást, szorzást, osztást és a különböző operátorokat.
A moduláris osztás elkészítésénél először átgondoltam, hogy mi is az, aztán logikalilag felépítettem: az osztó nem lehet nulla, ha negatív számmal osztunk akkor az eredeti szám is negatív lesz.

A TitanFloat logolásának intervallumellenőrzésénél a cél az volt, hogy a végtelen és a nem szám eseteket helyesen tudjuk kiírni.
A TitanCharString osztály esetén módosítható karakterláncokat kell elképzelni.
A számomra legfontosabb fejezetben (Loggolás) nem csak a feladatomat mutattam be, hanem a loggolás eredetét is. A hajósok által leírt hajónaplóba események kerültek rögzítésre. Leírták például a vészhelyzeteket és kezelésüket, a pontos dátumot és az eltelt időt. Napjainkban előforduló példákkal illusztráltam a loggolást, az egyszerű megközelítésmódtól eljutottam az összetettebbig. A légi közlekedésben a pilótáknak mind a mai napig ajánlott „LogBook”-ot vezetniük, ugyanis ezt állásinterjúkon gyakran kérik tőlük. Itt könnyebben átláthatóak a megtett utak és a teljes repülési idő is.

Elektronikus megoldások is érkeztek a logolás megkönnyítésére, például a logisztikában sok sofőr időt spórolt a papíros log bejegyzések elhagyásával, mások viszont az egyéni életritmusuk tartását nehezen igazítják egy szoftverhez. Tulajdonképpen ha belegondolunk a beléptetőrendszerek is logfileokat generálnak: rögzítik, hogy mikor és ki lépett be az adott területre.

Ezután biztosan arra gondol az olvasó, hogy miért kell ezeket rögzítenünk? Ha nem történt hiba feleslegesnek tűnhet, de gondoljunk arra az esetre amikor hiba történik. A való életben gondolhatunk például arra, hogy valaki védőfelszerelés nélkül megy munkavédelmi területre és baja esik. A beléptetőrendszer segítségével visszakereshetjük, hogy mikor és kivel tartózkodott ott. A hibáknak a tesztelés során szeretnénk tudni az okát és ehhez elengedhetetlenek a múltbeli események.
Hibakeresés, tesztelés, hibamegelőzés esetén egyaránt hasznos. Levezettem hogyan értettem meg a belső működését és milyen logikai gondolatmenet kellett ahhoz hogy elkezdődhetjen a kódolás. A loggolásnál az összetett struktúrákra is gondolni kellett, az üzeneteket bufferelni kellett és a felhasználó beállításaira is figyelni kellett. A consolera logolás végső kódját és az ötleteimet is bemutattam. Összehasonlítottam a fileba és a konzolra logolást.

Külön fejezetben foglalkoztam azzal, érthetően szemléltetve hogy a felhasználó milyen beállításokat eszközölhet egy logfile esetén. Ezeket egyenként magyarázattal láttam el.
Log() függvényeket különböző osztályokban is létre kellett hozni, az egész számok és a karakterek logolását részletesen elmagyaráztam. Példakóddal és teszteléssel.
A karakterek esetén vizsgáltam a nyomtatható és nem nyomtatható karakterek és a template matchelés is.

Szakdolgozatom legfontosabb része, ahol egy leegyszerűsített log látható. Ezen kívül konfigurációs file részletről és való életben használt nem Titán specifikus konfigurációs fileból is látható részlet. A logfile nevében lévő metakarakterek is különböző jelentéssel bírnak, ezeket táblázatba foglalva magyaráztam meg.
Az utolsó fejezetben a tesztelés fontosságára hívtam fel a figyelmet. Dawn Haynes előadása alapján bemutattam mi az az Agile gondolkodásmód és a különböző teszttervezési technikákról is írtam Dibuz Sarolta és Csopaki Gyula cikkje alapján. Szakdolgozatomban alfejezetet kapott a Test Driven Development és érdekességként a „TDD a Titánban”.
Szakdolgozatom egyfajta útmutatóként szolgál leendő fejlesztőknek és hallgatótársaimnak.

7 Irodalomjegyzék

Aho, A. L. (2013). *Compilers: Principles, Techniques and Tools.* Pearson.

Armour EU projekt. (letöltve 2018 február.) Forrás: https://www.armour-project.eu/wp-content/uploads/2016/08/D22-Test-generation-strategies-for-large-scale-IoT-security-testing-v1.pdf

Bíró, S. (letöltve: 2018. május). *Agilis Szoftverfejlesztés.*

Colin Willcock, T. D. (letöltve: 2018 január.). *TTCN-3.*

Compiler Definíció. (letöltve: 2018. május).

Forrás: https://www.techopedia.com/definition/3912/compiler

Dibuz Sarolta, C. G. (letöltve: 2018. május). Infokommunikációs protokollok.

Magyar Tudomány.

Dr. Erős, L. TTCN-3 Mérés (letöltve: 2017. december). *http://qosip.tmit.bme.hu/foswiki/bin/view/Meres/TTCN*.

Eclipse. (letöltve 2018.április). *Eclipse hivatalos fóruma*. Forrás: https://www.eclipse.org/forums/index.php?t=thread&frm_id=297

EclipseFórum.

https://www.eclipse.org/forums/index.php/t/1073729.

https://www.eclipse.org/forums/index.php/t/1086353/

https://www.eclipse.org/forums/index.php/t/1083842/).

https://www.eclipse.org/forums/index.php/t/1090226.

https://www.eclipse.org/forums/index.php/t/1076112/.

https://www.eclipse.org/forums/index.php/t/1082243.

Ericsson. (2013). *TTCN-3 course material.*

Fraunhofer Fokus Titant használ. (letöltve 2018. április). Forrás: https://www.fokus.fraunhofer.de/de/fokus/news/bosch-connected-world-internet-der-dinge_2018_02

Grenning, J. W. (letöltve: 2018. február). *Test-Driven Development for Embedded C.*

Haynes, D. (2017). *Being more agile without doing agile.* HUSTEF 2017, Budapest, Magyarország.

Intelligens szállítmányozási rendszerek. (letöltve 2018. május). Forrás: http://www.ttcn-3.org/index.php/downloads/publicts/publicts-etsi/65-publicts-its

Louden, K. (dátum nélk.). *Compiler Construction Principles and practice.*

M2M. (letöltve: 2018. március). Forrás: https://www.telekom.hu/uzleti/szolgaltatasok/egyeb-szolgaltatasok/m2m

M2M tesztelés. (letöltve 2018.május). Forrás: http://www.ttcn-3.org/index.php/downloads/publicts/publicts-etsi/132-smartm2m-test-suites

Martin, R. C. (dátum nélk.). *Clean Code.*

MQTT. (letöltve 2018. május). Forrás: https://www.hivemq.com/blog/how-to-get-started-with-mqtt

OneM2M. (letöltve: 2018. március). Forrás: http://www.ttcn-3.org/index.php/downloads/publicts/publicts-onem2m/128-publicts-onem2m-core

Saner. (letöltve: 2018. május). https://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8326467.

Szabados, K. (2014. február). Advanced TTCN-3 Test Suite validation with Titan.

Szabó J. Z, C. T. (letöltve 2018. március). Titan, TTCN-3 test execution environment. *Híradástechnika*, old.: http://www.hiradastechnika.hu/data/upload/file/2007/2007_1a/HT_0701a-6.pdf.

Szabó János Zoltán, C. T. (letöltve 2018). Titan, TTCN-3 test execution environment. *Hiradástechnika*.

szórólap, Fraunhofer Fokus (2018. május). Forrás: https://cdn0.scrvt.com/fokus/8b97fcd0ae224ca2/9694adadb60c/IoT_Flyer_NEU_140218_EN.pdf

TTCN-3 konformancia tesztelés. (dátum nélk.). Forrás: http://www.ttcn-3.org/index.php/downloads/publicts/publicts-etsi/67-publicts-epassport

TTCN-3. (letöltve: 2017. december). Forrás: http://www.ttcn-3.org/

Zoltán, S. J. (letöltve: 2018. január). Experiences of TTCN-3 Test Executor Development. *https://link.springer.com/content/pdf/10.1007/978-0-387-35497-2_14.pdf.*

8 - Ábrák, képek, táblázatok jegyzéke