# Tartalom



*Címlap: Korabeli telefonközpont az Andrássy úti Postamúzeum gyűjteményéből*

# Szoftverek a távközlésben

*gyori@tmit.hu*

Szeptemberi számunkban a távközlési szoftvereké, távközlési protokolloké a főszerep. Szinte hihetetlen, hogy az első tároltprogram-vezérlésű telefonközpont üzembe helyezése óta (1965) mindössze 41 év telt el, s azóta a távközlési berendezésekben alkalmazott szoftverek milyen változásokon mentek keresztül. Ami viszont négy évtized elmúltával sem változott, az a távközlési szoftverekkel kapcsolatos legfontosabb elvárás: a nagy megbízhatóság. Így már az első (1 ESS) központban működő vezérlő is duplikálva volt, hogy minimálisra csökkentsék a leállás veszélyét.

Az idők folyamán aztán a tároltprogram-vezérlésű központok egyre komplexebb rendszerré váltak. Miután az egyes funkciókat szétválasztották, egyre több processzor összehangolt munkája végezte a kapcsolóközpontokban szükséges tevékenységeket. Emellett az egyes kapcsolóközpontoknak nem volt elég önmagukban hibátlanul működni, hanem biztosítani kellett az együttműködést más központokkal is A tároltprogram-vezérlésnek köszönhetően újabb és újabb szolgáltatások váltak népszerűvé. Ezeket a szolgáltatásokat az előfizetők nemcsak központon belül, hanem az egész hálózatban szerették volna használni. Erre a digitális központok térnyerésével vált igazán lehetőség. Mindebből következik, hogy a központok közötti együttműködést szabályozó protokollok is egyre bonyolultabbak lettek.

A távközlési szoftverekben megtörtént a gyártó-specifikus, vagy mai divatosabb szóval élve platform-függő és az együttműködést valamint a funkcionális működést leíró részek megvalósításának a szétválasztása. Ha leegyszerűsítve szeretnénk fogalmazni, az egyes berendezésekben működő távközlési szoftverek megfelelnek a funkciókat leíró protokollok implementációjának. A folyamatokat leíró protokollok formalizálására szükség volt a leíró nyelvekre. Az első, már általánosan használt leíró nyelv az SDL volt, amit ma is széles körben használnak, immár nemcsak a távközlés területén.

A protokollok leírása mellett a helyes működés vizsgálata is egyre fontosabb kérdéssé vált. A távközlési szoftverek fejlesztésénél egyre nagyobb hangsúlyt kapott a tesztelési folyamat, majd annak automatizálása. Nem véletlen, hogy ebben a számban is több cikk foglalkozik a tesztelési folyamattal:

A „hagyományos” távközlő hálózatokban a tudás mindig a telefonközpontban volt, az előfizető telefonkészüléke kezdetben semmi, később minimális tudással rendelkezett. A napjainkban lezajló változás, az internet és a távközlő hálózatok konvergenciája azt eredményezi, hogy a korábban a kapcsolóközpontokban koncentrálódó intelligencia szétosztódik. Az okos telefonokban, számítógép terminálokban kell megvalósítani, implementálni azt a tudást, amivel a többi hasonló terminállal kapcsolatot tudunk teremteni, mely kapcsolat képessé teszi a berendezést arra, hogy adatot továbbítson, fogadjon más terminálóktól. Ennek egyik speciális módja az úgynevezett peer-to-peer kapcsolat létesítése, ami lehetővé teszi a végpontok közötti közvetlen adatátvitelt. Egy ilyen alkalmazás megvalósítása nem a csak számítógépek között lehetséges, hanem erre alkalmas szoftverek felhasználásával mobiltelefonokba is telepíthető.

Első mondatainkban már említettük, hogy a távközlési szoftverekkel szemben támasztott követelmények igen magasak. Ezért a nyílt forráskódú szoftverek alkalmazása a távközlés területén nem igazán volt jellemző. A Carrier Grade Linux megjelenése óta a távközlési gyártók is komolyan vehetik a nyílt forráskódú szoftvereket. Egyik cikkünk azt is bemutatja, hogyan kísérletezhetünk otthon saját fejlesztésű telefonközpont építésével.

*Győri Erzsébet,*
*BME Távk. és Telematikai Tsz.*
*vendégszerkesztő*

*Szabó Csaba Attila,*
*főszerkesztő*

# Modern szoftverfejlesztési irányok: integrált tesztelés

RÉTHY GYÖRGY

*Ericsson Magyarország Kft.*
*gyorgy.rethy@ericsson.com*

***Kulcsszavak: szoftverfejlesztés, integrált tesztelés, tesztkomponensek, keretrendszer, TTCN-3***

*A cikkben áttekintjük a bonyolult szoftverek fejlesztésének tipikus folyamatát azért, hogy megfogalmazhassuk a fejlesztés hatékonyságát növelő tesztmegoldásokkal szembeni elvárásokat és megvizsgáljuk, hogy a TTCN-3 miképpen felel meg ezeknek az elvárásoknak.*

## 1. Bevezetés

Idézzük fel a klasszikus viccet: „Jót, gyorsan, olcsón. Ön ebből kettőt választhat." Persze mint az élet más területein is, a távközlési szoftverek fejlesztése terén sem ilyen egyszerű a képlet. Közismert, hogy a távközlési szoftverek bonyolultsága gyorsan nő, amihez éles piaci verseny társul. A vezető eszközfejlesztő cégek – beszéljünk akár hálózati eszközökről, akár végberendezésekről – szempontjából létfontosságú, hogy egy-egy új funkció az ő szoftverükbe építve jelenjen meg először a piacon. Mindemellett a versenyben az egyre bonyolultabb szoftverek minőségét nem csak megőrizni, hanem még növelni is szükséges. Ez új kihívásokat jelent a bonyolult szoftver-rendszerek fejlesztőinek, melyekre technológiaváltással kell felelniük. Jelen cikk arra a kérdésre keresi a választ, hogy ez a váltás milyen hatással van a szoftverfejlesztésben alkalmazott tesztelési megoldásokra.

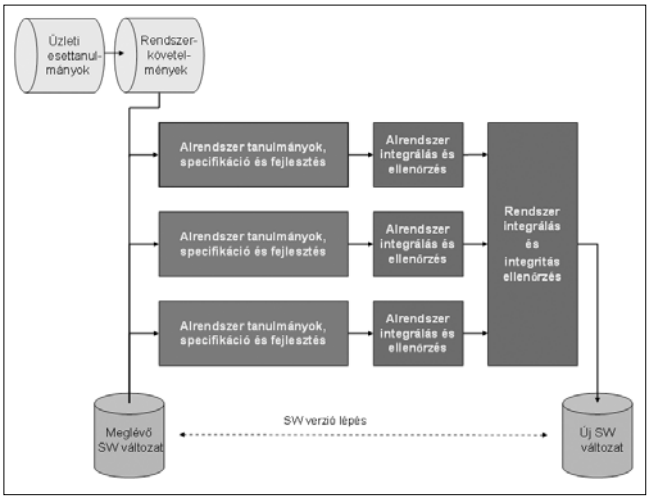
## 2. A szoftverfejlesztés folyamata

Ehhez először a szoftverfejlesztés folyamatát kell megismernünk. A könnyebb kezelhetőség érdekében minden bonyolult rendszert kisebb alrendszerekre kell osztani. Például a hálózati csomópontokban a különböző protokollok kezelését, a hívásvezérlési logikát, előfizetői adatbázis kezelést, számlázási információk kezelését, a csomópont menedzselési funkcióit külön alrendszerek valósítják meg. Egy-egy nagyobb új hálózati képesség bevezetése álltalában valamilyen üzleti modell, üzleti esettanulmányok (business use cases) alapján történik. A változások rendszerint több hálózati eszközt érintenek, ezek fejlesztése összehangoltan kell történjen.

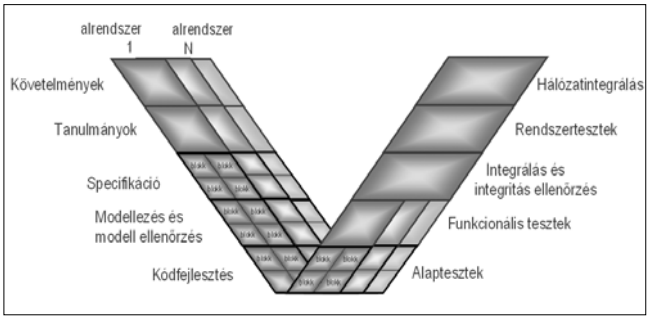
Ezért az üzleti esettanulmányokat először az egyes rendszerekre vonatkozó követelményekre kell lebontani, majd ezeket leképezni az érintett alrendszerek által teljesítendő követelményekre *(1. ábra)*. Ezután lehet az egyes alrendszerekben elvégezni a szükséges szoftverfejlesztéseket. Az egyes alrendszerek elkészülte és külön-külön történő ellenőrzése (tesztelése) után az alrendszereket újra teljes rendszerré kell integrálni és ellenőrizni a rendszer komplett működését is. Az egyes alrendszerek fejlesztése és tesztelése párhuzamosan történik, azokon külön fejlesztői csoportok dolgoznak, gyakran más-más országokban.

Az alrendszer szintű szoftverfejlesztés ismertetéséhez segítségül hívjuk az úgynevezett V-modellt. Ez nem más, mint nemzetközileg is elismert német szabvány a szoftvertermékek életciklusának kezelésére [1]. Ez természetesen magában foglalja a fejlesztési folyamatot is, melyet a *2. ábra* kicsit egyszerűsítve és a távközlési területre specializáltan mutat be.

*1. ábra Távközlési szoftver fejlesztés főbb lépései*




*2. ábra A távközlési szoftverek fejlesztésének folyamata*



A V-modellnek természetesen része, de az egyszerűség kedvéért a 2. ábrán külön nem ábrázoltuk az alrendszer követelmények születésének folyamatát; a követelményeket fogjuk fel inkább a fejlesztés bemenő adataként. A bevezető végén feltett kérdés megválaszolásához szintén lényegtelenek a tanulmányi és a specifikációs fázis részletei, kivéve azt, hogy az alrendszer követelményeket tovább bontják szoftver blokk szintű funkciókra és a blokkok közti kommunikáció követelményeire. Ez rendszerint egyszerre jelenti meglévő blokkok továbbfejlesztését és új blokkok megjelenését az alrendszerben.

A modellezés és a modell ellenőrzése (verifikáció) természetszerűleg csak olyan szoftverfejlesztési folyamatban van jelen, amelyiknél valamilyen formális modellt használnak, mint amilyen az SDL, UML stb. Jelen cikkben ezzel a területtel nem foglalkozunk külön. A modellből történő teszt generálásnak gazdag irodalma van, a gyakorlatban eddig mégsem nem terjedt el széles körben.

Egyrészt a fejlesztés alatt álló programblokk működését leíró modellek legtöbbször nem, vagy csak korlátozottan alkalmasak teszt generálásra, ezért ilyen esetekben a teszteléshez egy külön modellt kellene fejleszteni. Másrészt kevés, nagy szoftverrendszereknél is megbízhatóan és hatékonyan használható eszköz áll rendelkezésre. Amit ezzel kapcsolatban meg kell még jegyezni, hogy a modellből történő tesztgenerálás esetén a modell mindenre kiterjedő ellenőrzése különösen kritikus, hiszen a modellben lehetnek olyan logikai hibák, melyeket a belőle generált tesztrendszerrel nem lehet felfedezni.

A kódfejlesztés fázisában történik a meglévő blokkok módosítása és az új programblokkok kódjának tényleges megírása. Az egyes blokkokat külön programozók vagy programozói csoportok fejlesztik. Ehhez a fázishoz szorosan kapcsolódik az alaptesztelés, melyet természetszerűleg az adott blokk programozói hajtanak végre. Ennek két alapvető módszertana van. Az úgynevezett „fehér doboz tesztelés” (white box testing) azon alapul, hogy a tesztelt blokk belső működése, algoritmusa teljes mértékben ismert. A programozó lépésről lépésre futtatja le a kódot, folyamatosan ellenőrizve, hogy az az elképzeléseknek megfelelően működik-e. Ehhez egyrészt a szabványos fejlesztői környezet részét alkotó debuggerek, másrészt – programozási nyelvtől függően – egyéb fejlett szoftvertesztelő eszközök állnak rendelkezésre, mint például a JUnit (Java-hoz), vagy ennek változatai más nyelvekre (NUnit C#-hoz, PyUnit Python-hoz, CPPUnit C++-hoz), vagy az IBM Purify a C/C++ kódok memóriakezelésének ellenőrzéséhez.

Olyan szoftverblokkoknál, melyek rendelkeznek definiált alkalmazási programozói interfésszel (API), a fekete doboz tesztelési módszer (black box testing) is alkalmazható. Ekkor nem használunk a blokk belső működésére vonatkozó információt, hanem annak definiált külső viselkedését, az interfészein a különböző gerjesztésekre adott válaszait vizsgáljuk.

A funkcionális tesztek során azt vizsgálják, hogy az egyes blokkokból összeállított alrendszerek funkcionálisan megfelelően működnek-e. Ezen teszteket nem a programozók, hanem külön e célra kiképzett tesztelők végzik. A távközlésben használt HW eszközök gyakran nagy megbízhatóságú drága eszközök, valódi idejű operációs rendszerekkel. Elsősorban anyagi okokból az alap- és a funkcionális teszteket gyakran nem ezeken hajtják végre, hanem a majdani operációs rendszert szimuláló, szabványos informatikai eszközökön (PC/Linux vagy UNIX/Solaris platformok) futtatható szimulátorokon.

Az integrációs fázisnak két alapvető célja van. Egyrészt az egyes alrendszerek ellenőrzése a tényleges cél-HW eszközökön, másrészt az alrendszerek teljes rendszerré integrálása és a komplett rendszer működésének ellenőrzése. Másképp fogalmazva tesztelik, hogy a rendszer teljesíti-e folyamat elején megfogalmazott funkcionális követelményeket.

A rendszertesztek során a nem-funkcionális követelményeket vizsgálják, mint például a kezelt előfizetők száma, forgalmi, stabilitási, megbízhatósági jellemzők, szoftverváltási eljárások megbízható működése stb. Ezeket a vizsgálatokat természetesen a célhardveren futtatott teljes szoftverrendszeren végzik.

Míg az eddigi folyamatok egy adott hálózati eszköz kifejlesztését célozták, s azt vizsgálták, hogy annak interfészei és egyéb paraméterei megfelelnek-e a vele szemben támasztott követelményeknek, a hálózatintegráció célja az eszköz hálózatba integrálása, annak vizsgálata, hogy képes-e más eszközökkel együttműködni (eltekintve a korábbi fázisokban rejtve maradt hibáktól, leegyszerűsítve a különböző eszközök követelményeinek kompatibilitását teszteli). Ebben a fázisban nem csak az adott gyártó, hanem különböző gyártók eszközeinek együttműködését is vizsgálják.

Mint láthatjuk, a folyamat több különálló tesztelési fázist tartalmaz, melyek azonban integráns részei a folyamat egészének. Természetesen, a gyakorlatban ezek nem különülnek el élesen, több fázis rekurzív részfolyamatot alkot. Említhetnénk a kódfejlesztés és az alaptesztelés viszonyát, ahol az alapteszteket általában maga a kód programozója hajtja végre és azonnal javítja is saját kódját.

De szemléletesebb példa, hogy a funkcionális tesztek alatt talált hibákat a tesztelők jelentik a kódfejlesztőknek, akik azokat kijavítják, elvégzik a szükséges alapteszteket, majd a javításokat megküldik a funkcionális tesztet végzőknek, akik ellenőrzik a javítás helyességét. Itt jelentkezik egy érdekes probléma. Nem elég ugyanis arról meggyőződni, hogy a hibát tényleg kijavították-e, azt is ellenőrizni kell, hogy a javítás nem vitt-e újabb hibákat a szoftverbe. Vagyis az egyszer már sikeresen végrehajtott teszteket újra végre kell hajtani, s azoknak a korábbi eredményt kell adniuk. Ezt regressziós teszteknek hívják s így a funkcionális tesztek fázisa valójában két tevékenységi kört takar.

Bonyolult szoftvereknél egy új változat fejlesztési folyamata meglehetősen időigényes, gyakran elérheti a

másfél-két évet. Mint a bevezetőben említettük, a mielőbbi piacra kerülés minden cég alapvető üzleti érdeke, így a folyamatokat – a termék minőségének megőrzése mellett – le kell rövidíteni. Első gondolatunk lehetne, hogy hatékonyabb projektszervezéssel, az egyes fázisok rövidítésével ezt el lehet érni. Ez azonban kétélű fegyver. A projektek hatékonysága mindig is szempont volt a cégek működésében, s az utóbbi években különösen sok figyelmet kapott. Így egyedül ettől valójában jelentős eredmények nem várhatók, míg az egyes fázisok lerövidítése magában hordozza a szoftverminőség romlásának veszélyét (feszített kódfejlesztési határidők, elégtelen idő a tesztelésre stb.) Más utakat kell tehát keresni.

Az egyik ilyen lehetőség az egyes fázisok párhuzamosítása. Ennek egyik megközelítése az úgynevezett integrálás-központú fejlesztés (Integration Centric Engineering, ICE). Arról van szó, hogy a hagyományos fejlesztési modell szerint egy új, n+1-edik szoftver változat kifejlesztése az

$$SV_{n+1} = ( SV_n + \Delta )$$

képlet szerint történik, ahol „$SV_n$" a meglévő szoftverváltozatot, „$SV_{n+1}$" a kifejlesztendő új változatot, míg $\Delta$ a jelenlegi fejlesztési fázisban hozzáadandó új funkcionalitást jelöli. Ezt a folyamatot a *3. ábra a)* része mutatja, ahol a 2. ábra szerinti fejlesztési fázisok sorban követik egymást. Az ábrán Qn szimbolikusan, negyedévekben jeleníti meg az idő folyamatát.

De mi van akkor, ha nem szükséges a teljes $\Delta$ funkcionalitást megvalósítani egy működőképes közbenső szoftver változat (jelöljük pl. $SV_{n.a}$ -val) előállításához? Például, ha $\Delta$ a SIP protokoll szerinti híváskezelést jelöli, a felépítési fázist (INVITE, 200 OK és ACK üzeneteket) megvalósítva egy működőképes **n.a** változatot kapunk. Ebben az esetben a funkcionális tesztek megkezdéséhez nem szükséges megvárni a teljes $SV_{n+1}$ szoftver változat elkészültét, a felépítési fázis tesztjeit megkezdhetők az $SV_{n.a}$ változaton is. Ez esetben a felépítési funkció tesztjeit a bontási funkció kódfejlesztésének kezdetével egyidőben lehet elkezdeni.

Az ICE szerinti folyamatban, melyet a *3. ábra b)* része szemléltet, a szoftverfejlesztés képlete

$$SV_{n+1} = ( SV_n + \delta_1 + \delta_2 \ldots \delta_i)$$

formára módosul, ahol $\delta_j$ jelöli az egyes, önállóan megvalósítható részfunkciókat.

Természetszerűleg

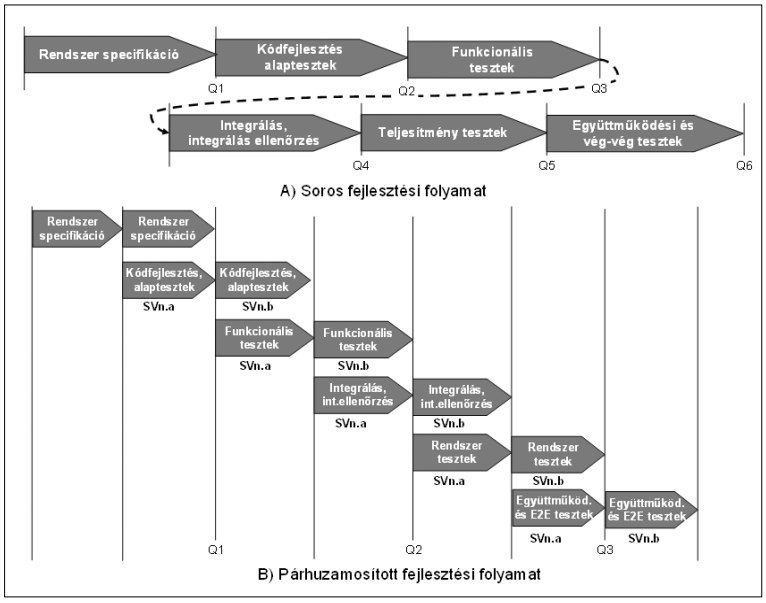
$$\Delta = \sum_{1}^{i} \delta_j \ .$$

Vagyis az ICE az egyes szoftver fejlesztési fázisok kisebb ütemekre bontásán alapul. Az ábrából is jól látható, hogy a teljes folyamat a kódfejlesztési és a tesztelési fázisok párhuzamosítása következtében jelentősen lerövidült.

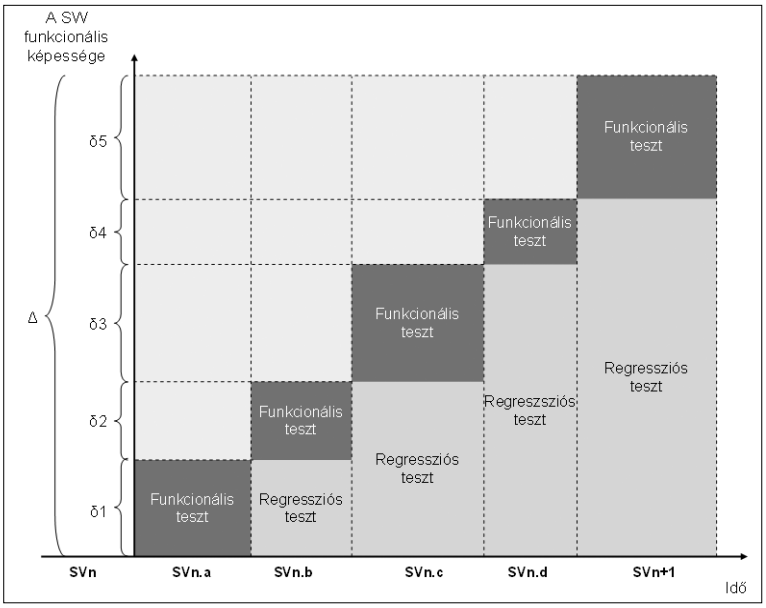
Az ICE alkalmazása azonban új probémákat is felvet. A gyakorlatban az ICE folyamat egyes ütemeiben fejlesztett funkciók, $\delta_k$ és $\delta_l$ nem teljesen függetlenek egymástól, a $\delta_k$-hoz fejlesztett szoftver blokkok valamilyen mértékben módosításra szorulhatnak $\delta_l$ fejlesztése alatt. Így a későbbi fejlesztési ütemekben szükség van a korábbi ütemekben már tesztelt szoftverblokkok regressziós újratesztelésére.

Ahogyan az a *4. ábrán* is jól látható, az elvégzendő regressziós tesztek mennyisége a fejlesztés későbbi ütemeiben robbanásszerűen megnő a korábbiakhoz képest. Ez különösen kritikus a késői, de aránylag kevés új funkciót tartalmazó közbenső változatoknál, mint például az ábra $SV_{n.d}$ tesztelési ütemének esetében, ahol a $\delta_4$ funkcionális tesztjei mellett legfeljebb

*3. ábra Fejlesztés rövidítése ICE-szal*




*4. ábra Funkcionális tesztelés alakulása az ICE folyamat fázisaiban*



ugyanannyi idő alatt kell a $\delta_1 ... \delta_3$ funkciók regressziós tesztjeit is végrehajtani. Ez a probléma természetesen nem csak a funkcionális teszteknél, hanem ugyanígy az integrációnál és az integritás ellenőrzésénél is előáll.

A fentiekből könnyen látható, hogy a hagyományos tesztelési módszerek alkalmazásával az ICE model nem, vagy csak komoly kompromisszumokkal használható. Vagy a tesztelési fázisokban rendelkezésre álló emberi és gépi erőforrásokat kell jelentősen növelni (költségnövekedés) vagy – a termék minőségét kockáztatva – a regressziós tesztek mennyiségét minimalizálni. Igazából egyik út sem kívánatos.

## 3. Milyen a jó tesztrendszer?

Mint láttuk, a különböző tesztelési tevékenységek a teljes szoftverfejlesztési folyamban meglehetősen nagy részt képviselnek, mind idő, mind a szükséges erőforrások szempontjából. Így érthető, hogy közelről sem elegendő az, ha egy adott teszteszköz alkalmas a kérdéses funkcionalitás vizsgálatára. Az alkalmazott folyamathoz jól illeszkedő tesztmegoldások sokat javíthatnak egy fejlesztési projekt mutatóin, míg a folyamatban bekövetkezett változásról tudomást nem vevő, „hagyományos" tesztelési módszerek ellenkezőleg, még ronthatják is ezeket.

A fentiek ismeretében meg tudjuk fogalmazni a hatékony tesztrendszerrel szembeni követelményeket is:

**1) Automatizált teszt végrehajtás és kód újrahasznosítás**

Habár első látásra ez két külön követelménynek tűnhet, valójában szorosan összefüggnek. Gondoljunk csak a 4. ábrán látott szituációra. Az ICE szerinti fejlesztések hatékonyságában meghatározó, hogy milyen gyorsan és mekkora erőforrásokkal hajthatók végre a regressziós tesztek az egyes (főleg a későbbi, pl. $SV_{n.d}$) ütemekben.

Mind a tesztvégrehajtás sebességét, mind erőforrásigényét automatikus tesztvégrehajtással lehet kordában tartani. Az automatikus tesztvégrehajtás mindig valamilyen teszt kódhoz kapcsolódik, melyet először létre kell hozni, hogy aztán a kódot lefuttatva a teszt automatikusan végrehajtható legyen. Ebből a szempontból másodlagos, hogy a kódot miképp hoztuk létre: modellből generáltuk-e, egy előző teszt végrehajtás menetét „vettük-e fel" és tároltuk el, vagy a futtatott kódot esetleg kézzel írták.

Az viszont nem lényegtelen, hogy mekkora előkészítő munka szükséges az egyes regressziós tesztek előkészítéséhez. Ez úgy minimalizálható, ha az előző ütemekben (példánkban $SV_{n.a} ... SV_{n.c}$) használt funkcionális tesztek változtatás nélkül, vagy minimális változtatással az aktuális ($SV_{n.d}$) ütemben automatikus regressziós tesztként is végrehajthatók.

De ugyanígy számottevő, hogy szimulált teszt környezetben – például a funkcionális teszteknél – használt tesztkód minimális változtatással alkalmazható legyen célhardver-környezetben is, például integritás vizsgálatokban.

**2) Egységesség és széleskörű használhatóság**

Minden cég alapvető érdeke, hogy az általa használt eszközpark minél kevesebb típusból álljon. Nincs ez másképp a szoftverfejlesztésben sem. Több szempont miatt is szükséges az eszközök egységesítése. Egyfelől ez változatlan mennyiség mellett is csökkenti az eszközparkra fordított teljes költséget csakúgy, mint az eszközök frissítéseire fordított költségeket. Gondoljuk csak meg, hogy a protokoll frissítéseket minden eszköztípushoz meg kell rendelni, s ennek ára – egy a piac előtt járó cég esetében – a teszteszköz szállítója által ráfordított teljes munkaköltség.

Másfelől szintén fontos, hogy a fejlesztés során az emberi erőforrásokat könnyen lehessen – tervezetten vagy eseti jelleggel – az egyik területről a másikra átirányítani (ennek egyik speciális esete a tesztelési fázisok összevonása, például az integritás ellenőrzési és rendszervizsgálatok egy részét a funkcionális tesztek fázisában végezni el, mely egy másik, az ICE-val nem ellenkező, sőt azzal jól integrálható lehetőség egy hatékonyabb fejlesztési modell alkalmazására). De ez csak akkor valósítható meg hatékonyan, ha a váltás során nem kell új eszközök használatát és új teszt módszereket megtanulni.

Az egységesítés a tesztelés esetében azt jelenti, hogy univerzális teszt megoldások kellenek, melyek a fejlesztés minden fázisában, a modell-ellenőrzéstől a hálózatintegrációs tesztekig (2. ábra), minden blokknál, alrendszernél és a komplett rendszernél is használható. És legyen ez igaz minden fejlesztett rendszer (mint a különböző hálózati eszközök, bázisállomások, rádióhálózat vezérlők, hívásvezérlők, hálózat-menedzselő és számlázási eszközök stb.) esetében is.

**3) Bonyolult rendszerek tesztelése**

A bonyolult szoftverrendszerek jellemzője, hogy egy-egy tesztnél számos interfészen kell egyidőben és koordináltan küldeni gerjesztő üzeneteket és értékelni a rendszer válaszait. Egy-egy bonyolultabb teszt eset több száz üzenetet is tartalmazhat és a válasz sokszor nem feltétlenül azon az interfészen érkezik, mint amelyiken a gerjesztő üzenetet küldtük. Gyakran egy-egy teszt végrehajtása közben kell a tesztelt szoftver beállításain változtatni, vagy ellenőrizni, hogy a szoftver megfelelően kezeli-e adatbázisait, például az előfizető állapotát, jelzéslinkek állapotát, számlázási információkat stb. Más szóval meg kell oldani az egyes interfészek közötti tesztkoordinációt is.

A hatékonyságot alig-alig segíti, ha csak egy-egy interfészt tudunk automatikus tesztfuttatással kezelni, vagy az interfészek között nincs automatikus koordináció. Ez utóbbi feladatot a tesztelést végző szakembernek kell ellátnia, ami fáradtságos és a hibalehetőségek gazdag forrása. Nem kell magyarázni, mennyire megnövelheti ez a teszteléshez szükséges időt.

Az elsőnek említett három követelmény azt hiszem, fontosságban minden más szempontot megelőz.

**4) Integrált tesztkörnyezetbe illesztés**

Tudni kell, hogy egy tesztet lefuttani önmagában nem elég. Általában a tesztelés előkészítése is összetett feladat és a végrehajtás után gondoskodni kell az eredmények utóéletéről. Létre kell hozni a tesztelt szoftver számára a megfelelő – szimulált vagy hardver – környezetet, ezt kapcsolni kell a használt tesztrendszerhez és mindkét oldalt megfelelően konfigurálni. A tesztelt szoftver konfigurációját sokszor tesztesetről tesztesetre változtatni kell. Ugyanígy tesztesetenként változhat a végrehajtáshoz szükséges teszt eszközök listája.

Tönkreteheti a tesztet, s így csökkentheti a hatékonyságot, ha futtatás közben egy másik tesztelő a saját tesztjeinek elvégzéséhez megváltoztatná például a szoftver beállításait vagy más célra kezdené használni a teszteszközöket. Ezért a teszt végrehajtásához a tesztelőnek elő kell jegyeznie a szükséges erőforrásokat, majd mikor azok felszabadultak le kell foglalnia, a teszt végrehajtása után pedig felszabadítania azokat.

Nyilvánvaló, hogy hatékony tesztelés úgy érhető el, ha ezeket a feladatokat egy tesztelést segítő rendszer automatikusan látja el. Ugyanígy a tesztek eredményeinek rögzítését, logok elmentését és a eredmények utóéletét (mely hibalapok születtek, érkezett-e javítás az adott hibára, újra lett-e tesztelve és annak mi lett az eredménye, statisztikák készítése stb.) is hatékonyabb egy megfelelő eszközzel segíteni, mint kézzel végezni. Az Integrált Fejlesztő Környezetek (Integrated Development Environment, IDE) mintájára a tesztelést segítő (grafikus) rendszereket Integrált Teszt Környezetnek (Integrated Test Environment, ITE) is nevezik. Ha viszont a tesztelő munkáját ITE segíti, fontos, hogy a teszt végrehajtása is innen történjék. Így a tesztelőnek csak egyetlen felületet kell (tudnia) kezelni.

**5) Gyors teszt-előkészítés**

Fent azt állítottuk, másodlagos, hogy miképp állítjuk elő az automatikus teszt végrehajtásnál használt kódot. Másodlagos a regressziós tesztek volumenéhez képest, de messze nem lényegtelen. Bizony ennek hatékonyságát is figyelembe kell venni amikor a hatékony tesztrendszerről beszélünk.

**6) Könnyű használhatóság**

A könnyű használhatóság valójában az egyszerű és gyors használatot jelenti, amikor „minden adja magát”. Ez sem elhanyagolandó tényező a teszt előkészítésének és végrehajtásának hatékonysága szempontjából.

## 4. Miért pont a TTCN-3?

A hagyományos teszteszközök a teszt automatizálást két módszerrel teszik lehetővé: vagy saját programozási nyelvük van, szkripting nyelvnek (scripting language) is hívják, vagy lehetővé teszik egy tesztfuttatás „felvételét”, szerkesztését és későbbi visszajátszását. Az előbbi többször szöveges, ritkábban grafikus, az utóbbi általában grafikus formában működik.

Vannak ezen kívül de facto-szabványként elterjedt leíró nyelvek, mint például a Python vagy az Expect, és az ezeken alapuló teszt eszközök. Ezek a megoldások szenvednek attól, hogy nem álltalános célú nyelvek, hanem valamilyen specifikus probléma megoldására találták ki őket, így egy-egy teszteszköz korlátozott számú interfészt és protokollt támogat. Vannak ugyan kivételek, de ezek az eszközök álltalában vagy szimulált vagy célhardver-környezetben történő használatra készülnek. Így ilyen eszközökből számos típus szükséges az összes tesztfeladat megoldásához. Ezen eszközökre szintén igaz, hogy a programozási lehetőségeik korlátosak.

Tipikusan az időzítők, az alternatív események (mint „A üzenet I interfészen” vagy „B üzenet J interfészen” esetek) kezelése, valamint a várt üzenetekben a helyettesítő karakterek (wildcards) használata problémás. Így ha két futtatás között valamilyen, a teszt szempontjából egyébként lényegtelen, információ megváltozik, az automatikus teszt végrehajtás – helytelenül – hibás ítéletet jelez. Ezen eszközök automatikus koordinálása nehezen vagy egyáltalán nem megoldható feladat.

Az egyik lehetséges megoldás ezen eszközök integrálása egy ITE-be. Ez megoldhatja a teszt koordinálást, de a programozási korlátokat nem oldja fel, s nem a használt eszközpark egységesítése felé mutat.

Mi hát akkor az üdvözítő ötlet, mely elvezet az általunk keresett megoldáshoz? A másik út egy szabványos teszt leíró programnyelv, amely elég rugalmas és kifejező ahhoz, hogy minden szoftvertesztelési területen használható legyen. Két ilyen nyelv is született, mégpedig a TTCN. Az előző mondat nem sajtóhiba eredménye. Megértéséhez a TTCN nyelv(ek) fejlődésének története adja meg a kulcsot (lásd a jelen számban közölt „Tesztelés a telekommunikációban” című cikket). A TTCN-2-ről TTCN-3 ra történő váltásnál a nyelvet jóformán teljesen újradolgozták (mint a fenti cikk is említi, még a neve sem a régi).

Megváltozott a megjelenítési forma. Míg a TTCN-2 táblázatos-sorbehúzásos formában írta le a kívánt dinamikus viselkedést, addig a TTCN-3 ezt „rendes” szöveges programnyelvként [2] és választhatóan grafikus formában [4] teszi (szabványosítottak egy táblázatos megjelenítési formát is [3], de a gyakorlatban nem terjedt el). Változott és kibővült a tartalom. A TTCN-3 lehetővé teszi a teszt konfiguráció változtatását teszteset-végrehajtás közben, a végrehajtás kontrollálását a TTCN-3 kódból, szinkron (API) interfészek tesztelését, valamint a nyelvi elemek területén számos más újítást és kiegészítést is tartalmaz (például check és interleave műveletek, de ezekkel itt nem foglalkozunk).

A TTCN-3 nyelv az Európai Távközlési Szabványosítási Szervezet (European Telecommunication Standards Institute, ETSI) keretében, de széles egyetemi-gyártói-

használói-szabványosítási együttműködésben alkották meg. A szabványnak jelenleg hét része van [2-8], de öt újabb rész (IDL, C, C++ és XML leképezések TTCN-3-ra, illetve TTCN-3 forráskód dokumentáció) van készülőben). Az új nyelv fejlesztésében az ETSI, a Lübecki Egyetem (ma ugyanaz a szakember a Göttingeni Egyetem professzora), a Fraunhofer FOKUS, a Nokia, a Siemens, a TestingTech és az Ericsson szakemberei vállaltak aktív szerepet, de beadványaikkal még számosan segítették a munkát.

A nyelv bővítése és karbantartása – alapvetően ugyanezen résztvevőkkel – ma is folyamatos. Alkalmazása ma már egyáltalán nem korlátozódik a távközlés területére, használják az autóiparban, a vasúti távirányításban, orvosi berendezések vizsgálatainál, de például .NET alapú általános szimulációs vizsgálatokban is. A TTCN-3-at az ITU-T is adaptálta mint nemzetközi ajánlást (részleteket lásd a „Tesztelés a telekommunikációban” című cikkben). Egyszóval sikerült valóban széles körben használható megoldást alkotni.

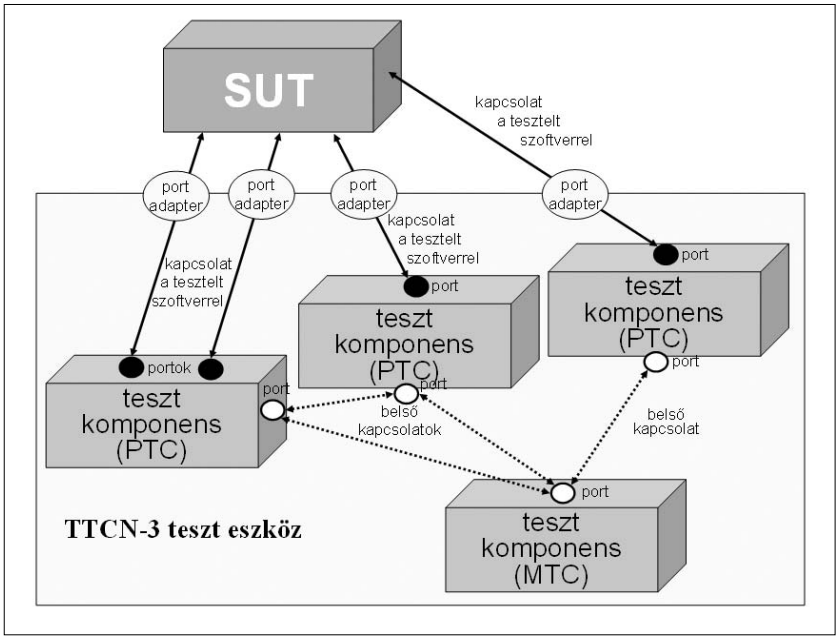
Itt kell megjegyeznem, hogy az Ericsson Magyarország a TTCN-3 teszt eszközök fejlesztésében a világon élenjáró eredményeket ért el. E műhelyből került ki a világ első működő TTCN-3-as teszteszköze, melyet TITAN névre kereszteltek. Ismereteink szerint a TTCN-3-at használó cégek között az Ericsson-ban használják legtöbben a nyelvet, és vele együtt a TITAN-t. Az Ericsson Magyarországon belül működő Teszt Kompetencia Központ feladata minden Ericsson szoftverfejlesztő egység ellátása TTCN-3 tesztkörnyezettel csakúgy, mint adaptálni a megoldást az egyes fejlesztő egységek speciális igényeihez.

A TITAN részleteivel ezen szám „TITAN, TTCN-3 tesztvégrehajtó környezet” című cikke ismerteti meg az olvasót.

Miért a TTCN-3 az általunk keresett megoldás?

Röviden: mert erre tervezték.

*5. ábra A TTCN-3 teszt rendszer általános felépítése*




Azok számára, akik ennyivel nem elégszenek meg, fejtsük ki részletesebben. Ehhez nézzük meg a TTCN rendszerek alapelveit (ez a nyelv egyes változataiban kevésbé változott) az *5. ábra* segítségével. A TTCN alapú teszt rendszer dinamikus működésének alappillérei a tesztkomponensek. Egy-egy tesztkomponens tulajdonképpen egy önállóan végrehajtott programkód a teszt végrehajtó rendszerben. Viselkedése teljesen független a többi tesztkomponensétől.

Mit csinál a tesztkomponens? Azt és csak azt, amit beleprogramoztak. Vagyis nincs előre eldöntött viselkedése, a nyelv a szokványos programozási konstrukciókon kívül (hurkok, feltételes viselkedés, ugrás stb.) a teszt-viselkedés elemeit definiálja. Ilyenek az üzenet küldése, vétele, alternatív események figyelése, időzítők kezelése, az ítélet kezelése stb. A tesztkomponensek kívánt viselkedése ezekből rakható össze.

Hány tesztkomponens lehet? Amennyit a teszteset megkíván. Minden tesztesetben van egy (és csak egy) fő tesztkomponens (Main Test Component, MTC) és lehet – elvben korlátlan számú – párhuzamos tesztkomponens (Parallel Test Component, PTC). A valós életben persze a fizikai végrehajtó környezet teljesítménye korlátozhatja a tesztkomponensek tényleges számát. A tesztkomponensekben TTCN kód fut, a külvilággal pedig portjaikon keresztül kommunikálhatnak. Akár a tesztelt rendszerrel (System Under Test, SUT), akár egymással.

Lényeges elem, hogy a TTCN absztrakt nyelv. A nyelv specifikációja nem feltételez végrehajtó környezetet, így nincsenek például különböző értéktartományú egész számok és különböző pontosságú lebegőpontos számok sem, csak egész és lebegőpontos számok vannak. De szintén nincs meghatározva a teszteszköz és az SUT közti összeköttetések módja sem, a TTCN programozónak mindössze a portokon átengedhető üzenetek típusait és irányait kell megadnia. Ezen a ponton a korábbi TTCN-2 és a TTCN-3 nyelvek eltérnek. A TTCN-2-es tesztrendszer a portok összeköttetéseit mind a tesztkomponensek között, mind a tesztkomponensek és az SUT között a háttérben, implicit építette fel, s így azokat a TTCN-2 kódból nem lehetett megváltoztatni. A TTCN-3-ban mindez a programkódból irányítható.

Nézzünk egy egyszerű példát. Tegyük fel, hogy az egyik tesztkomponensünk SIP hívásokat kezel, vagyis kívülről nézve SIP üzeneteket küld és vesz. Ha a SIP üzenetek küldése tesztkomponensek között zajlik, akkor a valós teszteszköz feladata az üzenetek továbbítása, annak módja nem ismert és eszközfüggő. De a teszt szempontjából nem is fontos, a lényeg, hogy az üzenet transzparensen átkerül egyik komponensből a másikba. Az SUT-nek küldött üzenetekkel más a helyzet, ezeket az SUT specifikációja szerinti protokoll veremben kell

szállítani, SIP esetében UDP datagrammokban vagy TCP üzenetekben. Vagyis a teszt tényleges végrehajtásánál a teszteszköznek kell a megfelelő szállító mechanizmust a SIP üzenetek „alá tennie”, amit az absztrakt TTCN-3 portok és az SUT közé beékelt port-adapterek segítségével hajt végre. A port-adapterek egy konkrét megvalósítására példa a fent említett TITAN cikkben leírt tesztport architektúra, míg egy másik példa a szabány szerinti TRI adapteres megoldás.

A nyelv más, eddig nem említett részleteire itt nem térünk ki. Erről több összefoglaló írás is megjelent, mind magyar [10], mind angol [9] nyelven.

Nézzük meg, hogyan támogatja a TTCN-3 és a TITAN az ICE alapú szoftverfejlesztési modellt, vagyis miképp teljesíti az előzőekben megfogalmazott követelményeket.

Automatizált tesztvégrehajtás elvben minden programkódon alapuló megoldással lehetséges. Elvben. De mint fent is írtuk, bonyolult rendszereknél ennek értékelhető gyakorlati haszna csak akkor van, ha megoldásunk a teszt során minden interfészt lefed és az interfészek közötti teszt koordinálás is megoldott. A TTCN-3 képes erre, vagyis gyakorlatilag bármilyen bonyolultságú rendszert lehet kizárólag TTCN-3 segítségével tesztelni. De szintúgy megoldható meglévő teszteszközök TTCN-3 környezetbe integrálása, s a TTCN-3 kódból történő vezérlése is. Mint láttuk, a komponensek közötti port kapcsolatokon keresztül a tesztkoordináció is könnyedén megoldható.

A *6. ábra* egy példát mutat be az Ericsson gyakorlatából bonyolult rendszerek teljesen automatizált tesztelésére TTCN-3-al. Ebben az esetben 3. generációs mobilhálózatok SGSN csomópontjának funkcionális vizsgálatát végezték úgy, hogy az SGSN körül minden más eszközt TTCN-3-ból szimuláltak [11].

A nyelv beépített algoritmust tartalmaz az ítéletek kezelésére. Minden tesztkomponensben egyes eseményekhez ítéletek rendelhetők, melyeket a TTCN-3 rendszer minden teszteset végén egyetlen teszteset ítéletben összegez. Így sikeres teszteknél nincs szükség a logok utólagos böngészésére és ítélethozatalra. Sikertelen teszteknél természetesen meg kell keresni az okot, ezt is segíti azonban, hogy tudjuk, melyik esemény okozta a sikertelen tesztítéletet.

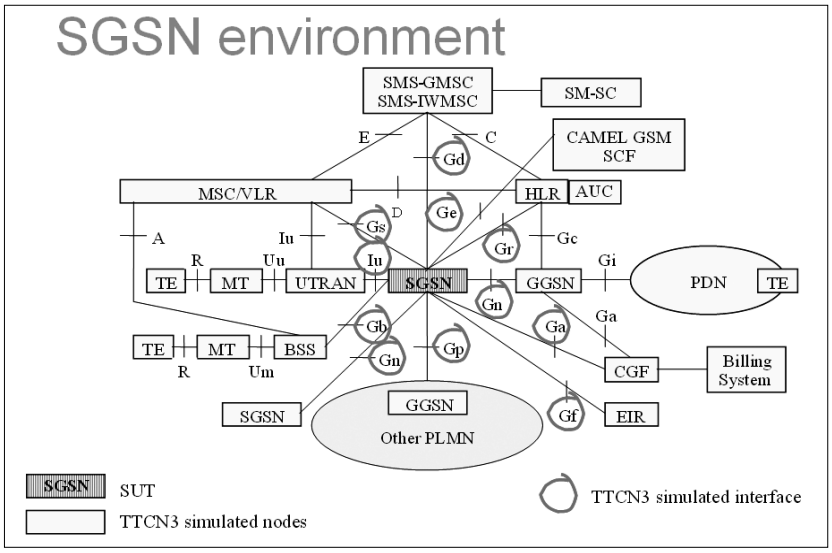
A TTCN-3-at nem egyszerűen automatizált teszt végrehajtásra, hanem felügyelet nélküli automatizált tesztelésre tervezték. Ennek egyik példája, hogy a TTCN-3 tesztkészletekben nem csak a teszteseteket, de azok végrehajtásának módját is megadhatjuk. Vagyis például egy teszteset végrehajtásakor kapott ítélettől függően választható meg, mely teszteset fusson kövekezőként, de lehetőség van feltételes, ciklikus, szelektív stb. teszteset végrehajtásra is.

Ha egy teszteset végrehajtása hibába ütközik (itt nem „fail” ítéletről van szó, melyet egyértelműen hibás SUT viselkedéshez társítunk, hanem végrehajtási hibáról), a teszteszköz köteles talpra állni. Az adott teszteset rendszerhiba (error) ítéletet kap, de a végrehajtás a következő tesztesettel folytatódik.

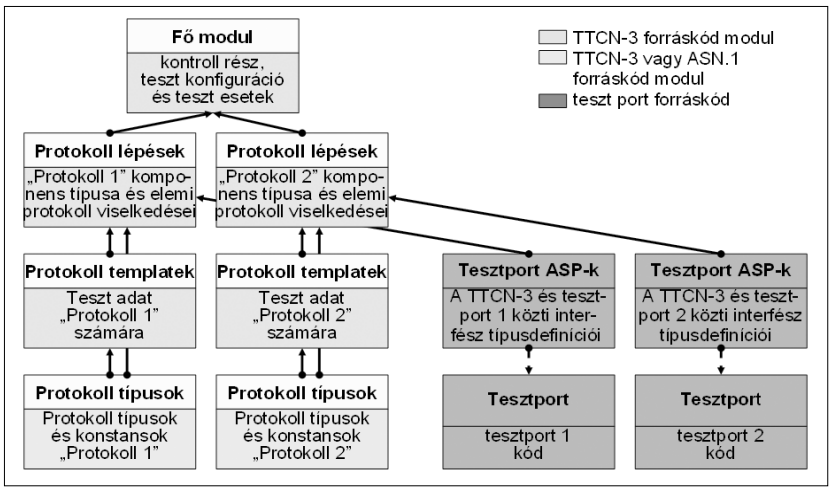
Mivel a TTCN-3 elsődleges formája ASCII szöveg, ennek tárolása, újrafordítása és futtatása valamint felhasználása más tesztkészletekben – a fájlformátum szempontjából – nem jelent problémát. A TTCN-3-ban végrehajtott funkcionális tesztek a regressziós vizsgálatok alatt gyorsan újrafuttathatók. Ezt segíti az is, hogy a TTCN (nem csak a 3-as) megengedi a kód futásidejű paraméterezését, így a TTCN-3 tesztkészlet egy időközben esetlegesen megváltozott laborkörnyezetben is újrafuttatható (IP címek, port számok stb. változása). Ehhez persze szükség van arra is, hogy a TTCN-3 kód fejlesztésekor ezt előrelátóan figyelembe vegyék.

Egy-egy adott blokkhoz, alrendszerhez vagy rendszerhez írt TTCN-3 kód

*6. ábra Példa bonyolult rendszerek tesztelésére TTCN-3-mal és TITAN-nal*




*7. ábra Tipikus TTCN-3 modul-struktúra*



újrahasználható más blokkok, alrendszerek vagy rendszerek tesztelésénél is, ahol ugyanazt a protokollt vagy funkciót kell vizsgálni.

A nyelv ezt moduláris felépítésével segíti elő. A TTCN-3 forráskód tetszőleges számú és tartalmú modulba szervezhető. Vagyis a tényleges újrahasznosításhoz ismételten csak elengedhetetlen az előrelátó tervezés. Egy TTCN-3 tesztkészlet tipikus modul struktúráját mutatja az előző oldalon a *7. ábra*.

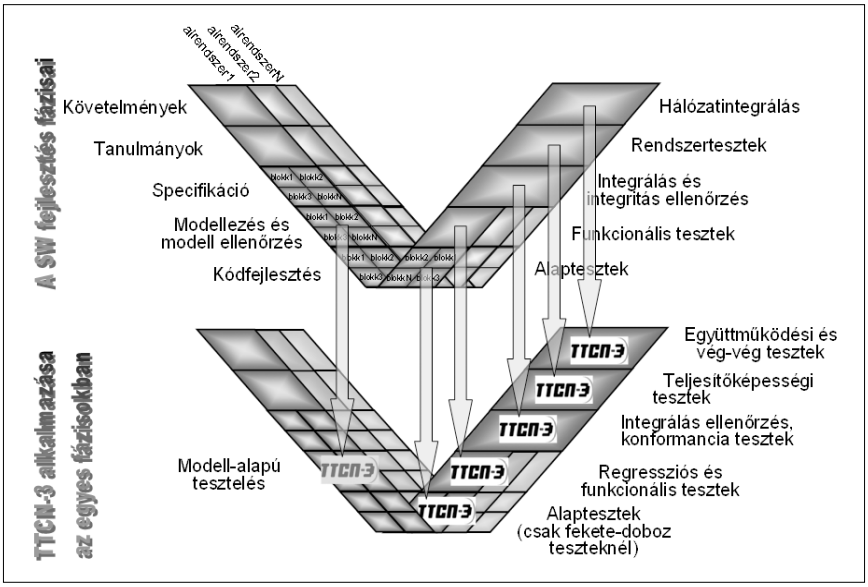
Mivel a tesztesetek rendszerenként eltérőek, tipikusan a tesztportok és protokoll modulok változtatása nélkül, a protokoll template és protokoll lépés modulok kisebb változtatásokkal újrahasználhatók.

Ehhez kapcsolódóan kell megemlíteni a TTCN-3 egy másik előnyét, mégpedig a könnyű protokoll-frissítést. Kiterjedt eszközrendszernél sokszor problémát jelent, hogy minden használt eszköz a fejlesztési projektek által megkövetelt határidőkre támogassa a legújabb protokoll verziókat. Általános bináris és szöveges protokollok esetében ehhez elég a TTCN-3 forrásmodulok módosítása, míg az ASN.1és XML protokolloknál a protokollokat leíró modulok közvetlenül a szabványból kiollózhatók (az XSD specifikációk automatikusan konvertálhatók ASN.1-be).

A fentiek ismeretében a TTCN-3 alapú tesztmegoldások széleskörű használhatóságát nem kell külön bizonygatni. Ennek egyik példájaként nézzük meg szimulált és a célhardver környezetekben történő tesztelést. Mivel a TTCN-3 kód absztrakt szinten írott, a környezetváltáshoz csak a használt tesztportokat kell lecserélni, a TTCN-3 kódban nem kell változtatni. Legalábbis akkor, ha a TTCN-3 kódot előrelátóan írták és az eltérő környezetek által megkívánt összes konfigurációs paramétert TTCN-3 futásidejű paraméterként definiálták.

Ezzel elértünk a TTCN-3 alapú megoldások egyik legnagyobb hátrányához. Ha belegondolunk, a TTCN használatakor valójában egy (tesztspecifikus) szoftverrel tesztelünk egy másik szoftvert. A TTCN-3 kód fejlesztésénél ugyanazokat a módszereket és folyamatokat kell alkalmazni, mint bármilyen más, például a tesztelt szoftver fejlesztése során. Ez egyrészt megköveteli, hogy a tesztfázisok előkészítése a korábban megszokottnál hamarabb kezdődjék, másrészt a hagyományos tesztelési megoldásokhoz szokott szakemberektől új gondolkodásmódot követel meg.

A széleskörű használhatóságnak van egy másik korlátja, nevezetesen, hogy a TTCN-3 nem igazán hatékony megoldás a fehér doboz módszer szerinti alapteszteknél (fekete doboz alapteszteknél már jól használható). Szintén nem alkalmas az interfészek alacsonyabb, első és második rétegeinek vizsgálatára. Ezt leszámítva azonban bármely rendszer bármely teszt fázisában hatékonyan alkalmazható.




*8. ábra*
*TTCN-3 elterjedtsége az Ericssonban, a fejlesztési folyamat egyes fázisaiban*

A megvizsgálandó követelmények közül a keretrendszerbe illesztés és a könnyű használhatóság maradt. Mindkettő a használt teszt eszköz tulajdonsága, nem a TTCN-3 nyelvé, így elsősorban a megfelelő teszteszköz kiválasztásakor játszik szerepet.

Az itt leírtak alapján arra következtethetünk, hogy a TTCN-3 alapú teszt rendszerek megoldást adnak a bevezetőben ismertetett kihívásokra, legalábbis a szoftverfejlesztés tesztelési vonatkozásait illetően. S valóban a *8. ábra* is ezt bizonyítja. Az ábrán a fenti V-modellt kiegészítettük egy ugyanolyan V-alakú ábrával, amely a TTCN-3 használatának elterjedtségét mutatja az Ericssonon belül. Mint látható, gyakorlatilag a fejlesztés minden fázisában használják a nyelvet és a TITAN-t (a modell-alapú tesztelés területén pilot projektekben).

## 5. Összefoglalás

A távközlési szoftverek piacán élesedő verseny, a szoftverek bonyolultságának növekedésével együtt új helyzeteket és kihívásokat teremt.

A cikkben áttekintettük a bonyolult szoftverek fejlesztésének tipikus folyamatát azért, hogy megfogalmazhassuk a fejlesztés hatékonyságát növelő tesztmegoldásokkat szembeni elvárásokat. Majd megvizsgáltuk, hogy a TTCN-3 miképp felel meg ezen elvárásoknak, s megállapíthatjuk, hogy ez a megoldás a bonyolult szoftverek fejlesztésének területén minden bizonnyal perspektivikus jövő előtt áll.

**Irodalom**


[1] http://www.v-modell.iabg.de/
[2] ETSI ES 201 873-1 v3.1.1 (2005-06)
Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
Part 1: TTCN-3 Core Language
[3] ETSI ES 201 873-2 v3.1.1 (2005-06)
Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
Part 2: TTCN-3 Tabular presentation Format (TFT)
[4] ETSI ES 201 873-3 v3.1.1 (2005-06)
Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
Part 3: TTCN-3 Graphical presentation Format (GFT)
[5] ETSI ES 201 873-4 v3.1.1 (2005-06)
Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
Part 4: TTCN-3 Operational Semantics
[6] ETSI ES 201 873-5 v3.1.1 (2005-06)
Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
Part 5: TTCN-3 Runtime Interface (TRI)
[7] ETSI ES 201 873-6 v3.1.1 (2005-06)
Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
Part 6: TTCN-3 Control Interface (TCI)
[8] ETSI ES 201 873-7 v3.1.1 (2005-06)
Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
Part 7: Using ASN.1 with TTCN-3
[9] J. Grabowski, D. Hogrefe, Gy. Réthy,
I. Schieferdecker, A. Wiles, C. Willcock:
An introduction to the Testing and
Test Control Notation (TTCN-3)
Computer Networks, Vol. 42. issue 3, 21 June 2003,
pp.375–403.
[10] Szabó János Zoltán:
A TTCN-3 tesztleíró nyelv,
Magyar Távközlés, XII. Évf. 2. szám, 2001. február
[11] Peter Eldh:
TTCN-3 in SGSN testing,
2nd TTCN-3 User Conference, Sophia Antipolis,
France, 2005. június 6-8.

# Tesztelés a telekommunikációban

JASKÓ SZILÁRD

*Pannon Egyetem, Információs Rendszerek Tanszék*
*szilard.jasko@weblab.hu*

DR. TARNAY KATALIN

*tarnay.katalin@axelero.hu*

***Kulcsszavak: tesztelés, TTCN, CSP, tesztelési szabványok***

*A modern társadalom szerves részévé vált a telekommunikáció, mindennapi életünk számos területe használja a vívmányait. Egyre népszerűbbek például az Internet és a mobil távközlés nyújtotta lehetőségek és szolgáltatások. A felhasználók többsége azonban nem tudja, hogy mindez csak úgy jöhet létre, ha a különböző gyártók betartják a szabványokat és piacra bocsátás előtt, elvégzik a megfelelő teszteket. Ennek a cikknek a segítségével egy átfogó képet kaphat a kedves olvasó a telekommunikációban használatos tesztelési eljárásokról és azok fontosságáról.*

## 1. Bevezetés

A távközlési rendszerek segítségével nap, mint nap kommunikálunk egymással és manapság ez olyannyira természetessé vált, hogy sok ember el sem tudná képzelni az életét ezek nélkül az eszközök nélkül. Gondoljunk csak bele, hogy a mobiltelefon mekkora népszerűségnek örvend! A technológia sikere nagymértékben köszönhető a mobilkészülékek egyszerű kezelhetőségének, valamint a társadalmi igénynek, ami arra irányul, hogy bármikor és bárhol, minden kötöttség nélkül, bárki tudjon kommunikálni, illetve elérhető legyen.

Azt azonban már kevesen tudják, hogy a háttérben nagyon sok és bonyolult rendszer, szervezet és emberi munka együttműködése szükséges ahhoz, hogy mindez létrejöhessen. A magyar nyelvben talán pont az ilyen szituációk szemléletes leírására szolgál „az ördög a részletekben rejlik" szólás-mondás. Most, hogy már van egy kis rálátásunk a téma összetettségére és fontosságára, azt hiszem mindenki számára világos és belátható, hogy a szabványok létrehozása, betartása és betartatása az egyetlen járható út, hogy a különböző gyártók termékei képesek legyenek az együttműködésre és így a rendszer egésze is működni tudjon.

A cikk segítségével betekintést nyerünk a tesztelés folyamatába, megismerhetjük az egyetlen szabványosított teszt eszköz a TTCN [1,2,4,9,10] fejlődését, valamint bepillanthatunk a tesztelés egy lehetséges továbbfejlődési irányába is.

## 2. Tesztelésről általában

Mindennapi életünk során gyakran találkozunk a teszteléssel és annak jelentőségével. Ezek a folyamatok mégis rejtve maradnak előttünk, mivel teljesen természetesnek vesszük azokat. A mindennapi bevásárlás során például számtalan ellenőrzési folyamat játszódik le. Nézzünk meg ezek közül párat. Ha gyümölcsöt szeretnénk venni, gondosan megvizsgáljuk, hogy egészséges és számunkra megfelelő legyen. Ezzel az eljárással tesztelje a növényt, hogy eleget tesz-e a mi elvárásainknak vagy sem. Egy másik pont a bevásárlásnál, amire valószínűleg mindenki jobban odafigyel, a pénztárnál való fizetés. Ezek a példák szemléletesen rámutatnak arra, hogy a tesztelés része mindennapi életünknek és egy átlagos ember is naponta többször találkozik olyan szituációval, ahol tudatosan vagy ösztönösen, de „tesztelnie" kell. Miután mindenki számára egy kicsit kézzelfoghatóbbá vált a tesztelés és annak célja, nézzünk egy rövid áttekintést az ipari alkalmazására.

Az ipari termelés világában, optimális esetben csak tudatos és előre tervezett tesztelésről beszélünk. Az ellenőrzési folyamatok célja itt is ugyan az, mint a korábban említett bolti példában, a hibátlan termék. Míg a gyümölcsvásárlásnál a kívánatos és egészséges vitaminforrás megvásárlása, addig a cégeknél a zökkenőmentes és a körülményekhez képest a legköltséghatékonyabb termelési folyamat segítése és megvalósítása a végcél. Fontos azonban megemlíteni, hogy az előbb említett célkitűzést csak akkor érhetjük el, ha gondosan és minden részletre kiterjedően tervezzük meg a tesztelés menetrendjét. Első lépésként azokat a pontokat kell meghatározni, ahol ellenőrizni kívánjuk a gyártási folyamatot. Vizsgáljuk meg, hogy mi történik akkor, ha rosszul választjuk meg ezeket az ellenőrzési pontokat.

Az egyik lehetőség, ha túl kevés helyen ellenőrizzük a termelési folyamatot. Ebben az esetben megnövekszik annak a valószínűsége, hogy hiba csúszik a gyártási folyamatba és ennek végeredményeként jó esély van arra, hogy növekszik a termék előállítási költsége. Érdemes megfigyelni azt a tényt, hogy a tesztelési pontok számának csökkentése rövidtávon minimalizálhatja a költségeket, azonban hosszútávon ellenkező hatást válthat ki. A másik eshetőség, amikor túlságosan biztosra szeretnénk menni és ezért túlzottan sok tesztelési pontot rakunk a gyártási folyamatba. Ennek a megvalósításnak két nagy hibája van: az egyik, hogy a túl sok ellenőrző pont indokolatlanul drágítja a termelést, a másik, hogy túlzottan lassítják a termelési folyamatot.

Ebből a két szélsőséges nézőpontból világosan látszik, hogy egy tesztelési rendszer megtervezése minden gyártási megvalósításra nézve nagy körültekintést igényelő folyamat. Miután a tesztelési pontok meghatározásra kerültek, pontosan definiálnunk kell az ellenőrzési folyamat menetrendjét. Ebbe a szakaszba tartozik minden olyan eszköznek és mérési eljárásnak a részletes megadása, amit felhasználunk a tesztelés folyamán. Ha mindez megvan, akkor már csak a tényleges ellenőrzés és az abból fakadó adatok kiértékelése van hátra.

A telekommunikáció tesztelés szempontjából speciális terület, hiszen ebben az esetben nem csak a fizikai eszközöket, hanem azok kommunikációs protokolljait is tesztelni kell. Ez merőben új szemléletet és módszereket eredményezett a tesztelés világában. Az új látásmódra azért volt szükség, mivel a protokollok speciális programok, amik kommunikációra is képesek. A másik probléma, amivel viszonylag hamar szembesültek a telekommunikáció résztvevői, hogy a rendszereiknek szükségszerűen együtt kell működniük, hiszen egyiküknek sincsen akkora piaci ereje, hogy ennek a hatalmas felvevőpiacnak minden szegmensét uralni tudja. Ezen átütő erő híján a közös cél érdekében mindenki által elismert normát fogadtak el és használnak a tesztelésre, ez pedig nem más mind egy szabvány, a TTCN.

## 3. Telekommunikációs tesztelési szabványok

Globalizálódó világunkban nagyon fontos szerephez jut a kommunikáció. Gondoljunk csak bele, hogy milyen sok szolgáltatást érhetünk el az Interneten vagy akár telefonon keresztül. A teljesség igénye nélkül nézzünk pár példát ilyen szolgáltatásokra: banki tranzakció, népszavazás (bizonyos országokban), tudakozó, vásárlás, e-önkormányzat és a sort még hosszasan lehetne folytatni. A legtöbb ember számára hamarosan ezek a dolgok természetesek lesznek és nap, mint nap igénybe fogják őket venni. Egy átlagos felhasználót azonban soha nem fog érdekelni, hogy a háttérben mennyire bonyolult rendszerek együttműködése szükséges mindehhez, pedig az igazság az, hogy rengeteg gyártó megszámlálhatatlanul sok terméke összehangolt munkájának végeredményeként tudjuk igénybe venni ezeket a szolgáltatásokat. A szabványok vitathatatlanul sokat segítettek abban, hogy mindez létrejöhessen. Ebben a fejezetben a telekommunikációs tesztelési szabványokról és azoknak a fejlődéséről olvashatunk.

**Rövidítések**

**ASN.1** – Abstract Syntax Notation One
**ATS** – Abstract Test Suite
**ETSI** – European Telecommunications Standards Institute
**ISO** – International Organization for Standardization
**IUT** – Implementation Under Test
**ITU** – International Telecommunication Union
**PCTR** – Protocol Conformance Test Report
**PDU** – Protocol Data Unit
**PICS** – Protocol Implementation Conformance Statement
**PIXIT** – Protocol Implementation eXtra Information for Testing
**TTCN** – Tree and Tabular Combined Notation (Testing and Test Control Notation)

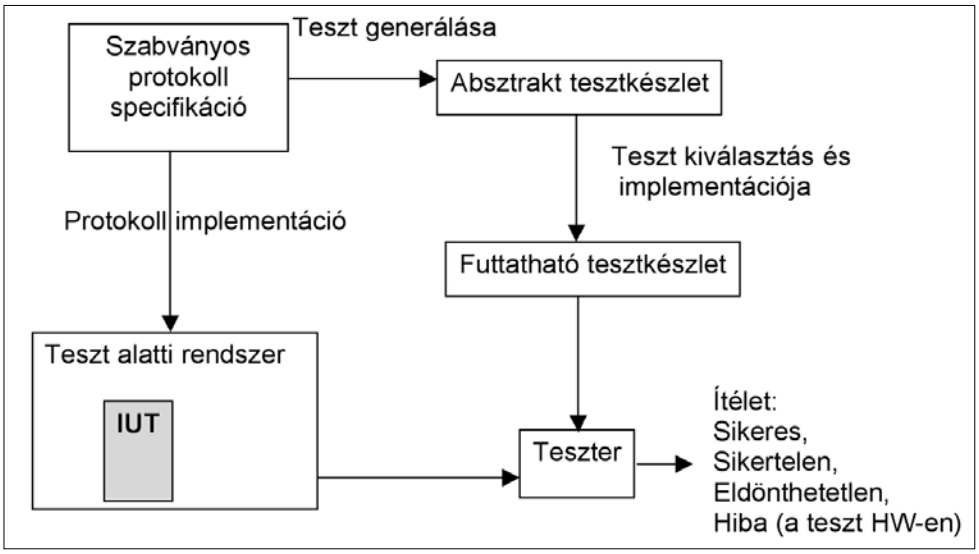
### 3.1. Tesztelési fogalmak és szabványok

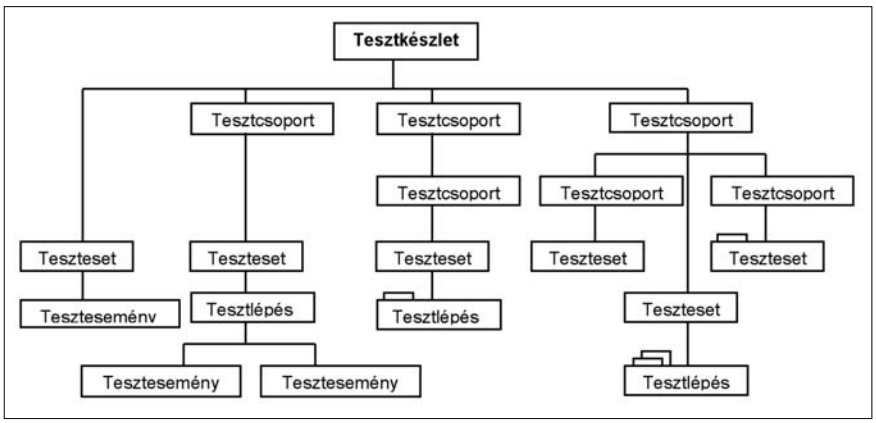
A cikkben korábban már volt szó arról, hogy miért kellett a tesztelésre külön szabványokat létrehozni, tehát a motiváció már mindenki számára ismert, most nézzünk pár konkrét dokumentumot, ami ezzel a témával foglalkozik.

Mindjárt az elején szeretném megemlíteni az egyik legfontosabbat, az ISO 9646-ost, ami a konformancia [3] tesztelés módszertanát definiálja. Ez a fajta ellenőrzési eljárás garantálja, hogy a létrehozott protokoll megfelel a szabványnak. Ennek azért van nagy jelentősége, mivel több cég terméke használ szabványos protokollokat a kommunikációhoz és ennek az eljárásnak a segítségével nagymértékben növelhető annak az esélye, hogy a termék a piacra kerülés után képes lesz kommunikálni más gyártó azonos protokollját használó eszközzel. Természetesen a szabvány pontosan definiál mindent, ami szükséges a konformancia teszt elvégzéséhez. Egy példát láthatunk erre az *1. ábrán*.

Az ábrából jól látható, hogy mindjárt a kezdeteknél rendelkezésünkre áll a szabványos protokoll specifikációja és ez alapján történik meg az implementáció. Ha a protokollt kifejlesztettük és integráljuk a futási környezetébe, akkor megkaptuk a teszt egyik résztvevőjét a teszt alatt álló rendszert (Implementation Under Test, IUT).

*1. ábra Tesztfolyam a szabványtól az ítéletig*






*2. ábra A tesztkészlet hierarchikus felépítése*

A tesztelés folyamata azonban értelemszerűen csak akkor indulhat el, ha a teszter beállításai is elkészültek. Első lépésként absztrakt teszt készletet (Abstract Test Suite, ATS) kell generálni, ami pontosan definiálja a lehetséges működési fázisokat. Protokollok esetében ez általában a kommunikációs folyam teljes bejárását jelenti. Az ATS-nek hierarchikus felépítése van, ahogy az a *2. ábrán* is jól megfigyelhető. A legkisebb egység a tesztesemény és a tesztlépés. Például egy üzenet elküldése vagy fogadása felel meg ezeknek az elemi részeknek. A hierarchia következő fokán a teszteset áll, ez ellenőrzi például, hogy a kapcsolat felépítése a szabványnak megfelelően zajlik-e. Értelemszerűen a tesztesetek tesztcsoportba rendezhetőek és a hierarchia csúcsán a tesztkészlet áll, ami tartalmazza a teljes teszt leírását.

Ha megvan az ATS, akkor implementálni kell a futtatni kívánt tesztet. Ehhez meg kell adni három dolgot a PICS-et (Protocol Implementation Conformance Statement) – ami tartalmazza az adott implementációban a protokoll tulajdonságait és paraméter értékeit –, a PIXIT-et (Protocol Implementation eXtra Information for Testing) – ami az absztrakt tesztkészlet paramétereinek konkrét értékekkel való feltöltését segíti elő – és a PCTR-t (Protocol Conformance Test Report) – amibe a tesztelés eredményét menti a teszter. Ha mindent beállítottunk, akkor lefuttathatjuk a tesztet és kiértékeljük a kapott adatokat. A teszt ítélete sikeres, sikertelen, eldönthetetlen és hiba a teszt-hardverben lehet. A sikeres értelemszerűen azt jelenti, hogy minden megfelelő. A sikertelen ítéletből arra következtethetünk, hogy valahol hibát követtünk el a szabvány implementálása során. Az eldönthetetlen esetben nincs elegendő információ a teszter birtokában, hogy eldönthesse, ami történt, az a hibás működés következménye vagy csak egyszerűen nem tervezték bele a tesztkészletbe. Később az ITU-T az X.290–X.296-os szabványcsaládjába átvette az ISO 9646-ot.

Egy másik nagyon fontos szabvány a Z.140-es. Ez a dokumentum definiálja a TTCN-3 mag nyelvét. A TTCN-3 egy jelölésrendszer, aminek a segítségével a legmodernebb tesztelési módszereket és struktúrákat használhatjuk. A TTCN-3 jövőbemutató felépítéséről tanúskodik, az a tény, hogy a magnyelv közvetetten programozható grafikus, vagy táblázatos programozási felület segítségével is. A táblázatos felületet a Z.141-es míg a grafikai programozói interfészt a Z.142-es ITU-T szabvány definiálja. Természetesen a TTCN-3-ról olvashatunk az ETSI által jegyzett ETSI ES 201 873 szabványban is. E rövid áttekintésből is látszik, hogy vannak átfedések a különböző szervezetek anyagai között, de ezeknek a dokumentumoknak a végső célja a jobb és hatékonyabb teszt módszerek és jelölőrendszerek definiálása lesz.

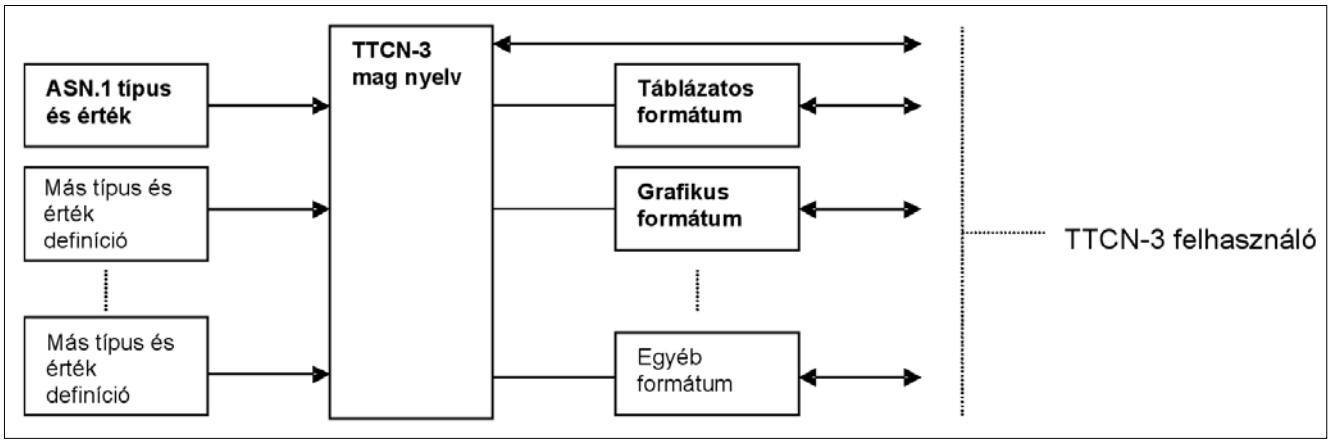
### 3.2. TTCN története

A szabványosító szervezetek és a telekommunikációban érdekelt cégek közös erőfeszítéseinek köszönhetően az 1980-as évek közepétől egy formális teszt nyelvet kezdtek el kidolgozni, hogy a piacra kerülő különböző eszközök gördülékenyen tudjanak együttműködni. A kommunikáló eszközök és az azokat kiszolgáló berendezések számának drasztikus növekedése tette szükségessé mindezt.

Az első tényleges szabvány 1995-ben látta meg a napvilágot és TTCN-nek (Tree and Tabular Combined Notation) hívták. A TTCN volt hivatva arra, hogy bárki le tudja ellenőrizni, hogy a terméke együtt tud-e működni más szabványos eszközzel. A szabvány első verziója elsősorban protokollok tesztelésére volt kifejlesztve és ezért nehézkesen lehetett alkalmazni egyéb eszközök ellenőrzésére. Talán ez az egyik oka, hogy az első verziót szinte csak a telekommunikáció berkein belül használták.

A TTCN-2 tulajdonképpen a szabvány első verziójának a kiterjesztése, amit 1997-ben adtak ki. A TTCN-2 segítségével hatékonyabban lehet konkurens rendszerek tesztjét elvégezni. A moduláris felépítés további előnyökkel vértezte fel ezt a tesztrendszert. Ilyen például, hogy a tesztleírások újrafelhasználhatóvá váltak a különböző futó munkák között. Egy másik nagy újítás, hogy ezek után többfelhasználós teszt készletek kifejlesztésére is lehetőség nyílt. Ebből a két újításból is látszik, hogy a TTCN-2 a maga korában egy nagyon előremutató eszköz volt.

2001-ben publikálták a szabvány harmadik verzióját. Fontos megemlíteni, hogy a TTCN-3 olyan sok újítást tartalmazott, hogy még a nevét is megváltoztatták, ami „Test and Test Control Notation” módosult. Ez a teszteszköz alkalmas 3. generációs protokollok tesztelésére is. A protokolloknak ez az új generációja lesz többek között hivatva a különböző hangi és (multimédiás) adatok hatékony forgalmazására. A TTCN legutolsó publikált változata méltán nevezhető modern és előremutató teszteszköznek. Nézzük meg, hogy miért is szolgált rá ez a rendszer ezekre a pozitív jelzőkre. Míg a korábbi verzióknál a táblázatos programozási felület volt az egyeduralkodó, a TTCN-3 esetében egy objektum orientált programozási nyelvhez hasonló struktúra,




*3. ábra A TTCN-3 strukturális felépítése*

a magnyelv kapta a főszerepet. Ehhez a központi egységhez különböző modulokat kapcsolhatunk, így egy rugalmas és sokrétű eszközt kapunk végeredményül. A modulok interfészek segítségével kapcsolódnak a magnyelvhez és funkció szerint két nagy csoportra oszthatóak.

Az egyik modulcsoport definiálja a magnyelv számára, hogy mit is értünk pontosan egyes változókon és azok típusán. Első hallásra egyértelműnek tűnik, hogy mit értünk például egész (integer) típusú változó alatt, de ha jobban belegondolunk a kívánt egész szám nagysága határozza meg, hogy hány bájton kell ábrázolni a kívánt számot. A teszteszközök esetében egy ilyen apróság miatt ugyan olyan körülmények mellett más eredményeket kaphatunk és ez a kiértékelés folyamán téves következtetésekhez vezethet. A TTCN-3 esetében az ASN.1 [5-8] jelölésrendszer a leginkább használt, de természetesen bármilyen erre a célra alkalmas eszközt hozzá tudunk kapcsolni a megfelelő interfész segítségével a magnyelvhez.

A modulok másik csoportja segítségével közvetett úton programozni tudjuk a magnyelvet. A gyakorlatban ez azt jelenti, hogy mi leprogramozzuk a tesztet, például egy grafikus vagy táblázatos programozási felület segítségével és ezt egy fordítón keresztül a magnyelv számára értelmezhető formára hozzuk. A TTCN-3 strukturális felépítését figyelhetjük meg a *3. ábrán*.

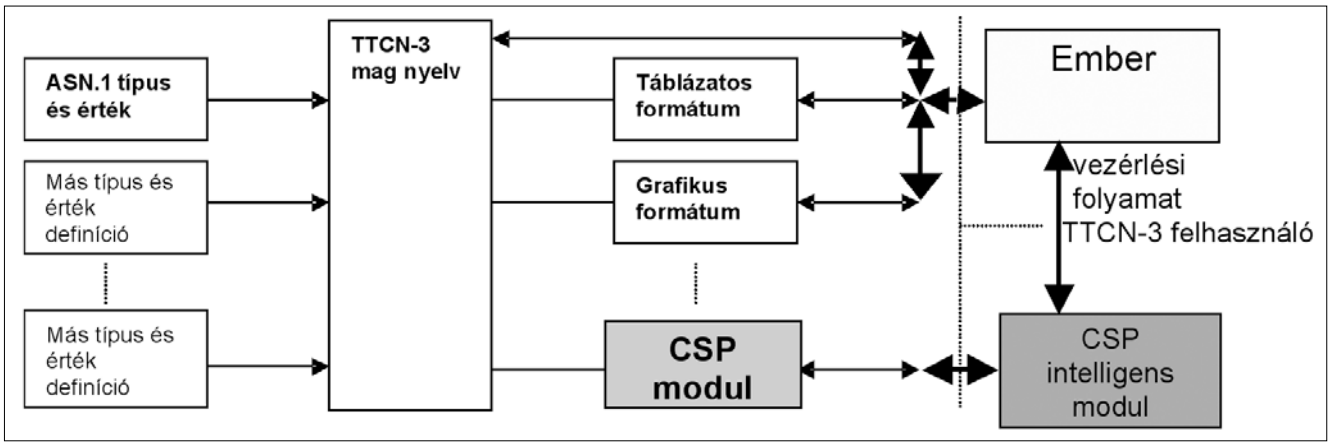
## 4. A tesztelés jövője

Egy tény biztosan kijelenthető a teszteléssel kapcsolatban, ez pedig az, hogy mindig szükség lesz rá. A filozófiáját tekintve azonban bekövetkezhetnek változások. A mai eljárásoknak a legnagyobb hibája, hogy platform és szoftver megvalósítás függő alkalmazások, így egy újabb tesztelési eljárás bevezetésénél a tesztleírásokat újra meg kell írni. Ez történt a TTCN-2 és TTCN-3 közötti váltásnál is.

Egy ígéretes irányvonal lehet a matematikai modellt és leírást használó tesztelés. Ennek a megoldásnak a segítségével self-adaptív tesztelés jöhetne létre, ami azt jelenti, hogy a teszter felismeri a tesztelendő protokollt, annak függvényében legenerálja az ATS-t. Az absztrakt tesztkészletből egy fordító segítségével átfordítja a megfelelő platform és szoftverkörnyezetre a tesztet. Ennek az eljárásnak a tesztelés automatizálásának lehetőségén túl a platformfüggetlenség is előnye. Jelenleg ígéretes kutatás folyik ezen a területen. A használt matematikai modell a CSP [10] (Communicating Sequential Processes) és az eddigi eredmények tükrében kijelenthető, hogy ez az elképzelés használható a tesztelés területén.

A *4. ábrán* jól látható az új tesztrendszer és a TTCN-3 kapcsolata. Ebben az esetben a CSP intelligens modul végzi a tesztelendő protokoll felismerését és a teszt

*4. ábra A TTCN-3 strukturális felépítése a CSP modulokkal kiegészítve*



elsődleges generálását. A CSP modul hasonló funkciót lát el mint a táblázatos vagy grafikai programozási felület, azzal a különbséggel, hogy ennek a kódját a CSP intelligens modul generálja. Végül a CSP modulban létrehozott leírást egy fordító átalakítja a TTCN-3 magnyelvére.

## 5. Összefoglalás

Úgy gondolom, hogy a 21. század egyik legmeghatározóbb irányvonala a telekommunikáció és az erre épülő iparágak lesznek. Erre utaló jelenség többek között az Internet és a mobil kommunikáció töretlen népszerűsége. Ez a cikk ennek a nagy és bonyolult világnak egy kis szegmensével, a teszteléssel foglalkozott. Bemutatásra került, hogy milyen fontos szerepe van a szabványoknak és a megfelelő ellenőrzésre szolgáló eszközöknek. Bepillantást nyerhettünk az egyetlen szabványos teszteszköz, a TTCN fejlődésébe, valamint, hogy milyen lehetséges továbbfejlesztési iránya lehet a tesztelésnek.

Mindezen tudás birtokában magabiztosabban mozoghatunk a telekommunikáció világában és esetlegesen elősegíthetjük újabb és még precízebb eszközök kifejlesztését, hogy ezzel is hozzájáruljunk az emberiség fejlődéséhez.

**Irodalom**


[1] ITU-T Recommendation Z.141 (2001),
The Tree and Tabular Combined Notation version 3 (TTCN-3): Tabular presentation format.

[2] ITU-T Recommendation Z.142 (Draft),
The Tree and Tabular Combined Notation version 3 (TTCN-3): Graphical format.

[3] ITU-T Recommendation X.290 (1995),
OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – General concepts.
ISO/IEC 9646-1:1994, Information technology – Open Systems Interconnection – Conformance testing methodology and framework
– Part 1: General concepts.

[4] ITU-T Recommendation X.292 (1998),
OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – The Tree and Tabular Combined Notation (TTCN).
ISO/IEC 9646-3:1998,
Information technology –
Open Systems Interconnection –
Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN)

[5] ITU-T Recommendation X.680 (1997) |
ISO/IEC 8824-1:1998,
Information technology –
Abstract Syntax Notation One (ASN.1):
Specification of basic notation.

[6] ITU-T Recommendation X.681 (1997) |
ISO/IEC 8824-2:1998,
Information technology –
Abstract Syntax Notation One (ASN.1):
Information object specification.

[7] ITU-T Recommendation X.682 (1997) |
ISO/IEC 8824-3:1998,
Information technology –
Abstract Syntax Notation One (ASN.1):
Constraint specification.

[8] ITU-T Recommendation X.683 (1997) |
ISO/IEC 8824-4:1998,
Information technology –
Abstract Syntax Notation One (ASN.1):
Parameterization of ASN.1 specifications.

[9] ETSI ES 201 873-2 (V2.2.1),
Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
Part 2: TTCN-3 Tabular Presentation Format (TFT).

[10] ETSI TR 101 873-3 (V1.1.2),
Methods for Testing and Specification (MTS);
The Tree and Tabular Combined Notation version 3;
Part 3: TTCN-3 Graphical Presentation Format (GFT).

[11] Roscoe, A.W.,
The Theory and Practice of Concurrency,
Prentice Hall Intern. Series in Computer Science,
ISBN 0-13-674409-5, 1997.

# Távközlési szoftverek tesztelése

DIBUZ SAROLTA

*Ericsson Magyarország Kft., K+F Igazgatóság*
*sarolta.dibuz@ericsson.com*

***Kulcsszavak: szoftverfejlesztés életciklusa, funkcionális tesztelés, teljesítmény-tesztelés, tesztelés-automatizálás***

*Ebben a cikkben röviden összefoglaljuk a szoftverek, különösképpen a távközlési szoftverek tesztelésének, minőségbiztosításának folyamatát. Foglalkozunk a tesztelés szervezési részével, valamint a tesztelés automatizálásával.*

## 1. Bevezetés

A szoftverek (nemcsak a távközlési szoftverek) komoly tesztelési folyamaton kell átessenek, mielőtt a felhasználóhoz jutnak. Tapasztalati tény, hogy a távközlési szoftverek fejlesztési költségeinek több mint 50%-át a tesztelés költségei teszik ki.

Egy hiba kijavítása annál olcsóbb, minél hamarabb találják meg a hibát a tesztelési folyamatban. Általános tapasztalat az, hogy ha egy tesztelő szakember talál meg egy hibát, akkor tízszer annyi idő illetve költség azt kijavítani, mintha maga a fejlesztő találná meg. Ilyenkor már többen vannak a folyamatban: a tesztelést végző szakember és a fejlesztő, esetleg nem is az, aki a kódot írta, hanem valaki más. A kód írása már régebben történt, és már nem emlékszik pontosan a fejlesztő, hogy hogyan is működik az adott kódrészlet, mindez drágítja a hibajavítást. Ehhez képest is tízszeres a költsége azon hibák kijavításának, amelyeket már egy eladott szoftver termékben a felhasználó talál meg. Ekkorra még több idő telik el a fejlesztés óta, és a teljes szoftverrendszerben, ami több elemből állhat, nagyon nehéz megtalálni azt, hogy melyik elem melyik modulja okozta a hibát. Ekkor már nem azok foglalkoznak a szoftverrel, akik írták, hanem az üzembe helyezők és a szoftverkarbantartást végzők. Kódolási hibák javítását persze kódolók végzik, de a legtöbb esetben nem az, aki eredetileg is írta a kódot, hiszen lehet, hogy már más feladatot kapott azóta.

A következőkben azt mutatjuk be, hogy a távközlési szoftverek tesztelésekor milyen feladatokkal állunk szemben. A második fejezet arról szól, hogy hogyan kell előkészíteni a tesztelést a szoftver fejlesztő projektekben. A harmadik fejezet a tesztelés helyét és szerepét mutatja be a szoftverfejlesztés folyamatában. Végül a teszt automatizálásról szólunk, amely nagy mértékben javítja a tesztelés hatékonyságát a projektekben.

## 2. Amit a tesztelésről tudni kell

Teszteléssel csak a hibákat tudjuk megtalálni, jobb nem lesz tőle a szoftverünk. Ezért nagyon fontos, hogy gondos rendszertervezés előzze meg a fejlesztést, aminek része a tesztelés gondos megtervezése is. A szoftverek fejlesztése során a tesztelés gyakorlatilag a rendszer tervezésével párhuzamosan elkezdődik, hiszen a tesztelést is meg kell tervezni, s még ha automatikus teszteket is készítenek, még több idő szükséges a tesztek előállításához. Ezért aztán időben hozzá kell látni hogy a tesztek rendelkezésre álljanak amikor végre kell hajtani azokat.

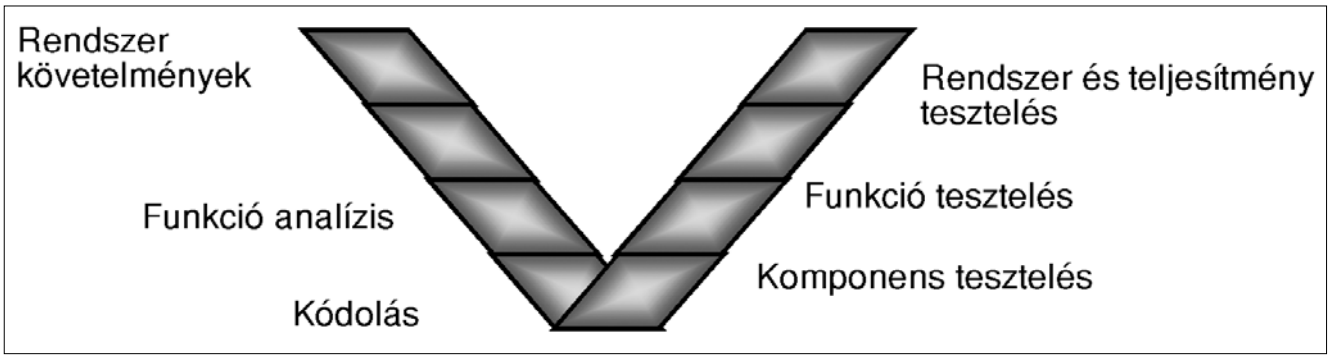
A tesztek tervezése során meghatározzák, hogy milyen eszközökkel, és milyen módon fognak tesztelni. Ezenkívül összegyűjtik a teszteseteket, tehát meghatározzák, hogy milyen tesztfeladatokat fognak elvégezni. A tesztek végrehajtásának első lépése a tesztkörnyezet felállítása. Ezután következik a tesztfeladatok végrehajtása. Ezek eredményeit kiértékelik, ehhez fontos a tesztek lezajlását mutató logok analizálása.

Megkülönböztetünk statikus és dinamikus tesztelést. Statikus tesztelés valójában a kód vagy kódmódosítás alapos átnézését, ellenőrzését jelenti. Ezt mindig más kódoló végzi, mint aki a kódot írta. Dinamikus tesztelés során már működésbe hozzák a szoftvert, amit tesztelnek. Gerjesztik a bemenetén és figyelik a válaszokat a kimeneten.

Beszélhetünk fehér doboz és fekete doboz tesztelésről is. Fehér doboz tesztelés során kihasználjuk azt, hogy látjuk a kódot, ismerjük annak felépítését, struktúráját. Fekete doboz tesztelés esetén csak a tesztelt szoftver külső viselkedése ismert. Ekkor szükséges a dinamikus tesztelés, mely során az interfészein keresztül gerjesztik a szoftvert és ellenőrzik, hogy a gerjesztés alapján a várt, helyes választ kapják-e.

## 3. A tesztelés helye a szoftverfejlesztésben

A távközlésben használt szoftverek nagyon nagy méretűek. Fejlesztésük modulonként történik, később a modulokat összeépítik, és így áll elő az egyre nagyobb méretű kód. A tesztelést is érdemes fázisokra bontani, a szoftverfejlesztési folyamatának megfelelően. A különböző teszt fázisokat mutatja az *1.ábra*.




*1. ábra A szoftverfejlesztés és tesztelés kapcsolata*

Már annak is ellenőriznie kell a kód helyességét, aki a kódot írta. Ha fejlesztő is talál hibákat, akkor nyilván saját maga javítja is ki a legkönnyebben.

Az első tesztfázis a fejlesztés során az elkészült modulok tesztelése, ezt nevezik modul- vagy komponenstesztnek. Ennek során a kódoló olyan teszteket végez el, amellyel ellenőrzi az általa írt kódot függetlenül a rendszer többi részétől. Ezek a tesztek elsősorban fehér doboz tesztek.

Amikor a modulokat összerakják, integrálják, további tesztek következnek. Ilyenkor már a kódolóktól független teszterek ellenőrizik, hogy a kód, illetve a több modulból összeállított rendszer megfelelően működike. Eközben valamennyi új funkcionalitást ellenőriznek tipikusan fekete doboz megközelítéssel. Általában nemcsak helyes adatokkal gerjesztik a rendszert, hanem azt is megnézik, hogy hibás bemenetekre vagy helyzetekre hogyan reagál, hogyan tűri ezeket.

Ha már leellenőriztük, hogy a rendszer tudja azokat a funkciókat, amiket a fejlesztés során meg kívántunk valósítani, még azt is meg kell nézni, hogy a rendszer hogyan fog viselkedni várhatóan a valóságos környezetében: bírja-e majd azokat a forgalmi terhelési viszonyokat, ami elvárható, milyen teljesítménnyel fogja végrehajtani a funkciókat.

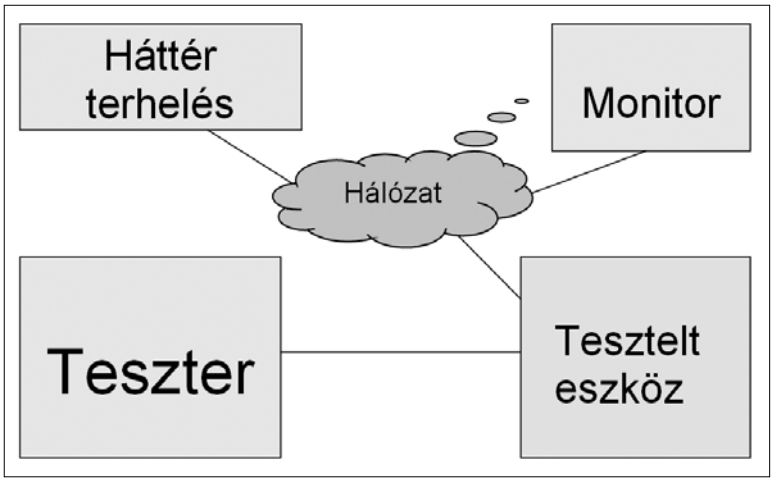
Ez a teljesítmény tesztelés feladata *(2. ábra)*, ahol a tesztelést végző eszközön kívül a tesztkonfiguráció gyakran tartalmaz háttérforgalmat generáló eszközt is, ezzel biztosítva azt, hogy különböző forgalmi helyzetekre el tudjuk végezni a mérést.

Távközlési szoftverek esetében mindig fontos a szabványos interfészek ellenőrzése és az, hogy együtt tude működni a rendszerünk más gyártók hasonló célra fejlesztett szoftverével. Ezt a konformancia és az együttműködési tesztek során érik el. Erről e szám egy másik cikkében olvashatunk részletesen.

A termékek új verzióit úgy alakítják ki, hogy módosítják, kiegészítik újabb modulokkal, elemekkel, amik az új funkcionalitást valósítják meg a szoftverben. Ilyenkor nem elég az új működést tesztelni, hogy az jól került-e megvalósításra, hanem azt is ellenőrizni kell, hogy nem rontottunk-e el valamit a rendszer régi működéséből. Ezt nevezik regresszió-tesztelésnek.

A regresszió-tesztelés tulajdonképpen minden olyan esetben nagyon hasznos, ha már letesztelt, ellenőrzött szoftverhez újabb szoftver részeket adunk, akár a termék új verziójának készítése, akár inkrementális módon történő szoftverfejlesztés során. Az inkrementális fejlesztés azt jelenti, hogy arra törekszünk a szoftverfejlesztés során, hogy a sok új és módosított modulból nem egyszerre hozzuk létre az új szoftvert, hanem egyszerre csak kisebb új részeket adunk a szoftverhez, és ezután ellenőrizzük, hogy az így keletkezett szoftver jól működik-e. Csak ezután következhet a következő modul integrálása. Ezzel érjük el, hogy nem egyszerre kell a nagy mennyiségű új szoftverben megtalálnunk, hogy hol a hiba a tesztelés során, hanem mindig abban a részben kell csak a hibát keresni, amit újonnan tettünk a kódba, hiszen előtte már jól működött. Ennek a logikus megközelítésnek az a hátránya, hogy gyakran kell lefuttatnunk szinte ugyanazokat a teszteket, azaz regresszió-tesztelni. A regresszió-teszt hatékonysága érdekében érdemes ezeket a teszteket automatizálni.

*2. ábra Tesztkonfiguráció teljesítmény tesztelésre*




## 4. Teszt-automatizálás

A tesztelés automatizálása azt jelenti, hogy automatikusan hajtjuk végre a tesztelést, és automatikusan értékelődik ki a tesztelés helyessége. Ehhez egy tesztelést végrehajtó eszközre, szoftverre van szükség, valamint arra, hogy meghatározzuk, előre definiáljuk, hogy hogyan tesztelünk, azaz elkészítsük a

tesztsorozatot. Így pontosan átgondolt, alapos teszteket lehet létrehozni, amiket emberi beavatkozás nélkül sokszor, gyorsan végre lehet hajtani. Ezek a tesztek akár éjjel is futhatnak, és a kiértékelésük is igen hatékony, általában szintén automatikus. Ezzel nemcsak hogy sok emberi munkát spórolunk meg, hanem ismétlődő, unalmas emberi munkát, hiszen ugyanazokat a teszteket többször kellene végrehajtani. A tesztek automatizálásához alapvetően új gondolkodási módra van szükség. Hiszen ehhez nemcsak a tesztelt rendszer működését kell megérteni, hanem a tesztelő rendszer működését is. A tesztsorozatok írásához pedig programozói ismeretekre is szükség van, ami nem feltétlenül szükséges a hagyományos teszteléshez.

A teszt-automatizálás teljesen új szemléletet hoz be a tesztelésébe. A tesztelés előkészítése során tovább tart az automatikus tesztek elkészítése, hiszen az automatikusan végrehajtható tesztprogramokat el kell készíteni.Viszont a teszt végrehajtása nagyon gyors. Ennek megfelelően kell tervezni a teszt-projektet. Azt is figyelembe kell venni, hogy bár a tesztek kiértékelése is automatikusan történik, mégis nagyon fontos, hogy a tesztelést felügyelő teszter értse a tesztrendszer és a tesztsorozat működését is, nem csak a tesztelt szoftverét. A tesztprogramok megírásához a teszteréknek is inkább programozói feladatokat kell ellátniuk. Azonban az egyre bonyolultabbá váló távközlési szoftverek tesztelésében már nem képzelhető el a megfelelő tesztek végrehajtása automatizálás nélkül.

## 5. Összefoglalás

A cikk röviden ismerteti a szoftvertesztelés fontosságát, szerepét és helyét a szoftver fejlesztésben, és hogy miért fontos a tesztelés automatizálását bevezetni a szoftverfejlesztő projektekben. Ez a tesztelésben alapvető szemléletváltással járó lépés, a tesztelés automatizálása elkerülhetetlen.

Ezt ma már nem kérdőjelezik meg egyetlen szoftverfejlesztő projektben sem. Azt azonban, hogy mely teszteket érdemes illetve lehet automatizálni, és milyen módon, mindig az adott fejlesztési környezet és a szoftverrel szemben támasztott követelmények döntik el. A projektek tervezésében figyelembe veszik a teszt automatizálásának igényeit, és azt is, amilyen lehetőségeket biztosít az automatikus tesztelés a projektben. A szoftvertesztelés módszere folyamatos átalakuláson megy át az automatizálás bevezetése miatt. További kérdéseket is felvet a módszer, például, hogy hogyan ellenőrizhető az automatikus teszteléshez fejlesztett tesztszoftver helyes működése.

Ha a távközlésben használt szoftvereink egyre bonyolultabbá válnak, akkor az is elmondható, hogy az ezeket automatikusan tesztelő szoftverek még bonyolultabbak lesznek. Így egy további kérdés, hogy az automatizált tesztelésben használt szoftvereinket hogyan tudjuk minél jobban felhasználni újabb projektekben.

## Hírek

**Magyar fejlesztésű szoftvert alkalmaz a Cisco a hálózati beléptetési rendszerében**

A Cisco Systems az EagleEyeOS™ magyar fejlesztésű, adatlopás elleni EagleEyeOS Professional biztonsági szoftverét alkalmazza Network Admission Control (NAC) hálózatbiztonsági programjában.

A Cisco NAC programja a kliensoldali biztonsági megoldások hálózati szintű ellenőrzését biztosítja, és elsősorban a hálózati férgek és vírusfenyegetések által okozott károk eshetőségét korlátozza. A NAC automatikusan érvényesíti a vállalati biztonsági házirendet az összes NAC kompatíbilis eszközön, és csak az informatikai biztonságért felelős rendszergazdák által beállított biztonsági házirendeknek megfelelő – például a legfrissebb telepített vírusvédelmi szoftvert futtató – kliensgépek, illetve végberendezések számára engedélyezi a hozzáférést.

Az EagleEyeOS Professional szoftverének alkalmazása révén a védelem újabb dimenziója, a belső adatlopás elhárítása valósul meg a NAC program használói számára. A szoftverrel többek között nyomon követhető és szabályozható a dokumentumok vándorlása, módosítása, sőt maguknak a külső eszközöknek a mozgása is.

A két szoftver együttes hatékonyságának legszemléletesebb példája a NAC rendszer védelmi, és a EagleEyeOS Zone funkciójának együttműködése. Amennyiben a NAC-ot és EagleEyeOS-t használó hálózathoz olyan eszköz csatlakozik, amelyen egyik program sem fut, akkor alapesetben a NAC kizárja azt. Ezzel szemben ha a NAC-ot és EagleEyeOS-t egyaránt használó gépet más hálózatra csatlakoztatják, a gép védelmét a továbbiakban az EagleEyeOS látja el, azaz lezárja a gép védett zónáját az idegen hálózat egyéb elemei elől. A NAC alapszabályainak megfelelően az integrált EagleEyeOS esetében is jelentős szerepet kap majd a céges biztonsági házirend: ha olyan gép csatlakozik a hálózathoz, amelyre telepítettek EagleEyeOS-t, de eltérő szabályrendszer fut rajta, akkor azt a NAC felismeri és kizárja a hálózatból.

# A konformancia- és együttműködés-tesztelés bemutatása

KRÉMER PÉTER

*Ericsson Magyarország Kft., Test Competence Center*
*Peter.Kremer@ericsson.com*

***Kulcsszavak: együttműködés-tesztelés, konformancia-tesztelés, ROHC***

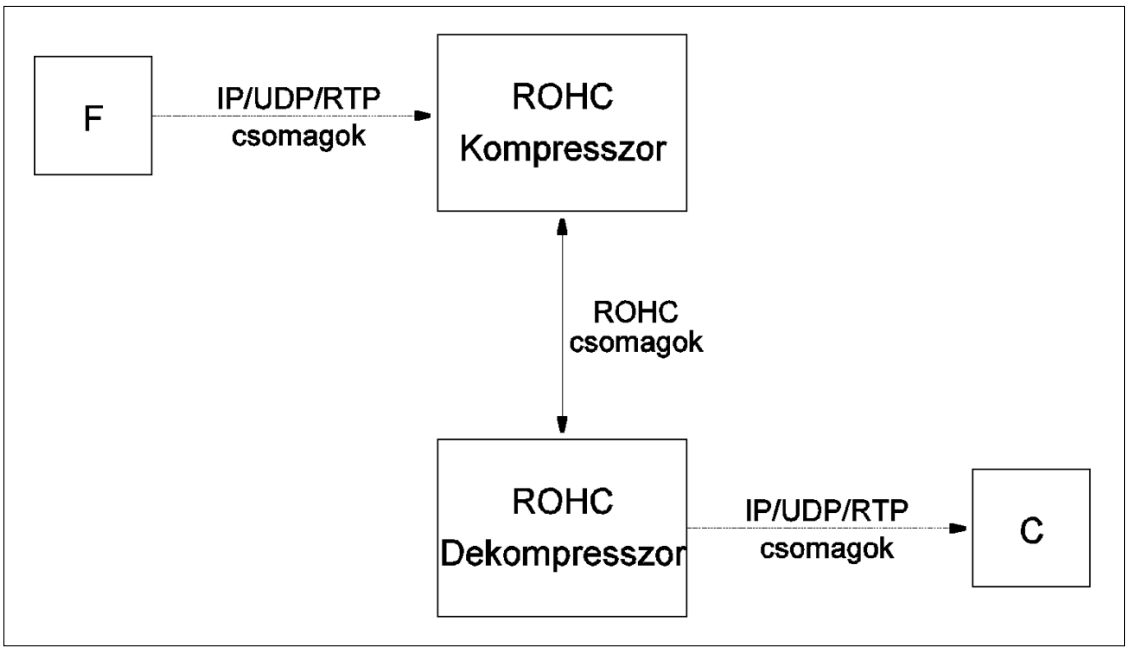
*A cikk elsődleges célja bemutatni az együttműködés-tesztelés folyamatát és fajtáit. Emellett azonban kitér a konformancia-tesztelésre is. Egyrészt azért, mert azon keresztül mutatja be az együttműködés-tesztelést. Másrészt pedig azért, mert manapság mindkét tesztelési mód fontossá vált és nem lehet csak az egyik vagy csak a másik módszer alapján megalapozott véleményt formálni egy-egy berendezésről, protokoll megvalósításról. A könnyebb érthetőség érdekében néhány konkrét példát is bemutatok a ROHC protokoll tesztelésére.*

## 1. Bevezetés

A két tesztelési módszer közül a konformancia teszt a régebbi, ezt a fajta tesztelést távközlési berendezések gyártói kezdték el használni annak bizonyítására, hogy a termékük megfelelően működik. Ez a módszer a mai napig nemcsak használatban van, de igen népszerű is. A bonyolult és drága berendezések miatt a távközlés világában nagyon fontos, hogy minden pontosan meghatározott módon történjen. Ez a megközelítés nem csak a távközlési szabványokban tükröződik (alapos, precíz leírás; jól definiált interfészek), hanem a berendezések teszteléséhez használt módszerekben is. Ezért használják az aprólékos, precíz konformancia-tesztelést.

Az együttműködés-tesztelés az IP protokollok fejlődésével kapott és kap jelenleg is egyre nagyobb szerepet. Ez a tesztelési módszer ugyanis az IP protokollok esetén a leggyakrabban használt eszköz a különböző implementációk ellenőrzésére. Az IP alapú protokollok egyszerű, olcsóbb eszközökön működnek, az ilyen berendezéseknek sokkal több gyártója és vevője van. Ebből következően a berendezések tesztelésénél is teljesen más a cél, így nyilvánvaló, hogy teljesen másfajta módszert kell használni a tesztelésnél is. Az IP-s világban elterjedt szabványokban általában nincsenek olyan jól definiált interfészek, mint a távközlési szabványokban. Ezen kívül gyakran előfordulnak olyan esetek, ahol az implementációra bíznak egyes döntéseket vagy egyszerűen csak nem definiálják az elvárt működést. Ehhez a megközelítéshez pedig az együttműködés-tesztelés áll közelebb.

A következő fejezetben bemutatjuk a ROHC protokollt, amelyet a továbbiakban példaként fogok használni. Ezután röviden bemutatom a konformancia-tesztelést, illetve a ROHC protokoll teszteléséhez használt teszt-elrendezéseket. A cikk további részeiben pedig az együttműködés-tesztelés részletes ismertetése következik.




*1. ábra*
*Tipikus ROHC elrendezés*

## 2. A ROHC protokoll rövid ismertetése

A ROHC (RObust Header Compression) protokoll IP csomagok fejlécének tömörítésére szolgál, pont-pont jellegű összeköttetések esetén. Amikor IP protokollt használunk vezeték nélküli hálózatokban, akkor a legnagyobb probléma általában az IP fejléc nagy mérete a hasznos adathoz képest. Beszédátvitel esetén a hasznos rész 15-20 bájt körül van, míg a fejléc (IPv4, UDP és RTP protokollok használata esetében) 40 bájt hosszú. A szűkös sávszélesség minél hatékonyabb kihasználásához tehát valamilyen tömörítésre van szükség.

A ROHC tehát egy olyan IP fejléctömörítő megoldás, amit kifejezetten vezetéknélküli hálózatokhoz fejlesztettek ki. Más tömörítő eljárásoktól eltérően nem az egy IP csomagon belüli redundanciát használja ki, hanem az egymást követő – de ugyanahhoz az adatfolyamhoz tartozó – csomagok közötti hasonlóságot. Egy adatfolyam esetében az IP cím például nem változik, így ezt az információt elegendő egyszer átküldeni a csatornán. Más esetekben egy mező értéke – a többi mező értékének ismeretében – kiszámolható (például a csomag hossza), így ezeket sem kell minden esetben továbbítani.

Látható tehát, hogy az IP csomagok tömörítéséhez és visszafejtéséhez nemcsak az adott IP csomag tartalmának ismerete szükséges, hanem az azt megelőző csomagoké is. Ez különféle adatbázisok és táblázatok létrehozását és folyamatos frissítését igényli mindkét oldalon. A ROHC hatékonyságára jellemző, hogy a 40 bájtos fejléc helyett mindössze 1 bájtban képes az eredeti csomag visszaállításához szükséges minden információt eltárolni – és ebben már az átviteli hibák elleni CRC védelem, valamint a ROHC csomag típusának megjelölése is benne van!

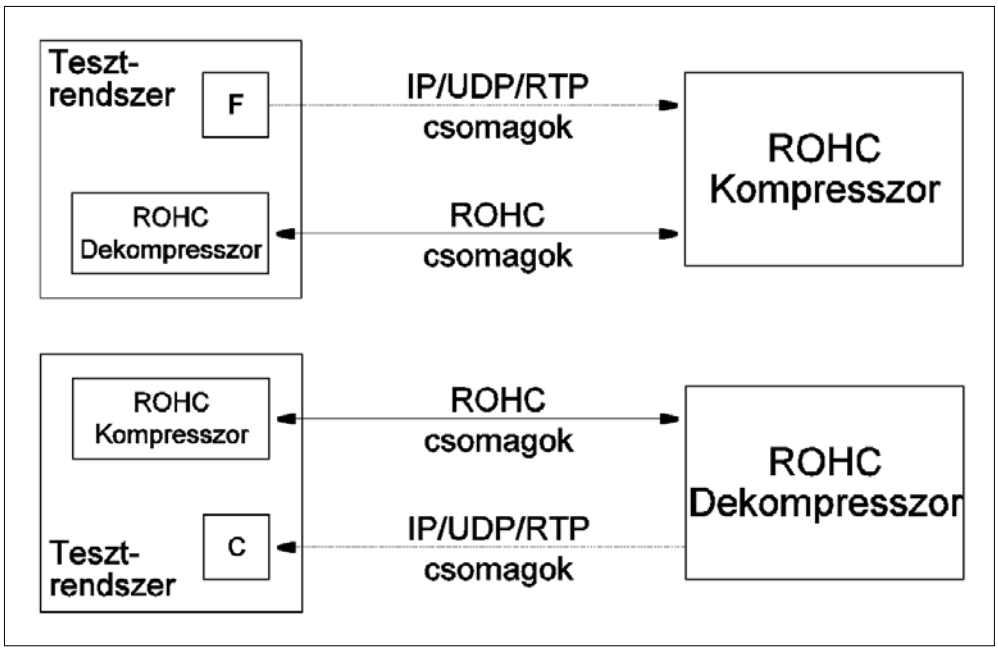
Az *1. ábrán* egy tipikus ROHC elrendezésre láthatunk példát. Az ábrán **F**-el jelöljük a forrás, és **C**-vel a célállomást. A forrásból jövő csomagokat a ROHC Kompresszor az aktuális állapotának megfelelően tömöríti. Az eredeti csomagok a ROHC Dekompresszor visszaállítja és elküldi a célállomásnak. A forrás és célállomás számára az eljárás teljesen átlátszó és semmilyen megkötést nem tartalmaz.

## 3. Konformancia-tesztelés

Konformancia-tesztelés során azt ellenőrizzük, hogy egy protokoll implementáció megfelel-e a szabványnak. Távközlési berendezések és protokollok esetében ez a tesztelési mód az elfogadott. Vannak olyan szabványosítási testületek (pl. ETSI, ITU, 3GPP), amelyek konformancia-tesztsorozatokat is készítenek, illetve szabványosítanak – ezeket egy szintén szabványos tesztleíró nyelven adják ki. A berendezések gyártói pedig ezen szabványos tesztsorozatok segítségével tesztelik termékeiket és ellenőrzik, hogy azok megfelelnek-e a vonatkozó szabványoknak.

Konformancia-tesztek során az implementáció belső működését nem ismerjük, ahhoz csak a specifikációban meghatározott interfészeken keresztül, protokollüzenetek felhasználásával férünk hozzá. Ahhoz, hogy egy protokollhoz könnyű legyen eszköz-független konformancia-teszteket írni, a külső interfészek pontos definiálása, valamint az interfészeken keresztül elérhető szolgáltatások minél szélesebb köre szükséges (például a protokoll különböző paramétereinek beállítása). Mivel a konformancia-teszteket szabványos (vagyis minden implementáción egyforma) interfészeken keresztül végezzük, ezért lehetnek a konformancia-tesztsorozatok implementáció-függetlenek. Ez azt is jelenti, hogy ugyanazt a tesztsorozatot változtatás nélkül futtathatjuk le különböző protokoll megvalósításokon.

Nagybonyolultságú berendezések esetében előfordul, hogy együttműködési teszteket is elvégeznek, de csak a konformancia-tesztek után. Ezeket az együtt-




*2. ábra*
*Konformancia-tesztelés elrendezése kompresszor és dekompresszor tesztelése esetén*

működési teszteket általában már a majdani vevő kérésére és annak hálózatában végzik. A cél ekkor annak megállapítása, hogy az adott eszköz képes-e együttműködni a meglévő berendezésekkel, amelyek általában több különböző gyártó termékei.

Konformancia-tesztelésben alapvetően háromféle teszttípust különböztetünk meg:

– normál viselkedés:
  a normál működés során fellépő eseteket vizsgáljuk, hibás üzeneteket nem küldünk;
– negatív viselkedés:
  az implementáció viselkedését vizsgáljuk olyan esetekben, amikor az szintaktikailag hibás üzeneteket kap;
– helytelen viselkedés:
  szintaktikailag helyes, de szemantikailag helytelen üzenetekre ellenőrizzük az implementáció viselkedését.

Természetesen nem tudunk minden lehetséges esetet megvizsgálni konformancia-tesztelés során, de a fenti három teszttípus mindegyike előfordul egy körültekintően megírt tesztsorozatban.

### 3.1. A ROHC protokoll konformancia-tesztelése

Konformancia-teszt esetében a kompresszort és a dekompresszort is külön-külön kell vizsgálnunk. Mindkét teszt-elrendezésre a *2. ábrán* láthatunk példát.

Kompresszor tesztelése esetén a tesztrendszerünk egyszerre viselkedik forrásként és dekompresszorként. A forrás előállítja az IP csomagokat, a dekompresszor segítségével pedig ellenőrizzük, hogy a vizsgált kompreszszor megfelelő ROHC csomagok állít-e elő. Az a ROHC csomag számít megfelelőnek, amely mind szintaktikailag, mind szemantikailag helyes és nem utolsó sorban az, amelyikből az eredeti IP csomag visszaállítható.

Dekompresszor-teszt esetében a tesztelő rendszerünk egy kompresszort és egy célállomást helyettesít. A dekompresszornak olyan ROHC csomagokat küldünk, amit egy jól működő dekompresszor képes feldolgozni és abból az eredeti IP csomagokat visszaállítani. A dekompresszor által rekonstruált IP csomagokat a tesztrendszer összehasonlítja az eredetivel (vagyis azzal, amiből a kompresszorunk a ROHC csomagokat előállította) és ebből megállapítja, hogy a tesztelt dekompresszor helyesen működik-e.

## 4. Együttműködés-tesztelés

Az együttműködés-tesztelés során azt ellenőrizzük, hogy ugyanazon protokoll két különböző megvalósítása képes-e egymással – a specifikációnak megfelelően – kommunikálni.

Fontos tulajdonsága az együttműködés-tesztelésnek, hogy nem egy, hanem egyszerre két implementációt vizsgál. Ha egy teszt során hibát észlelünk, akkor nem csak azt kell megállapítani, hogy mi okozta a hibát, hanem azt is, hogy melyik implementáció miatt történt a hiba. Mivel ezen implementációkat különböző cégek készítik (egy cégen belül ritkán készítenek több független implementációt), ezért érdekes módon a gyakorlatban nem az a legfontosabb kérdés, hogy mi volt a hiba, hanem az, hogy melyik cég terméke okozta azt.

Természetesen együttműködés-tesztelés után is maradhatnak még hibák az implementációkban. Előfordulhat például, hogy a specifikáció félreérthető vagy nem egyértelmű és mindkét implementációban ugyanazon a (hibás) módon valósítanak meg egy funkciót. Az ilyen fajta hibákat az együttműködés-tesztelés során nem mindig lehet felderíteni.

Az együttműködés-tesztelést azonban nem kizárólag csak protokoll-megvalósítások ellenőrzésére használják. Az IETF-ben (Internet Engineering Task Force – nemzetközi szabványosítási szervezet, amely internetes protokollokkal foglalkozik) magát a szabványosítás alatt álló protokollt is együttműködés-tesztelés során validálják. Egy protokoll specifikációt csak akkor fogadnak el szabványként, ha a protokoll két, egymástól független megvalósítása képes együttműködni. Vagyis a két implementáció együttműködés-tesztje tulajdonképpen magának a protokoll-specifikációnak egyfajta ellenőrzése is.

A bevezetőben már említettük, hogy az IP alapú protokollok esetében a specifikációk nem olyan részletesek, mint a távközlési protokollok esetében. Ennek következménye, hogy az egymásra épülő protokollok között nincsen olyan jól definiált, éles határ. Együttműködési tesztnél azonban nincs is erre szükség. Egyszerűen csak összekötjük a két berendezést és megfigyeljük, hogyan működnek. Az egyes üzenetek pontos megadása helyett a felhasználói interfészen keresztül utasítjuk a protokoll-implementációt a kívánt teszt (például kapcsolatfelvétel) lefuttatására. Ez az interfész általában implementáció-függő, tehát az adott rendszer alapos ismerete szükséges a tesztek végrehajtásához. Az együttműködés-tesztelés így egyrészt egyszerűbb – mert nem kell a protokollüzeneteket egyesével előállítani és elemezni, másrészt viszont bonyolultabb – mert csak az adott implementáció ismeretében lehet a teszteket végrehajtani.

A konformancia-tesztelésnél ismertetett háromféle teszttípus közül (normál, negatív, helytelen) az együttműködés-tesztelésnél általában csak egyetlenegy fajtát tudunk vizsgálni, ez pedig a normál működés tesztelése. Mivel a protokollüzeneteket egy – a szabványnak elvileg megfelelő – implementáció állítja elő, így a legtöbbször nincs lehetőségünk sem a negatív, sem pedig a helytelen viselkedés ellenőrzésére. A következő szakaszban láthatjuk majd, hogy hogyan lehet mégis ilyen eseteket is vizsgálni.

Az utóbbi időben érdekes tendencia figyelhető meg az együttműködés-tesztelés területén. Ahogyan egyre fontosabbá válnak az IP alapú protokollok, úgy az azok ellenőrzésére leginkább használt együttműködés-tesztelés is kezd egyre szervezettebbé válni. A szervezettség fokát tekintve az alábbi öt szintet különböztethetjük meg:

1. *Ad-hoc módon*

Ez a teljesen szervezetlen esetet jelenti, amikor mindenféle előzetes elképzelés és tervezés nélkül zajlik a tesztelés. A fejlesztők vagy megbeszélik előre, hogy tesztelni fognak vagy csak véletlenül találkoznak, például egy IETF megbeszélés után.

2. *Szervezett formában, segédeszközök nélkül*

Ebben az esetben már előre megbeszélik, hogy mikor fognak találkozni, és esetleg még tervet is készítenek arra, hogy a szabvány mely részeit fogják alaposabban leellenőrizni. A teszteléshez a szükséges alapvető környezetet (asztalok, székek, konnektorok, hálózati kapcsolat) a vendéglátó biztosítja.

3. *Szervezett formában segédeszközökkel*

Az előző szinttől abban különbözik, hogy itt az együttműködés tesztet szervezők az alapvető környezeten kívül olyan segédeszközöket is biztosítanak, amivel a tesztelés nemcsak könnyebb, de esetleg megismételhető is lesz (például ROHC esetében egy megfelelő forrás, amely képes ugyanazokat a csomagokat kiküldeni vagy megadott minta szerint változtatni).

4. *Informális leírás alapján*

A fentieken túl, ebben az esetben előre készítenek egy dokumentumot a végrehajtandó tesztekről. Ez tartalmazza az egyes tesztesetek részletes – szöveges – leírását a konfigurálástól a teszt közben végrehajtandó lépéseken keresztül egészen a teszt kiértékeléséig.

5. *Formális leírás alapján (automatikusan)*

Ez tulajdonképpen egy szabványos tesztleíró nyelven megírt tesztsorozat, amely az implementáció-függő paraméterek specifikálása után végrehajtható. A tesztsorozat összes tesztesete ilyenkor automatikusan – külső beavatkozás nélkül futtatható.

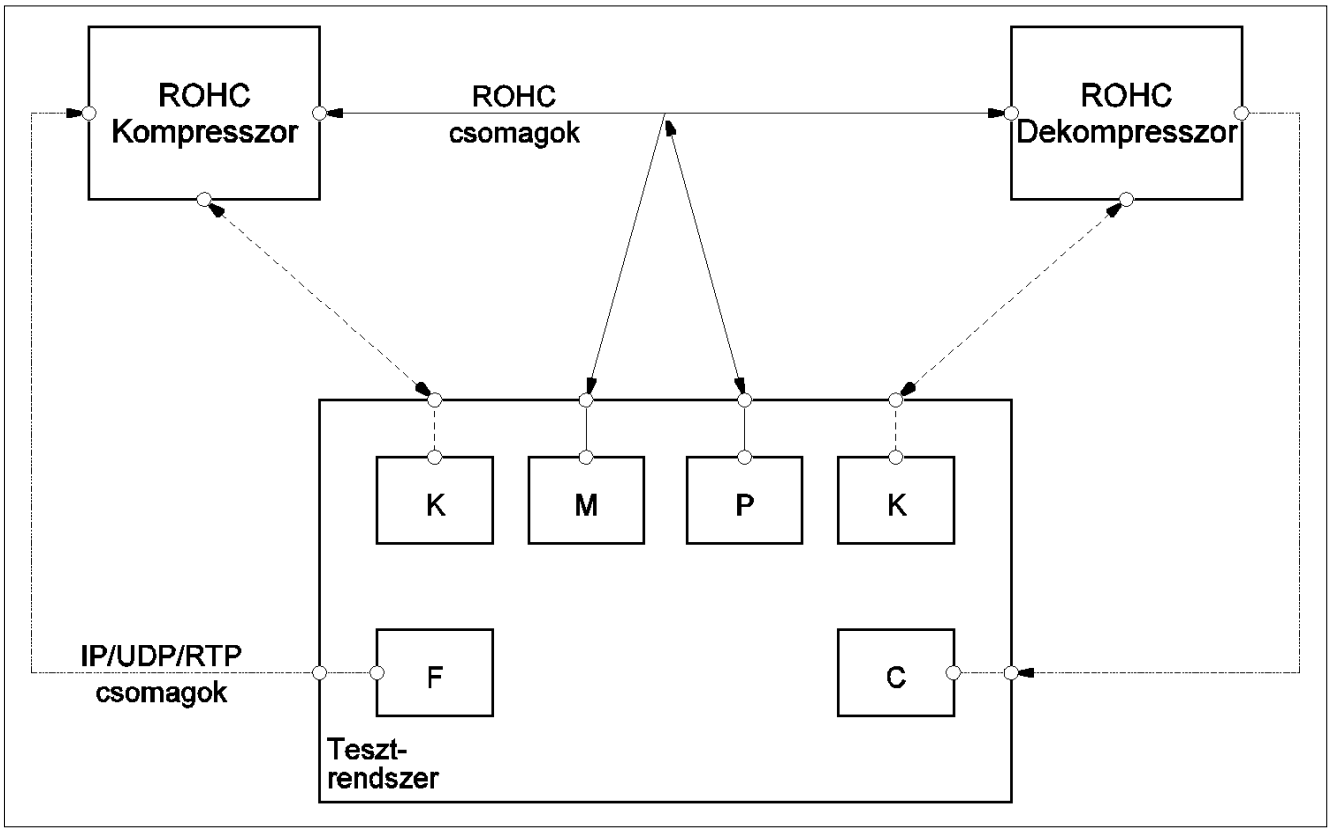
Ahogyan az egyes szinteken haladunk egyre feljebb, úgy válik a tesztelés egyre hatékonyabbá, vagyis a magasabb szinteken egyre kevesebb időre van szükség ugyanazon teszteset lefuttatására. A ROHC protokoll példájánál maradva, az első két szinten még az sem biztosított, hogy a forrás által kiküldött, illetve a célállomáshoz érkező csomagokat valamilyen módon rögzítsük. Ha pedig valahogy mégis sikerül rávenni az implementációnkat, hogy ezeket az adatokat elmentse, akkor ütközünk a következő problémába a különböző formátumú adatok összehasonlításával.

A konformancia-tesztelés már az 5. szinten, az együttműködés-tesztelés egyelőre a 2-3., nagyon ritkán pedig a 4. szinten helyezkedik el. Ez persze nem azt jelenti, hogy az egyik módszer jobb, a másik pedig rosszabb. Egyszerűen csak arról van szó, hogy az együttműködés-tesztelés még nem ért el arra a szintre, ahová a konformancia teszt. A fejlődés azonban egyértelműen látható, tehát már csak idő kérdése és az együttműködés-teszteket is automatikusan lehet majd végrehajtani.

### 4.1. A ROHC protokoll együttműködés-tesztelése

A *3. ábrán* az automatizált együttműködés-teszteléshez használt teszt-elrendezés látható. Az automatizált tesztek futtatásához a tesztrendszernek egy tesztleíró nyelven előre rögzített, összetett feladatot kell elvégeznie. Ebbe beletartozik a forrás és célállomások szimulálása, az implementációk konfigurálása, a kom-

*3. ábra Tesztelrendezés a ROHC protokoll automatikus együttműködés-teszteléshez*



presszor és dekompresszor közti csatorna monitorozása, valamint a forrás által kiküldött és a célállomás által fogadott csomagok összehasonlítása is.

A tesztrendszer az alábbi komponensekből áll:

- **F** – forrásként viselkedik, előállítja a kompresszor számára az IP csomagokat.
- **C** – célállomást helyettesíti, fogadja a dekompresszortól érkező csomagokat.
- **K** – e komponensek felelnek a kompresszor és a dekompresszor helyes beállításáért (konfigurálás).
- **M** – monitorozás a feladata, figyeli a kompresszor és a dekompresszor közötti ROHC üzeneteket.
- **P** – protokoll üzeneteket képes kiküldeni a csatornára, így lehetővé válik a helytelen viselkedés vizsgálata is. Ha a kompresszor és a dekompresszor nem áll egymással közvetlen kapcsolatban, hanem ezen a komponensen keresztül kötjük őket össze, akkor bizonyos csomagok eldobásával és helyettük hibás csomagok küldésével még a negatív viselkedést is tesztelhetjük.

Egy automatizált együttműködés teszteset végrehajtása több lépésből áll. Először a kompresszort és a dekompresszort kell megfelelően konfigurálni. Mivel erre nincs egységes interfész, ezért olyan megoldásra van szükség, ami minden rendszeren elérhető. Egyik lehetséges megoldás egy telnet kapcsolat létrehozása a tesztrendszer és a tesztelendő implementációk között. Így egy parancssoros felület érhető el, amin keresztül az esetek nagy részében a konfigurálás elvégezhető. Az egyes parancsok természetesen továbbra is implementáció-függőek lesznek, de a parancsokat paraméterként is kezelhetjük, ezzel a tesztsorozat már implementáció-függetlenné tehető.

A következő lépésben megvizsgáljuk, hogy egy adott bemenő IP csomag-sorozatra a tesztelt rendszer megfelelő választ ad-e. A szimulált forrás előálltja az előre definiált csomagokat és elküldi azokat a kompresszornak. A kompresszor előállítja a tömörített ROHC csomagokat, majd ezekből a dekompresszor visszaállítja az IP csomagokat. Az ellenőrzés kétféle módon történik, egyrészt a tesztrendszer figyeli a kompresszor és a dekompresszor közötti forgalmat, másrészt pedig összehasonlítja a forrás által kiküldött és a célállomás által fogadott IP csomagokat. Így nem csak azt tudjuk megállapítani, hogy a tömörítés és visszaállítás sikeres volte, hanem azt is, hogy a kompresszor megfelelő hatékonysággal működik-e.

Ez a tesztelrendezés egyébként nem csak tisztán együttműködés-tesztelésre használható. A monitorozó komponens megfelelő programozásával egyfajta konformancia jellegű együttműködés-tesztek végrehajtására is alkalmas. Vagyis akkor is lehet sikertelen egy ilyen teszt, ha az eredeti IP csomagok visszaállítása sikeres volt (vagyis az együttműködés sikeres), de a kompresszor nem a megfelelő vagy nem a szabvány által előírt legnagyobb hatékonyságot biztosító ROHC csomagformátumot választotta (vagyis a szabvánnyal nem konform).

## 5. Összefoglalás

A cikkben bemutattuk a konformancia-tesztelés és az együttműködés-tesztelés menetét és tulajdonságait. Látható, hogy egyik módszer sem jobb a másiknál, egyszerűen csak különbözőek. Sem a konformancia-, sem az együttműködés-tesztekkel nem lehet 100% biztonsággal ellenőrizni egy protokoll-implementációt. Ennek szemléltetéséhez nézzük meg az alábbi két példát:

*1. példa:*

Egy specifikáció szerint egy adott üzenetre 1 másodpercen belül kell az implementációnak választ küldenie. Az implementáció választ is küld, amely szintaktikailag és szemantikailag is helyes, azonban 2 másodperces késéssel. Ebben az esetben a konformancia-teszt sikertelen, de az együttműködés teszt – a másik implementáció viselkedésétől függően – még lehet sikeres.

*2. példa:*

Egy protokoll implementáció képes kapcsolatot felépíteni a szabványban meghatározott módon, így a konformancia teszt sikeres. Viszont nem képes ezt olyan ütemben vagy mennyiségben megtenni, ahogyan azt egy másik implementáció várná, így az együttműködés teszt sikertelen lesz.

Könnyen belátható tehát, hogy az egyik fajta teszt sikeressége nem garantálja a másik fajta teszt sikerét is, hiszen a két fajta teszt célja eltérő; az implementáció más-más tulajdonságát vizsgálja. A gyakorlatban ezért szükség van mindkét tesztelési eljárásra.

Manapság már mindenki kezdi elfogadni, hogy sem a szabványnak való megfelelést igazoló tanúsítvány (konformancia-teszt), sem pedig az implementáció/berendezés együttműködési képességeit vizsgáló teszt önmagában nem elég. Éppen ezért már nem az a kérdés, hogy melyik fajta tesztre van szükség, hanem sokkal inkább az, hogy melyik módszert mikor célszerű használni, milyen sorrendben, és hogyan lehetne a kettőt minél kevesebb ráfordítással végrehajtani.

**Irodalom**

[1] C. Bormann (ed.):
Robust Header Compression (ROHC),
RFC 3095, 2001 július

[2] ETSI TS 102 237-1 V4.1.1 (2003-12),
Telecommunications and Internet Protocol
Harmonization Over Networks (TIPHON) Release 4:
Interoperability test methods and approaches,
Part 1: Generic approach to interoperability testing

# Teljesítményvizsgálat elosztott tesztkomponensekkel

CSORBA J. MÁTÉ, PALUGYAI SÁNDOR

*Ericsson Magyarország, Test Competence Center*
*{mate.csorba, sandor.palugyai}@ericsson.com*

***Lektorált***

***Kulcsszavak: TTCN-3, teljesítményvizsgálat, párhuzamos tesztkomponensek***

*Cikkünkben TTCN-3 tesztkörnyezeten alapuló teljesítmény- és terhelésvizsgálatokra alkalmas szoftverkomponensek elemzésével foglalkozunk. Az általunk adott módszer célja teljesítménytesztek fejlesztésének támogatása a tesztkomponensek teljesítmény-kritikus viselkedésének előrejelzésével, már a korai tervezési fázis során. Módszerünk legfőképpen a tesztkomponensek megvalósításában rejlő sorbanállási mechanizmusokat veszi figyelembe, más TTCN-3 specifikus tulajdonságok mellett, melyek befolyásolhatják egy tesztrendszer késleltetéseit. A módszer használatával megbecsülhetjük azt a forgalmi korlátot mely alatt nagy biztonsággal érdemes TTCN-3 alapú tesztkomponenseket használnunk az adott vizsgálatainkra.*

## 1. Bevezetés

Távközlési berendezések, hardver és szoftver eszközök tesztelése kapcsán vizsgálatok széles skálájáról beszélhetünk. Végezhetünk funkcionális teszteket, protokollmegfelelőségi vizsgálatokat (konformancia-tesztelés), vizsgálhatjuk távközlési berendezések, valamint szoftver-megvalósítások együttműködési képességét, robusztusságát (stabilitás és megbízhatóság vizsgálata). Távközlési szoftverek fejlesztése során nagy jelentősége van az úgynevezett regressziós teszteknek, amelyek a folyamatosan fejlesztett szoftver újabb és újabb verzióinak gyakran ismételt vizsgálatára alkalmasak.

Általában egy rendszer reakciós idejét, illetve terhelhetőségét teljesítménytesztekkel állapíthatjuk meg. A teljesítményvizsgálatok körében is többfajta vizsgálati céllal találkozhatunk, úgy mint skálázhatóság vizsgálata, mennyire könnyen bővíthető egy rendszer. Stressz-teszt, azaz a rendszer viselkedése hirtelen és/vagy rendkívül túlméretezett terhelés alatt [1]. Teljesítményvizsgálat során azt a várható környezetet és terhelést próbáljuk meg előállítani, amellyel majd a működő rendszernek szembe kell néznie, és arra keressük a választ, mekkora az a terhelés, amelyet a rendszer még elvisel, illetve mekkora hibaarányt produkál, késleltetése hogyan változik a terheltség függvényében. A legtöbb esetben több felhasználó párhuzamos, egyidejű működésének szimulációjával vizsgáljuk az adott távközlési rendszert és miközben változó számú felhasználói populációt szimulálunk, vizsgáljuk a rendszerrel történő kommunikációt. Ez a tesztek részéről hatékony és elosztott működést feltételez.

Mivel a vizsgált megvalósítást fekete dobozként kezeljük, és pusztán kívülről stimuláljuk protokoll-üzenetek formájában, a tesztrendszernek képesnek kell lennie a megfelelő mennyiségű tesztüzenet előállítására. Ez gyakran a teszteket futtató hardver platform képességeibe ütközhet, amit általában még több hardverelem munkába állításával tudunk megelőzni. Ugyanakkor a felhasználók nem szekvenciális, egyidejű viselkedését úgyszintén több, párhuzamosan futó processz alkalmazásával tudjuk hatékonyan szimulálni.

Akár egy elég erős hardveren futó, több párhuzamos felhasználót szimuláló, akár a rendelkezésre álló több gépet hatékonyan kihasználó tesztek esetében elosztott tesztkomponensekről beszélhetünk. Mindkét esetben felmerül a kérdés, milyen stratégia szerint osszuk szét a funkciókat a tesztkomponensek között, valamint miként helyezzük el az egyes komponenseket a vizsgálatokban résztvevő hardveren, munkaállomásokon. A megfelelő stratégiának, mely szerint a tesztkomponenseket elhelyezzük, figyelembe kell vennie a munkaállomások kiépítettségét és ebből adódó eltérő sebességét csakúgy, mint az egy adott munkaállomáshoz rendelt szimulált felhasználók számát.

Egy végpont a tesztek futtatásához, elegendő felhasználó imitálásához, nem mindig rendelkezik elég erőforrással. További erőforrás szükségletet jelent, hogy az egyes tesztkomponensek egymás között is kommunikálhatnak és a konkrét teszt megvalósításától függően kommunikálnak is. Például koordinációs, szinkronizációs, start/stop üzenetek cseréje, futás közbeni statisztikai és kiértékelést segítő lekérdezések. A komponensek viselkedésének vizsgálatánál ezt a belső kommunikációt is figyelembe kell venni.

A teszteléshez használt szoftverkomponensek elemzésének célja, hogy támogassa a munkaállomásokon történő szétosztásukat. Az elosztási mechanizmusokat megkülönböztetjük aszerint, hogy a mechanizmus statikus, vagy dinamikus működésű, tehát csak a teszt futtatását megelőzően, vagy futtatás közben is használható [2].

A különböző elosztási stratégiák mindegyike figyelembe veszi a kialakított hardver/szoftver struktúra valamely tulajdonságát a döntések megkönnyítésére, úgy mint a processzorterhelést, memóriafoglalást, I/O műveletek gyakoriságát, a hálózati interfészek paramétereit. A felsorolt tulajdonságok figyelembevétele történ-

het statikusan, a komponensek elosztását megelőző számítások támogatásához, illetve folyamatosan egy kialakított hardver-monitorozó keretrendszer segítségével.

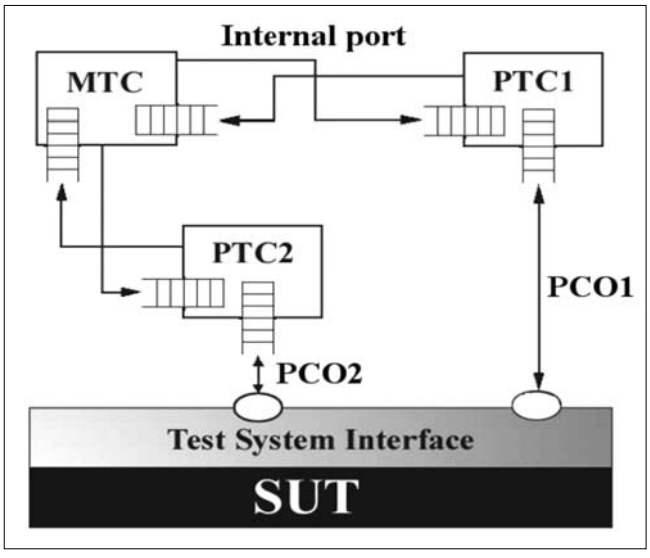
A következőkben egy sztochasztikus modellt adunk teljesítményvizsgáló komponensek viselkedésének becslésére [3], mellyel statikus elosztás valósítható meg. Használatával egy adott tesztkomponens üzenetvesztési valószínűségeit becsülhetjük meg különböző bemenő paraméterek mellett, és választ kaphatunk arra, hogy a komponens a bemenő paraméterekben leírt hardveren képes lesz-e a teszt-specifikációban leírt követelmények teljesítésére.

## 2. Teljesítménytesztelés elosztott TTCN-3 tesztkomponensekkel

Vizsgálataink során a Testing and Test Control Notation version 3 (TTCN-3) [4] tesztnyelvre, és az ezen a nyelven történő fejlesztésre és elosztott tesztvégrehajtásra koncentrálunk. A TTCN-3 tesztkörnyezetet széles körben alkalmazzák távközlési rendszerek tesztjeinek megvalósítására, de egyéb területeken is megvetette már a lábát. Jó példa erre gépjárművek kommunikációs rendszereinek vizsgálata.

Az utóbbi időben egyre nagyobb az igény a TTCN-3 nyelv használatára teljesítményvizsgálatok körében, ennek megfelelően több tanulmány is foglalkozik a nyelv ilyen irányú lehetőségeivel, kiterjesztésével és alkalmazásával [5-8]. Mivel a TTCN-3 nyelv egy magas szintű specifikációs nyelv, kérdéses lehet a hatékonysága egy alacsonyabb szintű, hardverközeli megvalósítással szemben, teljesítménytesztek írása esetén. Azonban számos úttörő projekt, számítás és mérés megmutatta, hogy a magas szintű tesztnyelv képes a hardver által nyújtott korlátok és erőforrások teljes kihasználására abban az esetben, ha a tesztek írása körültekintően és néhány ökölszabály betartásával történik.

*1. ábra*
*TTCN-3 tesztkomponensek és kommunikációs portok egy lehetséges elrendezése*




A TTCN-3 nyelvű párhuzamos tesztkomponensek (Parallel Test Components, PTCs) statikus vagy dinamikus elosztására kétfajta megközelítés ismert. A komponenseket elosztó mechanizmus megvalósítható egy köztes réteg (middleware) beiktatásával és az elosztó mechanizmus külön implementálásával, valamilyen más nyelven [9]. Ez azonban, a middleware megvalósításától függően, egy újabb szűk keresztmetszet lehet a tesztrendszer számára. Ezért mi a komponensek elosztását és az úgynevezett terheléselosztást (load control) szintén TTCN-3 nyelven valósítjuk meg, kihasználva a nyelv által adott lehetőségeket. A nyelv által biztosított platformfüggetlenségnek köszönhetően a PTC-k elosztása könnyen testre szabható különböző hardver környezet használatához.

## 3. Tesztkomponensek sztochasztikus vizsgálata

Az általunk felállított sztochasztikus modell TTCN-3 nyelvű PTC-k eseményfeldolgozó kapacitásának figyelembevételével képes megbecsülni az adott komponens üzenetvesztési valószínűségét, azaz használhatóságának korlátját. A tesztrendszer üzenetei lehetnek belső (koordinációs) üzenetek, illetve a külvilággal történő kommunikációt megvalósító, tényleges tesztüzenetek. Az elosztott tesztrendszer több PTC-ből és egy fő, vezérlő komponensből (Main Test Component, MTC) állhat. A köztük lévő kommunikáció az úgynevezett test portokon (Points of Control and Observation, PCOs) keresztül történik, csakúgy mint a vizsgálat célját képező rendszerhez (System Under Test, SUT) történő kapcsolódás *(1. ábra)*.

A kommunikációs portok elméletileg egy-egy különálló várakozó sorral rendelkeznek, amely a gyakorlatban a tesztkomponenst futtató munkaállomás memóriájának egy szelete. Ezek a PCO-k egymással versengenek a rendszer erőforrásaiért. A versenyhelyzetet meghatározza az egyes portokra érkező igények érkezési intenzitása, illetve a tesztkomponens által futtatott teszteset jellemzői, bonyolultsága.

A komponensek vizsgálatára egy diszkrét idejű kvázi születési-halálozási folyamat (D-QBD) alapú modellt állítunk fel [10], melynek segítségével a komponenshez tartozó PCO-k versenyhelyzetének elemzésével becslést adunk a komponens üzenetvesztésére vonatkozóan. A becslés történhet a tényleges megvalósítást megelőzően (visszacsatolás a tervezési fázisban), illetve a megvalósítást követően a PTC-k munkaállomásokra történő elosztási stratégiájának támogatására (statikus elosztás).

A PTC viselkedését leíró modell a következő TTCN-3 specifikus mechanizmusokat veszi figyelembe: a komponensen futó teszteset alternatíváinak – melyek leírják a lehetséges végrehajtási utakat (a SUT-tel történő kommunikáció protokolljának véges állapotú automatája alapján) – működése. Az úgynevezett snapshot logika [11] – mely a TTCN-3 üzenetfeldolgozás során hasz-

nált mintaillesztési mechanizmusa – által okozott késleltetés. Valamint, a bejövő és kimenő üzenetek teszt portokon (PCO-kon) kialakuló sorbanállásának hatásai.

Ahhoz, hogy ezeket a mechanizmusokat a modell képes legyen számításba venni, a következő bemenő paraméterekkel rendelkezik:

(a) illeszkedési valószínűségek (találati arányok) a tesztkomponens alternatíváinak minden egyes (a TTCN-3 kódban található *alt* struktúra összes) bejegyzésére,

(b) érkezési intenzitások a tesztkomponens minden egyes portjához külön-külön.

Minden egyes PCO-hoz a modell feltételezése szerint Poisson-eloszlás alapján érkeznek a protokoll-üzenetek, így ebben az esetben az eloszlás várható értékét adjuk meg. Továbbá bemenő paraméter még – mivel diszkrét idejű modellről van szó –, a diszkrét időegység, melyet annak az időnek a függvényében állítunk be, ami a komponenst futtató munkaállomáson szükséges egy beérkező üzenet megvizsgálásához (template matching művelet végrehajtásához). Ez a paraméter gyakorlatilag a CPU sebességétől függ. Ezeknek a paramétereknek a segítségével építjük fel a D-QBD-t egyértelműen definiáló állapot-átmeneti mátrixot (*P* mátrix).

Esetünkben a D-QBD folyamat egy speciális QBD lesz, mely irreguláris 0. szinttel rendelkezik és a szintek száma véges, ennek megfelelően alakul a folyamatot leíró *P* mátrix és almátrixai:

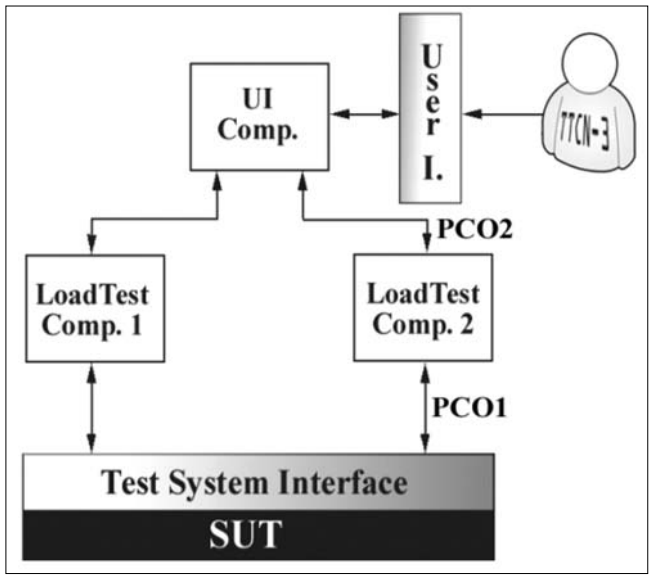
$$\underline{\underline{P}} = \begin{bmatrix} \underline{\underline{B}}^* & \underline{\underline{C}}^* & 0 & 0 & 0 & \dots \\ \underline{\underline{A}}^* & \underline{\underline{B}} & \underline{\underline{C}} & 0 & & \\ 0 & \underline{\underline{A}} & \underline{\underline{B}} & \underline{\underline{C}} & 0 & \\ 0 & 0 & \underline{\underline{A}} & \underline{\underline{B}} & \underline{\underline{C}} & \\ & & & \dots & \dots & \underline{\underline{C}} \\ & & & & \underline{\underline{A}} & \underline{\underline{B}}^{**} \end{bmatrix} \quad (1)$$

Az így kapott QBD modell ezután hatékonyan számítható mátrix-analitikus módszerekkel, melynek során a folyamat állandósult állapotbeli megoldását keressük, és így kapjuk a folyamat tetszőleges állapotának előfordulási valószínűségét [12]. Az általunk definiált véges, kétdimenziós modell két konkurens PCO-t használó tesztkomponens leírására alkalmas. Kettőnél több PCO leírása N-dimenziós modell felállítását teszi szükségessé, melynek számítása jelenleg is izgalmas feladat, de nem megoldhatatlan [13].

## 4. Alkalmazási példa

Tételezzük fel, hogy tesztrendszerünk három párhuzamos TTCN-3 komponensből áll *(2. ábra)*. A rendszer felhasználója egy grafikus felületen keresztül kapcsolódik egy komponenshez, ami a felületet kezeli, és vezérli a két másik, ténylegesen teljesítményvizsgálatot végző PTC-t.

Ilyen elrendezésben vizsgáljuk a *LoadTestComp.2* nevű PTC működését, és kiszámítjuk a csomagvesztés valószínűségét a komponens belső megvalósításának

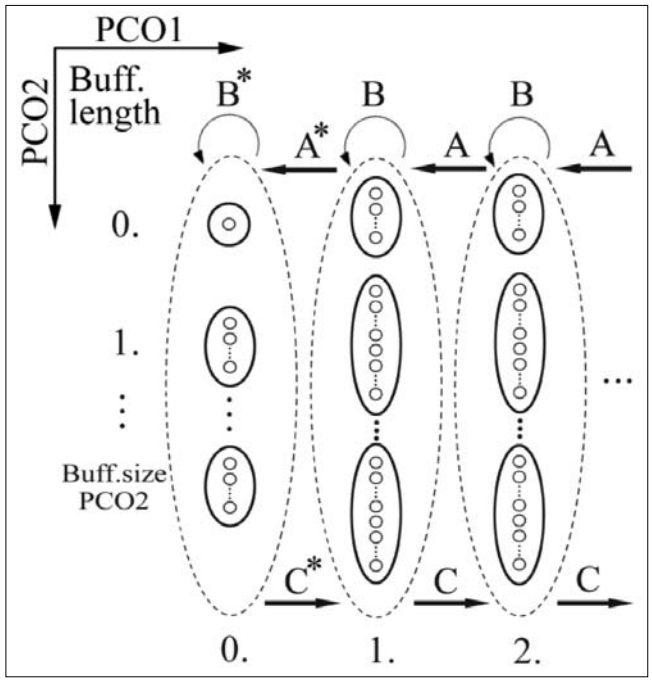



*2. ábra Tesztelrendezés három PTC-vel*

és két kommunikációs portjára (PCO1 és PCO2) érkező csomagok intenzitásának függvényében. Az így kapott kétdimenziós modell látható a *3. ábrán,* a teszt-portok pufferelését leíró szintekkel és – a szinteken belül – a TTCN-3 komponens alternatíváit leíró fázisokkal.

A vizsgálat tárgyául (SUT) ebben az esetben sokféle távközlési berendezés elképzelhető, például egy ATM kapcsoló, vagy egy IP útvonalválasztó. A példabeli elrendezésben a vizsgált komponens a PCO2 jelű porton keresztül kommunikál a felhasználóval kapcsolatot tartó *UI* komponenssel, míg a tényleges teljesítményvizsgálat protokoll üzenetei a PCO1 porton érkeznek és távoznak. Ez egy teljesítmény vizsgáló komponens esetében azt is jelenti, hogy a két port forgalmának intenzitása jelentősen eltér. A felhasználó konfiguráló üze-

*3. ábra*
*A kétportos PTC-t modellező kétdimenziós QBD folyamat*



netei, illetve lekérdezései (PCO2) – már csak az emberi reakcióidő viszonylagos lassúsága miatt is – rendkívül ritkának tekinthetők a SUT-tel történő kommunikációhoz hasonlítva, melynek nagyságrendje másodpercenként több ezer üzenetet is jelenthet.

A tesztkomponensek tervezéséhez megvizsgálhatjuk a PTC által még éppen kezelhető forgalom intenzitását a modell segítségével. Tekintsük bemenő paraméternek az érkezési intenzitásokat és számítsuk ki a modell állandósult állapotbeli megoldását. Így kapjuk a stacionárius állapot-valószínűségeket ($\pi_i$ vektorok) a modell minden állapotára és minden szintjére vonatkozóan (2). A vektorok kiszámítása minden *n*-re a rekurzív *R* mátrix segítségével történhet, mely az állapot-átmeneti mátrix almátrixainak *(A, B, C)* segítségével kapható [12].

$$\underline{\pi}_1 = \underline{\pi}_0 \cdot \underline{\underline{R}}; \underline{\pi}_2 = \underline{\pi}_1 \cdot \underline{\underline{R}}; \ldots \underline{\pi}_n = \underline{\pi}_0 \cdot \underline{\underline{R}}^n \qquad (2)$$

Amint rendelkezünk az állandósult állapotbeli megoldással, kiszámíthatjuk a terhelés okozta versenyhelyzetben alulmaradó porton létrejövő üzenetvesztés valószínűségét (3). Ehhez összegeznünk kell azon állapotok valószínűségét, amelyekben tartózkodva a rendszer már nem képes újabb érkezéseket kezelni. Ezek az állapotok az irreguláris 0. szinten és a reguláris szinteken ($j = 1 \ldots \infty$) a PCO2 pufferméretének megfelelő utolsó (vertikális, lásd 3. ábra) szintek állapotai (*i, k* állapotok).

$$\Pr(Loss_{PCO2}) = \sum_i \pi_{0i} + \sum_{j=1}^{\infty}\left(\sum_k \pi_{j_k}\right) = \ldots \underline{\pi}_{0_l} + \sum_{j=1}^{\infty}(\underline{\pi}_{j_m}) =$$

$$\underline{\pi}_{0_l} + \sum_{j=1}^{\infty}(\underline{\pi}_{1_m} \cdot \underline{\underline{R}}^{j-1}) = \underline{\pi}_{0_l} + \underline{\pi}_{1_m} \cdot \sum_{k=0}^{\infty}(\underline{\underline{R}}^k) =$$

$$= \underline{\pi}_{0_l} + \underline{\pi}_{1_m} \cdot (\underline{\underline{I}} - \underline{\underline{R}})^{-1} \qquad (3)$$

Az egyszerűsítések után kapott formulával ezután megbecsülhetjük, mekkora az az érkezési intenzitás, a *LoadTestComp.2* névre hallgató komponens esetében, melyet adott üzenetvesztési valószínűség alatt még kezelni tud. Ezáltal a tesztek adott forgalomra méretezhetővé válnak még a tényleges futtatást megelőzően.

## 5. Összefoglalás

Teljesítménytesztek tervezése és megvalósítása bonyolult feladat TTCN-3-mal, de bármilyen más eszközzel is. Körültekintő tervezésnek kell megelőznie a teljesítménytesztek megvalósítását, hogy hatékonyan működjenek a végrehajtásra szolgáló platformon. A létrehozott tesztkomponens modellünk lehetővé teszi meglévő tesztek elemzését is, valamint képes egyfajta visszacsatolást nyújtani a tesztek tervezői számára.

További terveink közt szerepel a model továbbfejlesztése, hogy alkalmassá váljon több konkurens PCO-t használó PTC-k leírására is egy N-dimenziós modell megalkotásával.

**Irodalom**


[1] R. Binder, Testing Object-Oriented Systems: Models, Patterns and Tools. Addison-Wesley, 2000.

[2] G. Din, S. Tolea, I. Schieferdecker, „Distributed Load Tests with TTCN-3”, TestCom 2006, LNCS 3964, pp.177–196.

[3] M. J. Csorba, S. Palugyai, S. Dibuz, Gy. Csopaki, „Performance Analysis of Concurrent PCOs in TTCN-3”, TestCom 2006, LNCS 3964, pp.149–160.

[4] ETSI ES 201 873-1 (V3.1.1) Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, 2005.

[5] Z. Wang, J. Wu, X. Yin, X. Shi, B. Tian, „Using TimedTTCN-3 in Interoperability Testing for Real-Time Communication Systems”, TestCom 2006, LNCS 3964, pp.324–340.

[6] H. Neukirchen, Z. Ru Dai, J. Grabowski, „Communication Patterns for Expressing Real-Time Requirements Using MSC and their Application to Testing”, TestCom 2004, LNCS 2978, pp.144–159.

[7] Z. Ru Dai, J. Grabowski, H. Neukirchen, „Timed TTCN-3 Based Graphical Real-Time Test Specification”, TestCom 2003, LNCS 2644, pp.110–127.

[8] Z. Ru Dai, J. Grabowski, H. Neukirchen, „Timed TTCN-3 – A Real-time Extension for TTCN-3”, TestCom 2002, pp.407–424.

[9] I. Schieferdecker, T. Vassiliou-Gioles, „Realizing Distributed TTCN-3 Test Systems with TCI”, TestCom 2003, LNCS 2644, pp.110–127.

[10] G. Latouche, V. Ramaswami, „Introduction to Matrix Analytic Methods in Stochastic Modeling”, The American Statistical Association and the Society for Industrial and Applied Mathematics, 1999. pp.83–99., pp.221–237.

[11] ETSI ES 201 873-4 (V3.1.1) – Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics, 2005.

[12] M. F. Neuts, „Matrix-Geometric Solutions in Stochastic Models”, Johns Hopkins University Press, 1981. pp.81–107., pp.112–114.

[13] T. Osogami, „Analysis of multi-server systems via dimensionality reduction of Markov chains”, PhD. thesis, Computer Science Department, Carnegie Mellon University, 2005.

# TITAN: TTCN-3 tesztvégrehajtó környezet

SZABÓ JÁNOS ZOLTÁN, CSÖNDES TIBOR

*Ericsson Magyarország, Test Competence Center*
*{janos.zoltan.szabo, tibor.csondes}@ericsson.com*

***Kulcsszavak: tesztelés, TTCN-3, tesztrendszer megvalósítás***

*Cikkünkben bemutatjuk az Ericsson TITAN nevű TTCN-3 tesztvégrehajtó környezetét, annak működését és belső felépítését. Megvizsgáljuk a TITAN fő jellemzőit és a lényeges különbségeket más, kereskedelmi TTCN-3 eszközökhöz képest. Fejlesztésünk eredményeként a TTCN-3 és a TITAN széles körben használt tesztmegoldás lett az Ericsson-on belül, emellett lehetőségünk volt részt venni az ETSI-ben folyó TTCN-3 szabványosítási munkában.*

## 1. Bevezetés

A 80-as és 90-es években az OSI rétegmodellre épülő távközlési rendszerek tesztelésére a TTCN nyelv 2. változatát használták. Széleskörű elterjedésének gátat szabott a nyelv nehézkes táblázatos leíró formátuma, valamint a korlátozott alkalmazási terület. Ennek hatására az ETSI a 90-es évek végén a nyelv legújabb változatának, a TTCN-3-nak a kidolgozásához kezdett.

A nyelv tervezésénél figyelembe vették, hogy az eddigi konformanciateszt-alkalmazásokon felül új tesztelési módszerekre is megfelelő legyen, mint például:

– együttműködési teszt,
– teljesítményteszt,
– robosztussági teszt,
– rendszerteszt.

Az ETSI szakított a klasszikus ISO OSI modell alapú felfogással, ezzel lehetővé téve az IP alapú rendszerek, valamint a programinterfészek (API – Application Program Interface) tesztelését is.

### 1.1. A TTCN-3 nyelv

A TTCN-3 nyelv első szabványsorozatát 2000-ben adta ki az ETSI. Azóta több verziója látott napvilágot. A legutolsó, aktuális szabványgyűjteményt 2005-ben jelent meg [1]. A szabványosítás alatt megpróbálták elfeledni a nehézkes TTCN-2-es struktúrákat és ezzel elejét venni az ismételt negatív megítélésnek, ami a TTCN előző verziója kapcsán elterjedt a távközlési szakemberek körében. A törekvés nem volt hiábavaló, mivel egy könnyen átlátható és a tesztelést nagymértékben segítő nyelv lett az eredmény. A TTCN-3 szöveges formátuma nagyban hasonlít a széles körben elterjedt C/C++ programozási nyelvre, így sok felhasználó számára mélyebb TTCN-3-as ismeret nélkül is érthetőek a nyelvi struktúrák. A TTCN-3 nyelvről és elemeiről egy részletes összefoglaló cikkben olvashatunk [4].

### 1.2. A TITAN története

A TITAN kifejlesztése 2000. elején egy egyetemi diplomamunka keretében kezdődött. A fejlesztés elsődleges célja egy teljesítménytesztek végrehajtására is alkalmas, hatékony, de ugyanakkor protokoll- és alkalmazásfüggetlen futtatókörnyezet létrehozása volt. Ehhez jó alapot szolgált az ETSI-ben éppen kidolgozás alatt álló TTCN-3 nyelv.

Kevesebb, mint 1 év alatt készült el a rendszer első prototípusa, amely a nyelv lehetőségeinek csak egy részét támogatta, de belső felépítése és működése a mostani állapothoz nagyon hasonlított [5]. A TITAN azóta is folyamatos fejlődés alatt áll, követi a szabványok változásait, egyre több kényelmi funkcióval rendelkezik, és mára szinte az összes TTCN-3 nyelvi konstrukciót támogatja.

A fejlesztés főbb mérföldkövei az alábbiak voltak:

– 2000: prototípus készítése,
– 2001: párhuzamos és elosztott teszt végrehajtás,
– 2002-2003: ASN.1 nyelv támogatása, szemantikus ellenőrzéssel,
– 2004: grafikus felhasználói felület,
– 2005: teljes TTCN-3 szemantikus ellenőrzés.

## 2. A TITAN felépítése

A TTCN-3 fordító és futtató környezet blokkdiagrammja az *1. ábrán* látható.

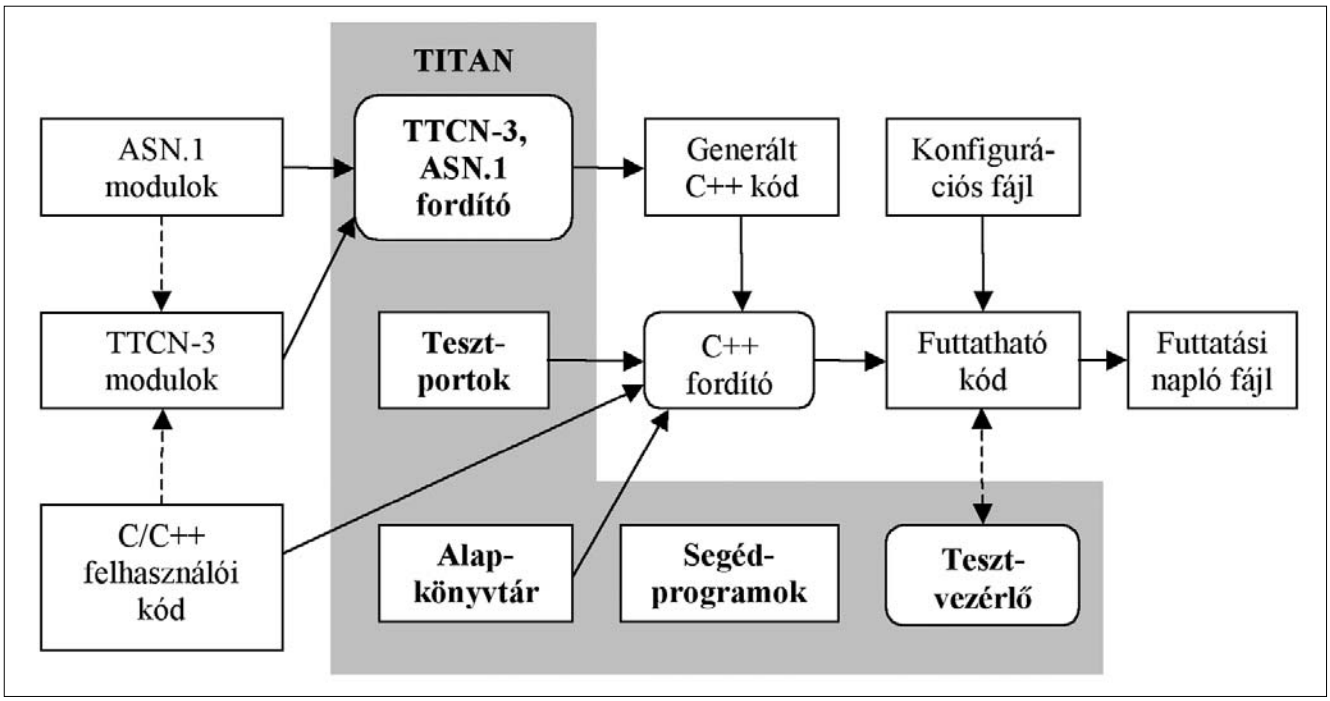
A TITAN részei a következők:

### 2.1. TTCN-3, ASN.1 fordító

A TTCN-3 és ASN.1 fordítóprogram a TITAN legnagyobb, legösszetettebb részegysége. Feladata a tesztkészletet alkotó modulok elemzése, ellenőrzése és a bennük található esetleges szintaktikai és szemantikai hibák jelzése. Ha a bemenet hibátlannak bizonyult, a fordító C++ nyelvű programmodulokat generál, melyek részét képezik a végrehajtható tesztsorozatnak.

### 2.2. Alapkönyvtár

Az alapkönyvtár tartalmazza a futtatható tesztsorozatoknak azt az állandó közös részét, amely független a tesztsorozatot alkotó modulok tartalmától. Az alap-




*1. ábra A TITAN blokkdiagrammja*

könyvtár kézzel megírt és előre lefordított C++ programkódból áll. Itt találhatók meg a TTCN-3 alaptípusait megvalósító osztályok, a nyelv beépített függvényeit és műveleteit (például időzítők, portok, komponensek kezelését) megvalósító funkciók. A tesztsorozat futtatásához szükséges egyéb, például a naplót készítő vagy a konfigurációs fájlt értelmező szoftvermodulok szintén az alapkönyvtár részét alkotják.

#### 2.3. Tesztportok

A tesztportok feladata az összeköttetés és a kommunikáció biztosítása a futtatható tesztsorozat és a tesztelendő rendszer között. A TTCN-3 nyelv a külvilággal való kapcsolatot kommunikációs portokon át küldött és fogadott absztrakt üzenetekkel modellezi. A tesztport tulajdonképpen egy TTCN-3 kommunikációs port C++ nyelvű megvalósítása a végrehajtható tesztsorozatban.

A TITAN egy jól definiált programozói interfészt, úgynevezett tesztport API-t biztosít a kimenő és bejövő üzenetek kezelésére. A tesztport megvalósítások operációs rendszer hívások és általában IP alapú protokollok segítségével kommunikálnak a tesztelendő rendszerrel.

Szintén a tesztportok feladata a TTCN-3 absztrakt, struktúrált üzeneteinek átviteli csatornán használt bitfolyammá alakítása (kódolása) és visszaalakítása (dekódolása).

#### 2.4. Segédprogramok

A TITAN-hoz tartozik még számos kis segédprogram, melyek a tesztírást, végrehajtást vagy az eredmények feldolgozását segítik. Található itt tesztfuttatást automatizáló szkript, naplófájlokat összefűző és formázó segédprogram stb.

#### 2.5. Teszt-vezérlő

Ha egy teszteset egyszerre több párhuzamosan futó tesztkomponenst használ, akkor ezek működését össze kell hangolni. A tesztrendszer központi koordinálását a futtatható tesztsorozattól független, TITAN-hoz tartozó program végzi. A tesztvezérlő program kapcsolatban áll a tesztrendszer összes komponensével, segítségével zajlik a tesztkomponensek létrehozása és leállítása valamint a közöttük lévő port kapcsolatok felépítése és bontása.

Teljesítmény-tesztelés esetén a tesztrendszer több számítógépre is eloszthető. Ilyenkor a gépek közötti terhelésmegosztás szintén a tesztvezérlő feladata. Hogy a tesztrendszerben ne legyen szűk keresztmetszet, a tesztvezérlő kizárólag koordinátori feladatokat lát el, üzenetek továbbításában és elemi teszteseményekben egyáltalán nem vesz részt. A TITAN elosztott teszt-architektúrájának részletes leírása [6]-ban olvasható.

A tesztvezérlő biztosítja a felhasználói felületet a tesztsorozat interaktív végrehajtása közben. A felhasználói felület lehet szöveges (parancssori) vagy grafikus. Ezen keresztül a felhasználó folyamatosan képet kap a tesztrendszer pillanatnyi állapotáról és a végrehajtott műveletekről, és szükség esetén be is avatkozhat: leállíthatja az éppen futó tesztesetet vagy egy új tesztesetet indíthat el.

## 3. A TITAN működése

#### 3.1. Szintaktikai és szemantikai ellenőrzés

A bemenő modulok szintaktikai elemzése a GNU flex és bison segédprogramokkal készített elemzők segítségével történik. Az elemzés során a bemenetből a fordító egy speciális memóriastruktúrát, úgynevezett ab-

sztrakt szintaxisfát épít, amely a további fordítási lépések alapjául szolgál. Az integrált ASN.1 és TTCN-3 fordítóprogram előnye, hogy a két nyelv definíciói egy közös és egységes szintaxisfába kerülhetnek. Ez megkönnyíti a TTCN-3 tesztek írását az ASN.1 leírással rendelkező protokollokhoz.

A szemantikai ellenőrzés feladata a hibás név szerinti hivatkozások, meg nem engedett műveletek, adattípus-ütközések és hasonló hibák kiszűrése. Az ellenőrző algoritmus egymenetes, tehát a szintaxisfát egyszer járja végig, de a bejárás sorrendjét a definíciók közötti hivatkozások befolyásolhatják. Szemantikai ellenőrzés során a legnagyobb kihívást a kétértelmű nyelvi elemek kezelése jelenti, ugyanis ASN.1-ben és TTCN-3-ban egyaránt előfordulnak olyan konstrukciók, melyeket a szintaktikai elemzés nem tud beazonosítani.

Ha a fordító akár szintaktikai, akár szemantikai hibát talál, az első hibaüzenet kiírása után nem áll meg, hanem folytatja tovább az ellenőrzést, hogy újabb hibákat derítsen fel. Az ellenőrző algoritmusokban hibaelfedést alkalmazunk, azaz egy létező, de hibás definícióra történő hivatkozás nem generál újabb hibaüzeneteket. Különben egyetlen egyszerű hiba is hibaüzenetek lavináját indítaná el.

### 3.2. Kódgenerálás

A C++ kód generálása szintén a szintaxisfa alapján történik, melyen a szemantikus ellenőrző már bizonyos egyszerűsítéseket végrehajtott, például a konstans aritmetikai kifejezéseket összevonta.

A generált kód legfőbb jellemzője, hogy statikusan tipizált, azaz minden TTCN-3 és ASN.1 adattípushoz külön C++ típus (osztály) tartozik. Ennek legfőbb előnye a futási sebességben jelentkezik, ugyanis a TTCN-3 adatértékek és mezők memóriabeli elhelyezését a C++ fordító automatikusan elvégzi.

További előny, hogy a TTCN-3 műveleteknél a típusegyezést a C++ fordító is ellenőrzi. Ez utóbbit a fordítóprogram azon régebbi változatainál használtuk ki, amelyek egyáltalán nem tartalmaztak TTCN-3 szemantikus ellenőrzést. A korai verziókban a C++ kódgenerálás a szintaktikai elemzéssel egyidejűleg történt, a szemantikai hibákra a C++ fordító hibaüzeneteiből lehetett következtetni.

A statikus típuskezelés egyetlen hátránya az összetett protokollok típusait leíró C++ osztályhierarchia nagy mérete és az emiatt megnövekedő fordítási idő. A típusok nagy kódméretét némileg ellensúlyozza, hogy a TTCN-3 értékekre, adatmintákra (templatekre) és viselkedés leírásokra (függvényekre, tesztesetekre stb.) tömör C++ kód generálható.

Dinamikus tipizálás esetén, melyet a legtöbb piacon kapható TTCN-3 eszköz alkalmaz, az adatértékeket a futtató környezet általános struktúrákból állítja össze, és minden műveletnél a típusegyezést is vizsgálni kell (például egy egész számokon végzett művelet egy általános adatstruktúrában kapja meg a paraméterét, és neki kell meggyőződnie arról, hogy tényleg egész számot kapott-e). E kiegészítő műveletek akár 10 vagy 100-szoros sebességcsökkenést is eredményezhetnek a TITAN-hoz viszonyítva.

### 3.3. Futtatható tesztsorozat előállítása

A teljes fordítási folyamat, így a tesztsorozat C++-ra fordítása, a generált C++ programkód és a tesztportok gépi kódra fordítása valamint a végrehajtható tesztsorozat összeszerkesztése, a make segédprogram használatával történik. A fordítási lépések közötti függőséget leíró Makefile-t a TITAN fordítóprogramja automatikusan elő tudja állítani a TTCN-3 és ASN.1 modulok, valamint a tesztportok ismeretében.

A gyakorlatban használt TTCN-3 tesztsorozatok általában sok modulból állnak. Megfigyelhető, hogy a tesztírási folyamat során a modulok nagy része (például egy protokoll üzeneteit leíró típusdefiníciókat tartalmazó modul) csak ritkán változik. Két fordítás és futtatás közötti változás általában csak néhány modult és ezen belül néhány programsort érint (leggyakrabban a teszteseteket leíró TTCN-3 utasítások változnak). Mivel egy teljes fordítás percekig, komplex tesztsorozatok esetén akár órákig is eltarthat, nem célszerű ezt a folyamatot minden apró módosítás után megismételni.

A TITAN fordítója és a make segédprogram együtt képes inkrementális fordítást végezni. Ez azt jelenti, hogy az előző fordítás részeredményeit felhasználva csak a változások által érintett modulokból generál C++ kódot, és csak a szükséges állományokat fordítja le újra. Az újrafordítandó modulok beazonosítása bizonyos esetekben nem egyszerű feladat.

Ha egy modul definícióit más modulok is használják, akkor az itt bekövetkező változások a használó modulokat is érinthetik, amelyek hatással lehetnek az őket használó modulokra, és így tovább. Ilyenkor előfordulhat, hogy egyetlen változás miatt modulok tucatját kell újrafordítani azért, hogy a végrehajtható tesztsorozat konzisztenciáját megőrizzük. Mindezek ellenére a gyakorlati példák azt igazolják, hogy a TITAN inkrementális fordítással jelentősen tudja csökkenteni a fordítási időt és ezáltal növelni a tesztsorozat-fejlesztés hatékonyságát.

### 3.4. Kódolás, dekódolás

A tesztfuttatás során a tesztelendő rendszernek küldött üzeneteket kódolni, míg az onnan fogadottakat értelmezni, dekódolni kell. Ennek megkönnyítésére a TITAN számos beépített kódolóval rendelkezik, melyek C++ nyelvű interfészen keresztül érhetők el. Így egy üzenet kódolása vagy dekódolása az üzenet bonyolultságától függetlenül néhány programsorban elvégezhető. Ezek a kódoló funkciók elhelyezhetők egy tesztportban vagy egy TTCN-3-ból hívható, de C++ nyelven megírt külső függvényben.

Az ASN.1-ben leírt üzeneteket a TITAN a BER szabályai szerint képes kódolni, a TTCN-3 adattípusokhoz kétféle, egy szöveges (TEXT) és egy táblázatos, bit alapú (RAW) kódolás létezik. TTCN-3 típusok esetén a kódolási szabályokat, melyek lehetnek egészen összetettek is, attribútumok segítségével lehet megadni.

### 3.5. Grafikus felület

A TITAN-hoz a tesztvezérlővel egybeépített grafikus felület is tartozik, mely a TTCN-3 tesztsorozatok fejlesztésénél és futtatásánál is segítséget nyújt a felhasználók számára.

A *2. ábrán* a grafikus felület fő ablakának képernyőképe látható. Bal oldalon találjuk a tesztsorozatot alkotó modulok, tesztportok és egyéb fájlok listáját. A jobb oldali ablakban pedig a fordítás menetét követhetjük nyomon, – most épp egy inkrementális fordítás kimenete látható.

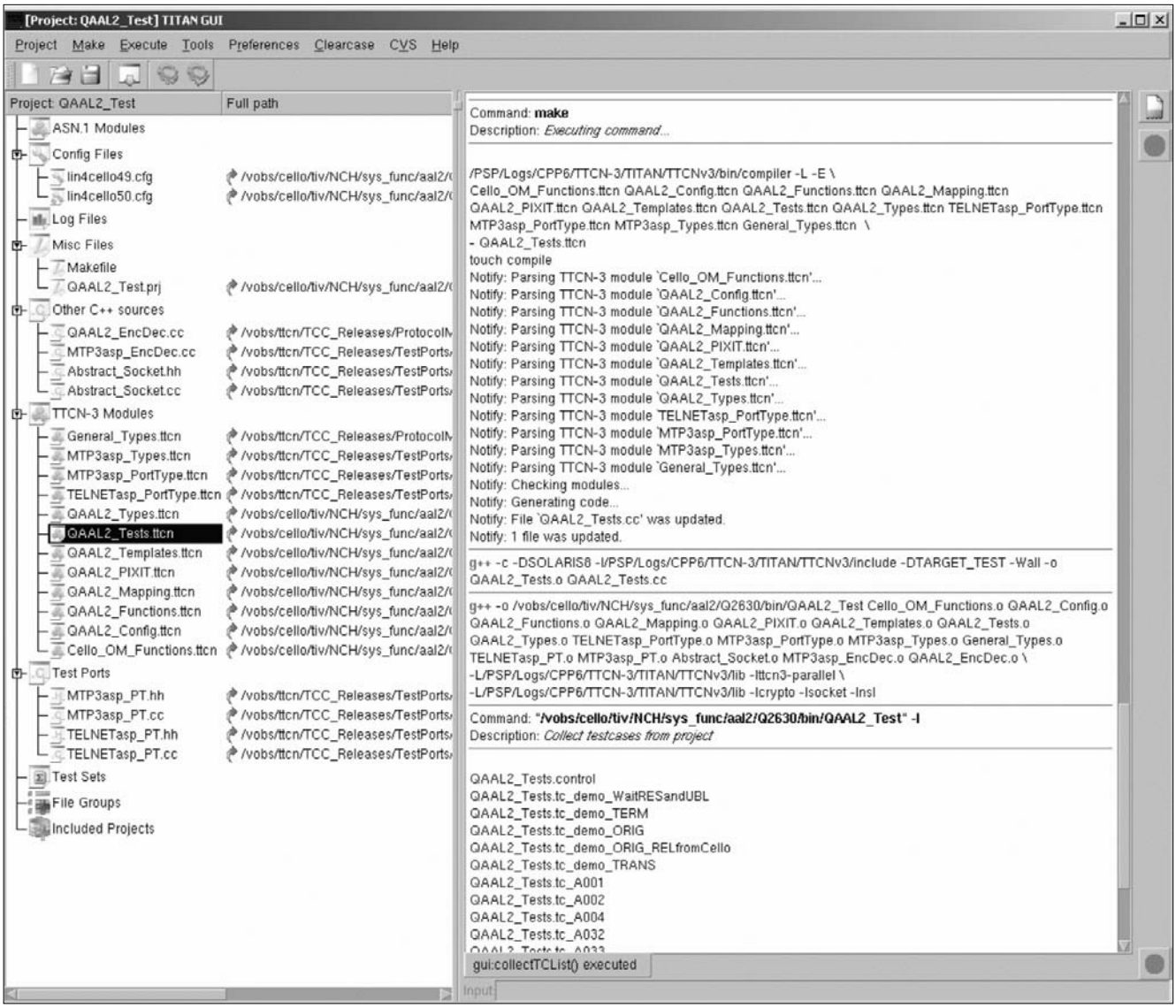
## 4. TTCN-3 interfészek

Egy futtatható TTCN-3 tesztsorozat a külvilágot interfészeken keresztül érheti el. Az ETSI két szabványban (TTCN-3 Runtime Interface, TRI [2] és TTCN-3 Control Interface, TCI [3]) hat ilyen interfészt definiált. A TRI két interfésze a tesztsorozatnak a tesztelendő rendszerrel illetve az operációs rendszerrel való kapcsolatát (pl. időzítés) írja le. A TCI négy interfészből áll: tesztmenedzsment (tesztesetek futtatása és a tesztsorozat paraméterezése), kódolás és dekódolás, tesztkomponensek kezelése és naplózás.

A szabványok programozási nyelvektől független adattípusok és eljárások segítségével írják le az interfészeket, és ezekhez C, JAVA és néhol XML nyelvű leképzést adnak. Az interfészek szabványosításának célja, hogy az ezeknek megfelelő TTCN-3 futtatókörnyezetek egymással csereszabatosak legyenek.

A TITAN a fenti szabványos interfészek közül jelenleg egyiket sem támogatja. Ennek több oka van: egyrészt amikor az interfész-szabványok 2002-ben és 2003-ban megjelentek, akkor a TITAN már egy kész, kiforrott rendszer volt a saját interfészeivel. Másrészt a fejlesztés során más szempontokat részesítettünk előnyben, mint a szabványosítók. Elsődleges célunk az volt, hogy egy teljes, működőképes és hatékony tesztrendszert állítsunk elő úgy, hogy a felhasználónak a lehető legkevesebb külső programmodult kelljen a rendszerhez hozzáfejlesztenie. Így a TITAN egyedül a tesztelendő

*2. ábra A grafikus felület*



rendszer felé biztosít programozói felületet, ez pedig a tesztport-interfész. A TRI és TCI összes többi funkcióját a TITAN beépített módon támogatja, ezekhez nem biztosít programozói hozzáférést.

A TITAN tesztport interfésze több szempontból is előnyösebb a TRI-nél. A tesztportok mindig egy TTCN-3 kommunikációs porthoz (és ezáltal egy protokollhoz) kapcsolódnak, így az egyidejűleg tesztelt többféle protokoll szétválasztásáról az interfész maga gondoskodik.

TRI használata esetén a tesztelendő rendszer felé irányuló összes üzenet egyetlen függvényhíváson és a felhasználó által írt, úgynevezett adapter modulon halad keresztül, és a szétválogatást itt kell megoldani. Ha a tesztelésbe egy újabb protokollt vagy rendszer-interfészt akarunk bevonni, akkor ez a TITAN-ban építőkocka elven egy új tesztport hozzáadásával könnyen megoldható, míg TRI esetén ugyanez az adapter újratervezését jelenti. A tesztport interfész elosztott teljesítménytesztek esetén is kedvezőbb, ugyanis a TTCN-3 párhuzamos teszt komponensekhez kapcsolódó tesztportok nem okoznak szűk keresztmetszetet.

A szabványos interfészek utólagos beépítése bizonyos esetekben nehézséget is okozhat, mert ezek a TITAN-nal ellentétben dinamikus tipizálást és többszálú működést feltételeznek. A TRI és TCI interfészei általában nem a hatékony működést preferálják, így ezek gyakorlati haszna is megkérdőjelezhető a TITAN-ban. Használatukkal többnyire nem lehetne a TITAN beépített funkcióihoz képest gyorsabb, egyszerűbb vagy kényelmesebb megoldásokat előállítani.

## 5. Összefoglalás

A TITAN fejlesztése és használata révén az Ericsson is bekapcsolódott az ETSI TTCN-3 szabványosító munkájába. Aktivitásunkat jól mutatja, hogy a nyelv szabványához eddig összesen beküldött 340 db módosító javaslat (Change Request) több mint fele (196 db) az Ericssontól származik. Ezek többsége a nyelv leírásában talált kétértelműségekre vagy ellentmondásokra ad feloldást, de számos olyan nyelvi kiterjesztést is javasoltunk, melyek a TTCN-3 használatát egyszerűbbé, kényelmesebbé teszik.

A TITAN 2003. óta hivatalosan elfogadott és támogatott teszt eszköz az Ericssonban. Azóta az Ericsson Magyarország Kft-ben egy közel 40 fős részleg foglalkozik a TITAN valamint a rá épülő TTCN-3 teszt megoldások (tesztportok, protokoll modulok és kész tesztsorozatok) fejlesztésével és karbantartásával. Megrendelőink az Ericsson termékfejlesztő részlegei a világ minden tájáról.

Bár a TITAN a cégen kívül piaci forgalomba nem került, így is jelentős és folyamatosan növekvő felhasználói táborra tett szert az elmúlt évek során.

Közel 50 tesztport és csaknem 100 protokoll modul készült eddig a TITAN-hoz, mely lehetőséget nyújt a távközlési rendszerek széles spektrumának teszteléséhez.

**Irodalom**

[1] ETSI ES 201 873-1 V3.1.1 (06/2005)
The Testing and Test Control Notation version 3.
Part1: Core Language
[2] ETSI ES 201 873-5 V3.1.1 (06/2005)
The Testing and Test Control Notation version 3.
Part5: TTCN-3 Runtime Interface (TRI)
[3] ETSI ES 201 873-6 V3.1.1 (06/2005)
The Testing and Test Control Notation version 3.
Part6: TTCN-3 Control Interface (TCI)
[4] Szabó János Zoltán:
A TTCN-3 tesztleíró nyelv,
Magyar Távközlés, XII. Évf., 2. szám, 2001. február
[5] János Zoltán Szabó:
Experiences of TTCN-3 Test Executor Development,
Testing of Communicating Systems XIV,
Application to Internet Technologies and Services,
Edited by Ina Schieferdecker, Hartmut König and
Adam Wolisz, Kluwer Academic Publishers, 2002.
[6] János Zoltán Szabó:
Performance Testing Architecture for
Communication Protocols,
Periodica Polytechnica, Electrical Engineering,
Budapest University of Technology and Economics,
2003. 47/1-2.

# Komponens rendszerek modell alapú tesztelése

BÁTORI GÁBOR, THEISZ ZOLTÁN

*Ericsson Magyarország, Software Engineering Group*
*{gabor.batori, zoltan.theisz}@ericsson.com*

**Lektorált**

***Kulcsszavak: szakterület specifikus modellezés, komponens alapú rendszerek, Deployment Tool***

*A távközlési szoftverek területén egyre növekvő igény tapasztalható az összetett rendszerek építése iránt, amely nehezen teljesíthető új fejlesztési paradigmák bevezetése nélkül. A szakterület specifikus modellezés újszerű megoldási módot kínál ahhoz, hogy megfelelő komplexitás menedzsmentet lehessen alkalmazni, valamint a modellezés során keletkező modelleknek a cikkben bemutatandó komponens alapú platformra (ErlCOM) történő átalakítása révén a végső rendszer összes képessége megvizsgálhatóvá válik. A módszer elterjedésével szükségessé válik a modell alapon készült szoftverek tesztelésére, de eddig még nem áll rendelkezésre egy általánosan elfogadott módszer erre nézve. A cikkben összefoglaljuk a szakterületen alkalmazható különböző módszereket, valamint bemutatjuk az általunk kifejlesztett Deployment Tool-t.*

## 1. Bevezetés

Napjainkban egyre több új programtervezési módszertan lát napvilágot. Ezeknek az új módszereknek a legfontosabb célja, hogy csökkentsék az egyre komplexebbé váló szoftverek előállításának költségeit, valamint újrafelhasználhatóságot lehetővé téve a korábban befektetett munkát hasznosítsák. Az egyik legdinamikusabban fejlődő, és legtöbbet kutatott, módszertan a modell alapú programtervezés. Több módszer is létezik a modell alapú tervezés megvalósítására, de a módszerek alapvető céljai a következőek:

- *Befektetések megőrzése,* hogy a szellemi és anyagi befektetésekre ne legyenek hatással a technológiákban bekövetkező változások.
- *Automatikus implementáció,* hogy növelni tudjuk a fejlesztés hatékonyságát azzal, hogy a modellből automatikusan előállítható a végső forráskód.
- *Tesztelés,* hogy a fejlesztés korai szakaszában is lehetőség nyíljon a modell ellenőrzésére, akár annak közvetlen futtatásával, továbbá, hogy a fordító programok által létrehozott alkalmazások minősége jobb és egyenletesebb legyen, mint a más módszerrel létrehozottaké.

Az Object Management Group (OMG) által támogatott Modell Vezérelt Architektúra (Model Driven Architecture, MDA) az egyike ezeknek a modell alapú módszereknek. Az MDA egyre elterjedtebbé válik komplex szoftverek fejlesztése során. Az újrafelhasználást úgy érhetjük el az MDA filozófia szerint, hogy szétválasztjuk az alkalmazást leíró funkcionális részeket egy adott platformhoz kötődő implementációs részletektől. Platform alatt nem csak egy adott hardware környezetet értünk, hanem ide tartoznak olyan technológiai kérdések is, mint például az adatábrázolás. Az MDA-ban alapvetően magas szintű modellekkel ábrázoljuk a különböző részeket. Az MDA az UML-t (Unified Modeling Language) használja a modellek ábrázolására, mely *de facto* szabvány az objektum-orientált tervezésben. Az UML egy univerzális modellező nyelv, így igen sokféle módot enged meg a tervezőnek az adott probléma ábrázolására.

Az MDA-ban történő szoftverfejlesztés során kétféle modell típus jelenik meg. Az egyik a platform független modell (Platform Independent Model, PIM), míg a másik a platform függő modell (Platform Specific Model, PSM). A PIM egy funkcionálisan teljes modellje a rendszernek, így alkalmas szimulációra valamint tesztelésre. A platformfüggő modell ezzel szemben azokat az információkat ábrázolja, ahogy egy általános probléma oldható meg egy adott platformon. A két modellből transzformáció segítségével megkaphatjuk az adott problémát megoldó, az adott platformhoz jól illeszkedő forráskódot. Sajnos az egyes szakterületek speciális problémáinak megoldásához nem mindig jól illeszkedik az UML nagyon általános struktúrája, ezért egy szakterületet átfogó rendszer specifikussága és az UML alapú fejlesztés általánossága közötti ellentmondás meta-programozási architektúra alkalmazásával oldható fel. Metamodellezés alkalmazásával szakterület-specifikus nyelveket készíthetünk, melyek meta-generátorok által támogatott fordító programokkal szakterület-specifikus platformokra képezhetőek le.

Egy ipari környezetben is többször alkalmazott metaprogramozható eszköz a GME (Generic Modeling Environment) [1], melyet a RUNES IST projekt [2] kapcsán elosztott hibatűrő komponens alapú rendszerek tervezésére használunk. A RUNES komponens rendszer reflexivitását és futási idejű újra konfigurálhatóságát kihasználva garantálható, hogy az aktuális rendszerkonfiguráció mindig eleget tegyen a metamodell szabályainak. A komponens rendszer és a modell alapú tervezés nyújtotta előnyöket felhasználva különböző tesztelési módszerek dolgozhatóak ki. A cikk összefoglalást nyújt a modellezés során alkalmazható ellenőrzési eljárásokról, és az általunk kifejlesztett új módszerről, mely segíti a komponens rendszerek helyességének ellenőrzését.

## 2. Modell alapú szoftverfejlesztés (szakterület-specifikus modellezés)

Mint a bevezetésben említettük, a modell-alapú szoftverfejlesztés újszerű módon közelíti meg a programfejlesztés kérdését, hiszen míg hagyományosan a legfontosabb területek a különböző programozási nyelvek és operációs környezetek és a segítségükkel előállított forráskód voltak, addig a modell-alapú programfejlesztés inkább a szoftverrendszer logikai működésének precíz leírására törekszik. A modellkészítés folyamata kap nagyobb szerepet, mivel az MDA az elkészült modellt végrehajthatónak tekinti, tehát a modell vagy interpretált környezetben vagy generatív technikát alkalmazó modell-interpreterek segítségével automatikusan futtathatóvá alakítható.

Mielőtt a folyamatot részleteiben is megvizsgálnánk, előbb egy-két alapfogalom bevezetése elengedhetetlen. A legfontosabb fogalom maga a modell és a hozzá tartozó metamodell. A modell írja le mindazt, amely a szoftverrendszer működését specifikálja, mind funkcionális kapcsolatrendszerében, mind ezek dinamikus viselkedésében. Az összes fontos tényezője a rendszernek meg kell, hogy jelenjen a modellben, hisz csak ily módon biztosítható, hogy később a modell automatikusan futtathatóvá váljék. Természetesen a modellben megjelenő elemek osztályozhatóak, és a közöttük fennálló kapcsolatok szabályok formájába önthetőek, más szóval élve a modell egy metamodellen alapszik, amely megadja a modellek absztrakt szintaktikáját és statikus szemantikáját. Az UML nyelv esetében az OMG szabványosította a megengedhető modellezési módszert, amely tetszőleges modellezési problémákat hivatott megoldani. Az általánosság egyben jó dolog, hiszen univerzális hatókörű, másrészt rossz, mert nem optimális egyik alkalmazási területen sem.

A szakterület-specifikus modellezés (Domain Specific Modeling, DSM) ezzel szemben azon az ötleten alapszik, hogy minden alkalmazási terület számára készít egy olyan specifikus metamodellt, amely az adott terület problematikáinak a legjobban megfelel, ezáltal lehetővé téve alkalmazás-családok gyors előállíthatóságát a metamodell és/vagy modell újrafelhasználása által. A létrehozott modellek absztrakciós szintjük szerint megfeleltethetőek az OMG PIM és PSM kategóriáinak, de lényükből fakadóan ezen besorolásnál rugalmasabbak. A szakterület-specifikus modellezés technikája megköveteli, hogy a metamodellek előállítása könnyű legyen, és hasonlóan az OMG által szabványosított MOF (Meta Object Facility) metamodellező nyelvhez, a DSM környezetnek is rendelkeznie kell egy hasonlóval. Továbbá a hatékony fejlesztés elengedhetetlenné teszi egy metamodell alapján modellek építését támogató szerkesztő program meglétét is. A bevezetésben megemlített GME egy mind kutatási, mind ipari környezetben jól alkalmazható DSM környezetet szolgáltat.

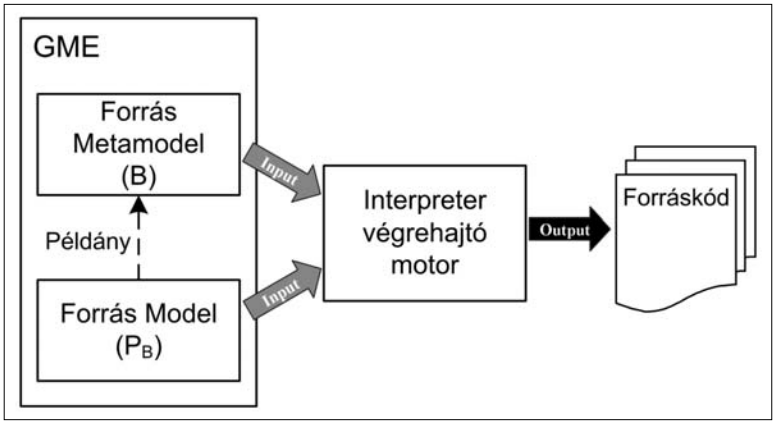
Miután az alapfogalmak bevezetésre kerültek, bemutatjuk a fejlesztés két alaplépését. Először azt vizsgáljuk meg, amikor a modellből generált kód valamely már meglévő programnyelv forrásszövegeként jelenik meg. Az *1. ábrán* látható, hogy a forrás metamodellt és modellt felhasználva a modell-interpreter metamodell mintákat felhasználva bejárja a modellt és a benne foglalt információk alapján előállítja a célnyelvű szöveget.

Fontos kiemelni, hogy míg a statikus szemantikát a metamodellen alapuló modell-szerkesztő ellenőrizte, addig a dinamikus viselkedés szemantikáját a modell-interpreter helyezi el a kódban.

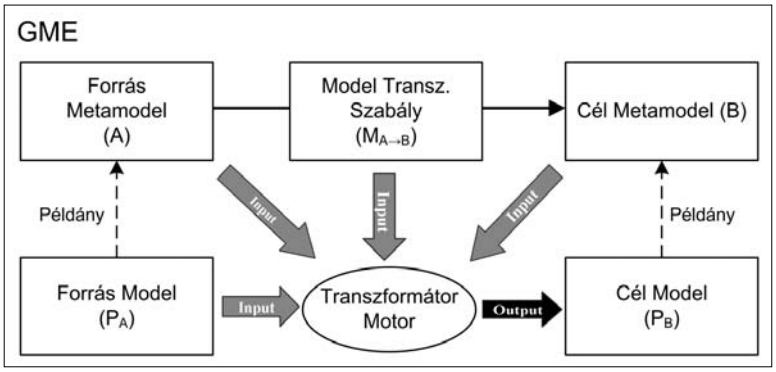
Természetesen maga a modell-interpreter is előállítható rekurzívan modell alapon és ily módon egy-két általános programozási mintát megvalósító véginterpretertől eltekintve a teljes folyamat modell-alapúvá alakítható. A második alapeset a *2. ábrán* látható. A különbség az előbbi esethez képest az, hogy itt most nemcsak a transzláció forrása, hanem a célja is metamodell, modell párosként reprezentálódik. A fordítás folyamatában a forrás-metamodell, a forrásmodell valamint a cél-metamodell ismeretében, továbbá a fordítási szabályok leírásának megadásával – amely az esetek többségében gráftranszformációs algebrát használ – a cél-modell automatikusan generálódik.

A fenti két alapeset természetesen kombinálható is oly módon, hogy végeredményében a szoftverfejlesztés gráftranszformációk sorozatokat lezáró modell-interpretációk rekurzív rendszerévé váljék.

*1. ábra Modellfordítás interpreterrel*




*2. ábra Modellfordítás gráftranszformációval*



## 3. Komponens alapú rendszerek

A komponens alapú technikák jelentős szerepet játszanak komplex elosztott rendszerek tervezésekor, mivel a komponensek létrehozásával a rendszer funkcionalitása könnyen „csomagolhatóvá” válik. A komponens rendszerek osztályozása is csomagolási tulajdonságuk alapján történhet.

Az általunk fejlesztett ErlCOM [3] és a RUNES [2] komponens rendszere is reflektív kauzális hierarchikus port alapú csomagolást biztosít a telepített rendszer működése során. A hierarchikusság arra utal, hogy a komponensek egymásba csomagolhatóak, a port alapúság azt jelenti, hogy a komponensek közötti kommunikáció kizárólag összekapcsolt kompatibilis interfészek segítségével történhet, reflektivitás a komponensek kapcsolatrendszerének futás közbeni lekérdezhetőségét foglalja magában, míg a kauzalitás fejezi ki azt, hogy ha a komponens kapcsolat-gráfon változtatunk, akkor a változtatás ténylegesen végre is hajtódik a futó komponensrendszeren. A tulajdonságok megvalósításáért a komponens futtató mag (Component Runtime Kernel, CRTK) felel.

A fent definiált komponensrendszer felfogható platform metamodellként, és egy adott telepített rendszer egy modelljeként. A modell-interpreter szerepét a CRTK veszi át a 4. fejezetben részletezendő semantic anchoring-nak megfelelően. Így gyakorlatilag a futtató komponensrendszert beleintegráltuk a modell alapú szoftverfejlesztés folyamatába, mint PSM-et. A különböző komponens rendszerbeli megvalósítást a megfelelő modell-interpreter alkalmazásával válthatjuk ki az alkalmazási területnek megfelelően. A különbség például az ErlCOM és a RUNES rendszer között abban jelenik meg, hogy míg az előbbi magas rendelkezésre állású, hibatűrő rendszerek építésére szolgál, addig az utóbbi főleg szenzorhálózatok esetén kerül majd alkalmazásra.

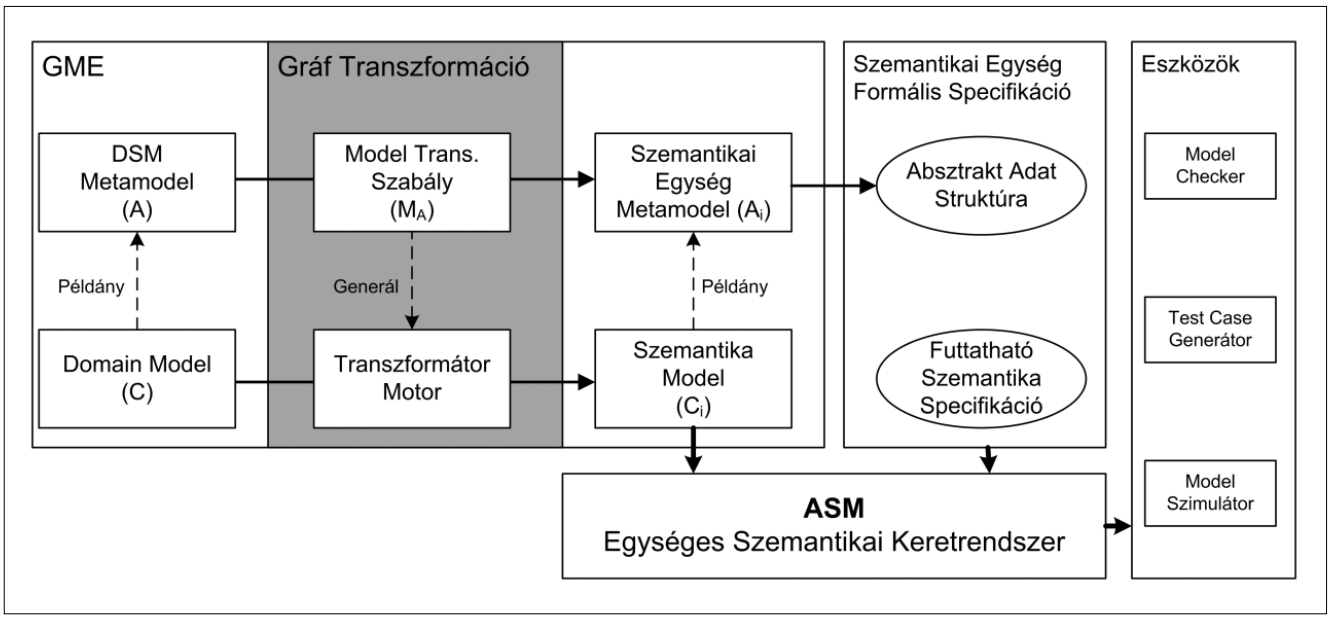
## 4. Tesztelés a modellalapú szoftverfejlesztésben

Metamodellezés során a megoldandó problémáknak először megpróbáljuk a statikus szerkezetét ábrázolni, azaz megtalálni azokat a strukturális építőköveket, melyekkel az adott problémák leírhatóak. A statikus elemek attribútumainak beállításával lehet az adott problémára jellemzőre alakítani egy adott strukturális elemet. Ám a statikus szerkezet leírása nem elegendő ahhoz, hogy a probléma dinamikáját is megfogalmazzuk. Ahogy az egyes szakterületek bevezetnek bizonyos fogalmakat, amiket az adott szakterület-specifikus modelleknek tükröznie kell, úgy az egyes szakterületek különböző dinamikát leíró modelleket is használnak. Erre azért van szükség, mert eltérő dinamika-leíró nyelvek alkalmasabbak különböző szakterületek problémáinak hatékonyabb, tömörebb leírására.

Azonban a szakterület specifikus dinamika-leíró nyelvek széles kategóriája kifejezhető az alapműködést leírók segítségével. Ilyen alapleírók például a következők lehetnek: véges állapotgép (Finite State Machine, FSM), időzített automata (Timed Automaton) vagy hibrid automata. Természetesen, hogy ezeket a nyelveket használhassuk a modellezés során, léteznie kell a nyelveknek megfelelő metamodelleknek a modellező eszközben.

A szakterület-specifikus modellek tesztelésénél komoly gondot okoz a modellező nyelvek diverzitása, ezért ha vizsgálatokat szeretnénk végezni a modelleken szükséges, hogy átalakítsuk egy olyan formára, mely megkönnyíti az ellenőrzések elvégzését. Ilyen megoldás lehet az úgynevezett semantic anchoring [4]. A semantic anchoring lényege *(3. ábra)*, hogy az egyes dinamikát leíró nyelveket megpróbáljuk leképezni egy matematikailag jól kezelhető közös nyelvre. Ez a közös nyelv a matematikailag bizonyítható Absztrakt Állapotgép (Abstract State Machine, ASM) [5], mely egy formális keret-

*3. ábra Eszköz-architektúra a szakterület-specifikus fejlesztéshez Semantic Anchoring-on keresztül*



rendszer szemantikai egységek specifikálásához. Az ASM-et sikeresen használták sok nyelv szemantikájának a leírására, így a C, a Java vagy az SDL (Specification and Description Language) specifikálására is.

Az ASM nyelvre épülve több teszteset-előállító és -bizonyító eljárás is létezik, így erre a nyelvre leképezve a szakterület-specifikus dinamika modellünket a már meglévő eszközöket felhasználva ellenőrizni tudjuk az általunk létrehozott dinamika leírásának a helyességét. Természetesen a leképzések nem triviálisak, de [4] bemutat egy gráftranszformáción alapuló eljárást. A gráftranszformációk nagy előnye, hogy nem csak a transzformációk kiinduló és végállomása formális modell, hanem magukat a transzformációs szabályokat is formális módszerekkel tudjuk megfogalmazni, és így a 2. fejezet alapján a transzformációs szabályok is modellnek tekintendők.

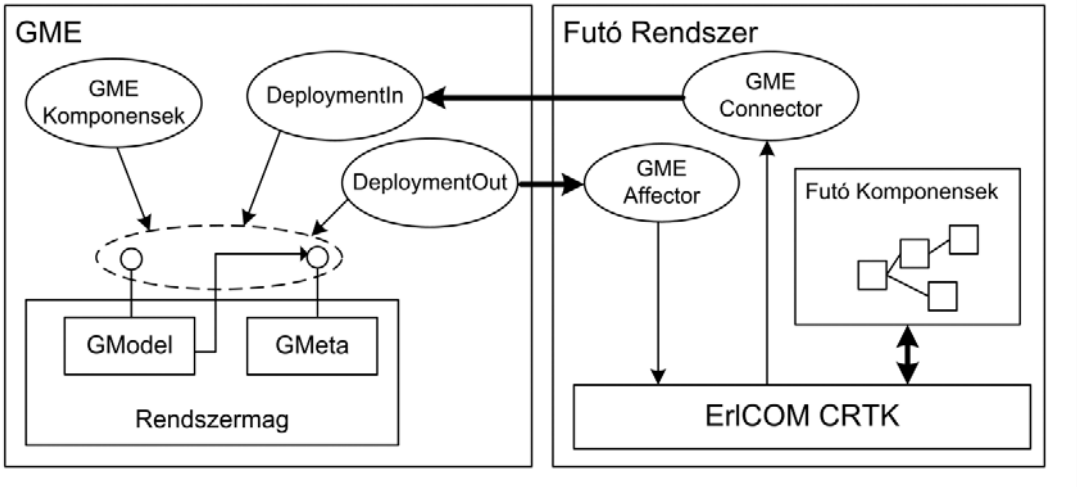
A semantic anchoring azonban nem mindig elegendő a teljes rendszer működésének a vizsgálatához, mivel a rendszer statikus szerkezete is kihatással lehet a végső implementáció működésére. Ilyen eset lehet az, amikor egy elosztott rendszer esetén a rendszer különböző elemeit rosszul helyezzük el az adott erőforrásokon. Ekkor a feleslegesen fellépő hálózati kommunikáció a különböző erőforrásokon elhelyezett elemek között lassíthatja az adatok feldolgozását, ami megnövekedett feldolgozási sorokat eredményezhet, melyek a rendszer használhatatlanságához vezethetnek. Bár az ASM nyelv alkalmas ilyen jellegű problémák leírására, azonban a leképzés igen bonyolulttá válhat, vagyis nem mindig ez a hatékony megoldás. Ilyen esetekben a megoldásnak az adódik, ha a forrásmodellünket leképezzük egy másik szakterület-specifikus nyelvre, amelyben a probléma egyszerűbben leírható, és hatékonyabban megoldható.

A fenti megoldás technikája természetesen nem csak matematikai módszerekkel történő kiszámítás lehet, hanem az adott feladat szimulációja is. A szimuláció során ellenőrizhetjük a rendszer működését, és az általunk legjobbnak ítélt megoldást választhatjuk ki. Egy hatékony módszer, amikor a forrásmodellt Matlab-ra képezzük le, amely köré sok, a modell különböző vetületét vizsgáló rendszer építhető. A Matlab-ra épülő szimulációs eszközök (Simulink, TrueTime) a rendszer magas szintű leírását teszik lehetővé, de a beépített matematikai modelljeik segítségével a modelleket „végre tudjuk hajtani”. A Simulink használatával lehetőségünk nyílik a modellünk korai állapotában való kipróbálására. A szimuláció eredményeit a kiindulási modellbe visszavezetve – általában manuálisan – precízebben tudjuk folytatni a modellezést. Mivel az így kapott modell jobban megfelel a valóságban elvárthoz, így a végső termék is megbízhatóbb minőségű lesz.

## 5. Tesztkörnyezet komponens-alapú rendszerekhez

A 3. fejezetben bemutatott komponens rendszer tesztelése során az előző fejezetben bemutatott tesztelési eljárásokon túl új módszerek is megjelennek, melyek a komponens rendszer specifikusságát használják fel. A komponens rendszer fő tulajdonsága, hogy reflektív, így működését meghatározza a metamodellje. A rendszer telepítése után csak a metamodellben adott általános szabályoknak megfelelően rendezheti át a szerkezetét.

Ezen a ponton nagyon nagy hasonlóság figyelhető meg a GME modell-szerkesztő és a futó komponens rendszer működése között. A GME-ben modellezés során csak a modellhez tartozó metamodell szintaktikai és statikus szemantikai szabályait betartva építhetünk modelleket, és ezeknek a szabályoknak a betartásáról a modellező eszköz állandóan gondoskodik. A komponens rendszer esetén a futó kód metamodell szabályainak betartására a komponens futtató mag ügyel. A fenti hasonlóságnak a következménye, hogyha a futó rendszer változásait visszavezetjük a kódot létrehozó modellező eszközbe, akkor magában a modellező eszközben nyomon lehetne követni a futó programban történő változásokat. Ez azt eredményezi, hogy ha a változásokról a modellező eszköz értesül, és megjeleníti őket, akkor egy adott pillanatban nem lehet eldönteni, hogy




*4. ábra*
*A deployment tool felépítése*

a modell-szerkesztőben megjelenő modellt a modellező személy vagy a futó program változásai hozták létre. Kihasználva ezt az ötletet, az általunk létrehozott úgynevezett deployment toollal megvalósítható az alkalmazás modell létrehozásának, transzformálásának, telepítésének és felügyeletének egyetlen eszközbe való dinamikus egyesítése.

Az előző oldali, *4. ábrán* ábrán látható, hogy a modell alapú szoftverfejlesztés teljes életciklusa egyetlen eszközbe a deployment tool-lal kiterjesztett GME-be egyesíthető. A deployment tool alkalmazásával megszűnik az az általános szemlélet, mely élesen elválasztja az alkalmazás létrehozásának és futásidejű karbantartásának a feladatát. Természetesen a két feladat egy eszközben egyesítése nem jelenti azt, hogy a két feladat során az eszközben tárolt elemeknek is ugyanúgy kell megjelennie. Sőt felhasználva GME nyújtotta lehetőséget különböző nézetek kialakítására, akár eltérő részletességgel, és grafikai megjelenítéssel is használhatjuk az eszközt az egyes feladatokra.

A deployment tool felépítése a 4. ábrán látható. A rendszer 4 fő egységen alapul. A *GME connector,* mely a futó komponens rendszerben történt változásokat a GME felé továbbítja. A *GME affector,* mely a GME-ben létrehozott változásokat a futó rendszeren érvényesíti. Valamint a GME-ben az előző két funkció fogadó egysége, a *DeploymentIn,* amely a futó rendszer változásait a modell-szerkesztőben megjeleníti, és a *DeplotmentOut,* amely a modell-szerkesztő által létrehozott változásokat a futó rendszernek elküldi.

Fontos megjegyezni, hogy a feladatok ezen csoportosítása által a GME-nek nem szükséges az adott rendszert futtató hardver környezetben futnia – az esetek többségében erre nem is lenne lehetőség –, hanem a rendszer változásai hálózaton keresztül is eljuthatnak a GME-hez. Így távoli felügyelő eszközként is használható a GME. Azonban, mivel a rendszer és a modell-szerkesztő egyszerre fut, így olyan változások is kiválthatóak, melyek ellentétes hatásúak, ilyenkor mindig a modell-szerkesztő által létrehozott változások a nagyobb prioritásúak.

Mivel a rendszer aktuális állapotát a modell-szerkesztő tárolja, így az adott modell elmentésével egy pillanatképet készíthetünk a rendszer konfigurációjáról, melyet különböző célokra lehet felhasználni. A pillanatkép legfontosabb felhasználási területe a rendszer tesztelése. A tesztelés során a teszter egy ilyen pillanatfelvételt kap a rendszerről, ahonnan elkezdheti a tesztelést, mivel a rendszer állapota az aktuális pillanatkép alapján visszaállítható.

A tesztelés tovább könnyíthető, ha a tesztelést nem valódi hardver platformon végezzük, hanem szimulált hardver környezetben. A szimulált platformnak előnyei, hogy nem kell drága speciális hardver erőforrással rendelkeznünk a rendszer futtatásához, hanem kiválthatjuk azt olcsóbb elemekkel, amelyeken a szimuláció azonosképp elvégezhető. Ezen kívül bizonyos szimulátorok egyéb előnyöket is nyújtanak, melyek jól párosíthatóak a modellezés előnyeivel. Például a Simics [6] szimulációs eszközt használva lehetőség nyílik a szimulált idő visszapörgetéséhez. Ezt a folyamatot szintén visszsza lehet vezetni a modell-szerkesztőbe, hogy így a modell szintjén is lehetőség nyíljon az időben előre- és hátralépésre.

## 6. Összefoglalás

A cikkünkben bemutattuk, hogy miképp nyújthat segítséget a szakterület specifikus modellezés összetett rendszerek fejlesztése során. A szakterület-specifikus modellek legjelentősebb hatása abban áll, hogy támogatást nyújt az adott terület szakértőinek, hogy egyértelműen és gyorsan tudják megfogalmazni az adott probléma megoldásait, és ezáltal sokkal hatékonyabbá teszi a szakterületen belüli fejlesztéseket. Több szakterületet átfogó rendszerek esetén a modellek egymásba történő automatikus átalakítása jelentősen támogatja a probléma egyre részletesebb szintű kidolgozását.

Mivel a teljes módszer formális alapokon nyugszik, így lehetőségünk nyílik különböző ellenőrzési módszerek bevezetésére. Sajnos egy általános, mindent átfogó ellenőrzési módszer nem létezik, de a rendszert különböző nézetek szerint vizsgáló módszerek jól integrálhatóak egymásba. A különböző ellenőrzési módszerek együttes alkalmazása során a végső termék sokkal megbízhatóbbá válik.

**Irodalom**

[1] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty, “Model-Integrated Development of Embedded Software”, Proceedings of the IEEE, Vol. 91, pp.145–164, January 2003.
[2] RUNES IST Project, http://www.ist-runes.org/
[3] G. Bátori, Z. Theisz, D. Asztalos, “Robust Reconfigurable Erlang Component System”, Erlang User Conference, Stockholm, Sweden, 2005.
[4] K. Chen, J. Sztipanovits, S. Abdelwalhed, E. Jackson, “Semantic Anchoring with Model Transformations”, ECMDA-FA 2005, LNCS 3748, pp.115–129.
[5] E. Boerger, R. Staerk, Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, 2003.
[6] Peter S. Magnusson et al, “Simics: A Full System Simulation Platform”, IEEE Computer, February 2002.

# Szemantikus protokollt alkalmazó mobil Peer-to-Peer kliensszoftver

FORSTNER BERTALAN, KELÉNYI IMRE

*BME Automatizálási és Alkalmazott Informatikai Tanszék*
*{forstner.bertalan, kelenyi.imre}@aut.bme.hu*

***Kulcsszavak: szemantikus Peer-to-Peer információ-visszakeresés, mobil szoftverfejlesztés***

*A korszerű okostelefonok egyre inkább hasonlítanak a kisméretű számítógépekre. Felhasználóik már nemcsak telefonálni szeretnének segítségükkel, hanem zenehallgatásra, fényképek készítésére és megosztására, információk olvasására is zsebkészüléküket használják. A szélessávú vezeték nélküli távközlés elterjedésével olyan hálózati alkalmazások költöznek át mobil készülékekre, amelyek eddig csak asztali környezetben voltak megszokottak. Sokoldalú felhasználási lehetőségei miatt egy ilyen nagyon lényeges szoftver lehet a teljesen elosztott hálózatra épülő információmegosztó alkalmazás, vagyis a Peer-to-Peer (P2P) szoftver. A fajlagosan magasabb távközlési költségek ugyanakkor azt igénylik, hogy az elosztott információ-visszakeresés során fejlett, kisebb adatforgalmat generáló, ugyanakkor a kliensek hálózati fluktuációját toleráló protokollt használjunk. Ebben a cikkben bemutatjuk az első, Symbian mobil operációs rendszerre fejlesztett Peer-to-Peer kliensszoftvert, valamint az adott célra kifejlesztett szemantikus protokollt.*

## 1. Bevezetés

Az információk, különböző fájlok, dokumentumok felkutatása a kezdetektől fogva nagy kihívása az informatika tudományának. A hálózatba kötött számítógépek és más eszközök (például mobiltelefonok) számának növekedésével a feladat csak tovább bonyolódott. A központosított megoldások mellett hamar kialakultak a teljesen elosztott információ-visszakereső alkalmazások, a különböző Peer-to-Peer rendszerek. Ezek lényege, hogy az egyes résztvevők azonos szerepkörrel vesznek részt a kommunikációban. Tipikus példa lehet egy kép- vagy zenemegosztó szoftver, ahol az egyes felhasználók egymás számára elérhetővé teszik saját tartalmaikat. A Peer-to-Peer alkalmazás összekapcsolja ezeket a felhasználókat és irányítja a fájlok keresését.

Az okostelefonok, vagyis smartphone-ok lehetővé teszik, hogy az említett fájljainkat (képek, zenék) magunkkal vigyük. Ezek a készülékek ugyanis – egy megszokott mobiltelefontól eltérően – fejlett funkcionalitással, komoly számítási teljesítménnyel, nagyobb háttértár kapacitással és viszonylag nagy kijelzővel rendelkeznek. Fejlesztői szempontból azonban még fontosabb, hogy a rendelkezésre álló környezet esetén komplexebb saját programok futtatására is alkalmasak lehetnek. A kizárólag a Java mobilra optimalizált verziójának futtatókörnyezetét (J2ME [1]) tartalmazó készülékektől eltérően az okostelefonok komolyabb operációs rendszerén natív alkalmazásokat is képesek futtatni. Néhány ismertebb smartphone operációs rendszer (Microsoft Windows CE [2], Linux egyes mobil disztribúciói [3]) előtt magasan őrzi piacvezető szerepét [4] a Symbian rendszere [5].

A Symbian céget 1998 júniusában alapította a Nokia, a Motorola, a Psion, valamint az Ericsson, és 1999-ben csatlakozott hozzájuk a Matsushita, majd 2002-ben a Sony-Ericsson és a Siemens. Ezen cégek célja egy mobil eszközökre szánt és a mobil környezetből adódó összes körülményt és követelményt figyelembe vevő operációs rendszer megalkotása volt. A kezdeményezés sikerét jelzi az egyre újabb verziókkal megjelenő operációs rendszer töretlen népszerűsége mind a készülékgyártók, mind a felhasználók körében. Legfőbb hátrányaként talán a konkurens technológiákhoz képest bonyolultabb programozói felületet lehet felhozni. Mindemellett mégis ez a platform tűnt a legcélszerűbbnek arra, hogy egy olyan összetettebb alkalmazást, mint a Peer-to-Peer kliens, mobil környezetbe ültessünk.

Önmagukban az okostelefonok leginkább a különböző típusú tartalmak tetszőleges helyen történő megjelenítésére, lejátszására nyújtanak a felhasználó számára csábító lehetőséget. A technológia egy másik irányának fejlődése, vagyis a mobil szélessávú hálózatok elterjedése teszi azonban lehetővé, hogy ezeket az információkat az asztali számítógépes környezetben megszokott módon osszunk meg másokkal. Miután a le- és feltöltési sebesség már nem okoz gondot, érdemessé vált elgondolkodnunk a Peer-to-Peer alkalmazások mobil környezetbe ültetéséről.

A kérdés vizsgálata során számos olyan körülményre bukkantunk, amelyek az eredeti fájlmegosztó protokollok hatékonyabb változatainak kifejlesztését tették szükségessé. Mivel a mobil kommunikáció költségei magasabbak a vezetékes kommunikációénál, a keresés adatforgalmát minimalizálni kell. A korszerű protokollok szemantikus úton próbálják a találati arányt növelni, az eredetileg véletlenszerű Peer-to-Peer hálózatot különböző megfontolások alapján strukturálni. Ezek a megfontolások azonban nem feltétlenül életképesek mobil környezetben, hiszen a mobil felhasználók fluktuációja viszonylag nagy, vagyis az egyes kliensek gyakran csatlakoznak, illetve hagyják el a hálózatot. Ennek oka nemcsak a hálózati forgalom, és így a költségek, illetve az

energiafogyasztás csökkentésének igényére vezethető vissza, hanem a mobiltelefonok sajátosságaiból is adódhatnak (például térerő hiánya, akkumulátor lemerülése stb.) Emiatt egy olyan szemantikus hálózati réteget dolgoztunk ki, amely egy adott csomópont (vagyis mobiltelefon) társhálózati kapcsolatait a lehető leggyorsabban igyekszik a felhasználó érdeklődési területei szerint átalakítani, és amely számára egy-egy csomópont kiesése a többi kliens keresési hatékonyságát, illetve eredményességét tekintve nem kritikus.

## 2. A Gnutella bemutatása

A Peer-to-Peer egyáltalán nem nevezhető új technológiának, hiszen a kezdetekben maga az Internetet is egy olyan decentralizált, Peer-to-Peer elven működő hálózatnak indult, melynek célja a számítási teljesítmény megosztása volt. Ez idővel, a fejlesztések hatására, fokozatosan megváltozott és eltolódott a kliens/szerver architektúra irányába, azonban a mai tendenciák azt mutatják, hogy kezdenek visszatérni az eredeti elképzelésekhez.

A P2P hálózatoknak számos felhasználási módja ismert, ilyenek többek között az információmegosztás és az információ visszakeresése. Az ilyen típusú hálózati protokollok közül az egyik legkiforrottabb a Gnutella. A protokollra épülő hálózatok célja a résztvevők erőforrásainak megosztása, amely gyakorlatilag bármi lehet (például kriptográfiai kulcsok), azonban a továbbiakban csak fájlok megosztásával foglalkozunk.

Fontos megjegyezni, hogy a modern Gnutella verziók már nem tisztán P2P rendszerek, a hálózatban a centralizált és a decentralizált topológia egyaránt megfigyelhető: bizonyos csomópontok kiemelt szerepkörrel rendelkeznek. Az ilyen csomópontok kinevezése azonban továbbra is felsőbb hatóság irányítása nélkül történik, így megmarad a függetlenség. A hálózat biztonsága szempontjából is nagyon előnyös a decentralizált topológia, hisz nem lehet egy, a központi szerver iránt intézett, támadással lebénítani a hálózatot. Továbbá fontos kiemelni, hogy a Gnutella felépítése moduláris: könnyen illeszthetők a protokollhoz kiegészítések, ily módon a szemantikus réteget támogató funkciókat is be tudtuk építeni a rendszerbe.

A Gnutella kliens első feladata, hogy további Gnutella csomópontokat találjon, majd ezekkel kommunikálva, felépítse a kapcsolatot a hálózathoz. Minden csomópont további csomópontokkal áll kapcsolatban (ezek száma általában 2-3), és így a kliens rajtuk keresztül éri el a hálózat többi részét. A Gnutella hálózat forgalma legnagyobb részt keresési kérésekből és válaszokból, illetve hálózati adminisztrációs üzenetekből áll.

Az üzenetek továbbküldése (útvonalválasztás) egy igen egyszerű sémára épül. A Gnutella hálózaton a serventeknek (servent – server and client, egy Gnutella csomópont) nincsen „címe” (ennek, ebben a kontextusban, nem is lenne értelme), így egy üzenet célba juttatásának egyetlen módja valamilyen elárasztásos módszer. Amikor egy servent megkap egy üzenetet, akkor feldolgozza annak tartalmát, majd továbbküldi a szomszédos csomópontoknak. Minden üzenet fejléce tartalmaz egy értéket (TTL – Time To Live), amely meghatározza, hogy az üzenet „milyen messzire” kerülhet a küldőjétől. Minden alkalommal, mikor egy servent továbbküldi az üzenetet, csökkenti annak TTL értékét, így szabályozható, hogy az üzenetek a hálózat milyen mélységéig jussanak el. Ha a TTL eléri a 0-át, a servent nem küldi tovább az üzenetet. Ez különösen fontos lehet például egy keresési kérésnél, ahol minél több csomópont kapja meg az üzenetet, annál több válaszra számíthatunk. Természetesen nagyobb TTL esetén a hálózat terheltsége jelentősen megnőhet, így a legtöbb kliens maximum 7-8 értékű TTL-el küldi az üzeneteket.

## 3. A Symella tervei

A mobil készülékek alacsony számítási teljesítménye és korlátozott háttértár-kapacitása egészen eddig nem tette lehetővé egy teljes értékű P2P kliens megvalósítását, ám az okostelefonok elterjedésével a helyzet megváltozott. Elsődleges célunk egy olyan hordozható készülékeken futó Gnutella kliens megalkotása volt, mely alkalmazkodik a mobil eszközök sajátosságaihoz, bármilyen más platformon futó Gnutella klienssel képes kapcsolatok kiépítésére, továbbá modularitása révén könnyen bővíthető: az új kutatási eredmények, implementálás után, a gyakorlatban is tesztelhetőek.

A kliens alapjául a jelenleg elérhető legkiforrottabb és egyben legelterjedtebb mobil operációs rendszert, a már említett Symbian OS-t választottuk, az alkalmazás ezért kapta a Symella nevet (a Symbian és a Gnutella szavak összekapcsolásából).

Bár a szoftver implementálása előtt már számos PC-s Gnutella kliens is a rendelkezésünkre állt, ezek a verziók egészen más szempontokat vettek figyelembe, mint amikre egy mobil alkalmazás készítésekor figyelni kell, így a legtöbb problémára új megoldást kellett találnunk. A kliens képes egy fájlt egyszerre több szálon is letölteni, mely jelentősen felgyorsítja az adatcserét, hisz a sávszélesség megoszlik a csomópontok között.

## 4. A SemPeer protokoll

A bevezetésben említett módon a Gnutella protokollra egy szemantikus réteget terveztünk, amelyet SemPeernek neveztünk el. Ennek a rétegnek a működését egy, a mindennapi életben megfigyelt jelenségből lehet a legkönnyebben megérteni. Eszerint a valós világban az emberek kapcsolatai különböző szálak mentén szövődnek [6]. Ha például valakinek a munkája a francia irodalommal kapcsolatos, akkor szakmai kapcsolatai is ilyen érdeklődésű emberek köréből alakul ki, hiszen a témával kapcsolatos kérdéseit leginkább ezek a kollégái tudják megválaszolni, nem pedig véletlenszerűen választott emberek. Hasonló megfontolással, ha valaki

a 19. századi klasszikus zenét kedveli, akkor – egy fájlmegosztó hálózatot tekintve – hasonló fájlokkal rendelkező felhasználók gyűjteményében tud a legnagyobb valószínűséggel ilyen zenét fellelni.

Az új szemantikus réteg tehát ez alapján, oly módon igyekszik egy adott csomópont kapcsolatait átalakítani, hogy az egyes szomszédok a szóban forgó csomópont különböző érdeklődési területei közül valamelyikkel rendelkezzenek. Megvalósított esettanulmányunkban a zenei stílusok képviselik ezt a fajta megkülönböztető dimenziót.

Az érdeklődési területek összehasonlítása a csomópontok által tárolt adatok alapján felépített [7,8] szemantikus profilok egybevetésével zajlik [9].

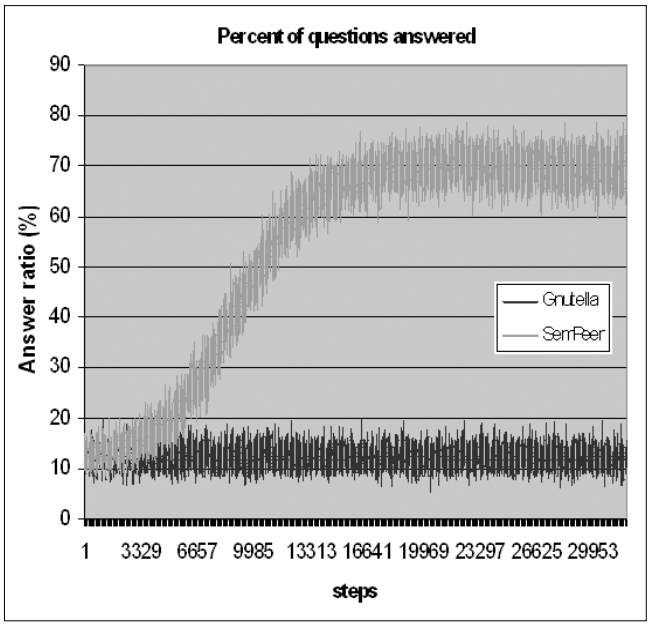
A protokoll mohó olyan értelemben, hogy a nagyobb hasonlóságot mutató kapcsolatokat részesíti előnyben, vagyis véges számú szemantikus kapcsolatai közül a kisebb hasonlóságú csomópontokat lecseréli a keresés során talált, nagyobb hasonlóságot felmutató csomópontokra. Mindehhez egyre mélyebb, konkrétabb szinten végzi az összehasonlítást. Ezáltal az idők folyamán egyre előnyösebb szomszédos csomópontokat képes kiválasztani, amellett, hogy már a kezdeti stádiumban is képes a véletlenszerű kapcsolatoknál jobb csomópontokat találni.

A protokoll az alap Gnutellához hasonlóan továbbra is strukturálatlan hálózatot alakít ki abban az értelemben, hogy a csomópontok a kereséshez, információ szerzéséhez lokális információkat használnak csak fel, így egy csomópont kiesése (ami mobil környezetben gyakorinak számít) nem okoz fennakadást a továbbra is megosztott adatok felkutatásában, elérhetőségében.

Egy további nehézség, amellyel a SemPeer protokoll kialakítása közben meg kellett küzdeni, a csoportképződés volt. Az algoritmus mohósága miatt ugyanis gyakran kialakulhatott az a helyzet, hogy egy keresőkérdés olyan csomóponthoz jutott vissza, amelyik már feldolgozta azt, hiszen az azonos érdeklődésű szomszédok kapcsolatai is logikusan az effajta profilú kliensek közül kerültek ki.

Ez nemcsak fölösleges hálózati forgalmat jelentett, hanem az egy keresőkérdés által elért csomópontok számának jelentős esését, így a keresett adat fellelési esélyének jelentős csökkenését is okozta. Ezt a hátrányt előnnyé kovácsolva ezért olyan mechanizmust is a protokollba kellett építenünk, amely a szemantikus réteg megfelelő, csoportképződést messzemenőkig támogató topológiáját biztosította [10].

*1. ábra*
*A Gnutella és a SemPeer protokoll találati arányainak összehasonlítása*




## 5. A protokoll teljesítménye

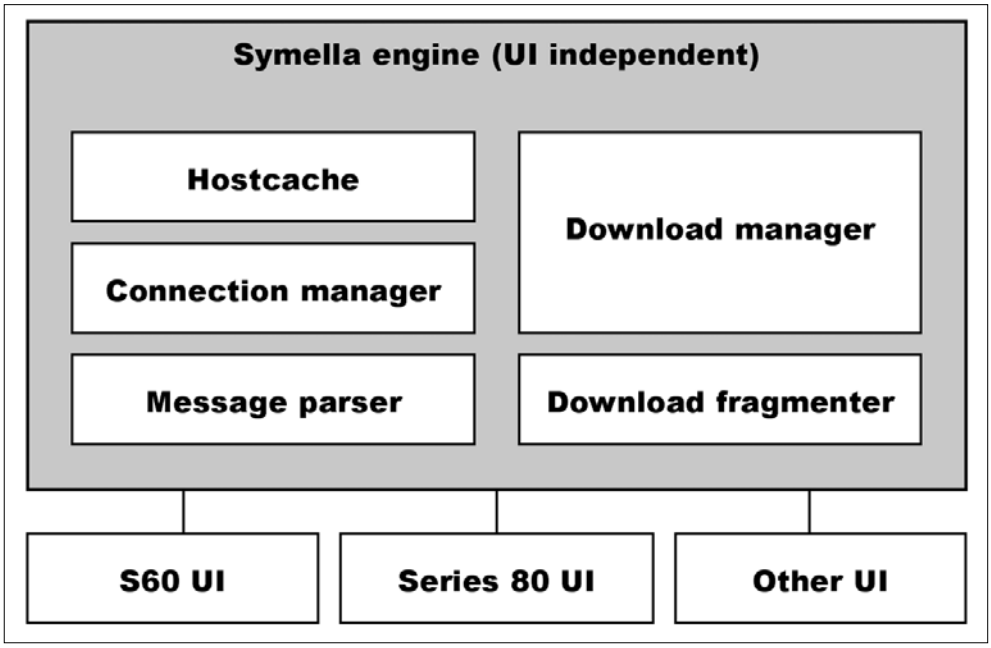
Ahhoz, hogy a protokoll teljesítményét megvizsgáljuk, egy szimulációs környezethez [11] illesztettük a saját protokoll-bővítményünket. A méréseket különböző tipikus paraméterekkel végezve megállapíthattuk, hogy milyen esetekben számíthatunk jelentős teljesítményemelkedésre a protokollunk alkalmazásától.

Érdekességképpen vizsgáljunk meg egy esetet, amikor a kliensek erősen egy témakörre koncentrált érdeklődési területtel rendelkeznek, és viszonylag kevés fájlt képesek tárolni. Ez tipikusan a kis memóriakapacitású okos mobil eszközök esete, ahol még egy albumnyi zenei anyag sem fér el a memóriakártyán. A szimulált hálózatban 16.000 csomópont található 4-7 kapcsolattal, amelyek átlagosan 5 fájlt képesek tárolni a saját zenei ízlésükből. Tíz főbb zenei ízlést különböztettünk meg. Meglepően jó eredményt kapunk, amennyiben egy csomópont a saját érdeklődési területének megfelelő zenéket keres: átlag 5-15 keresőkérdés eredményének visszaérkezésével a szemantikus protokoll olyan réteget hozott létre a véletlenszerű kapcsolódású alap Peer-to-Peer hálózat fölött, amelyben a találati arány az alap protokoll hétszeresét is elérte (0.11-ről 0.69-re emelkedett) *(1. ábra)*.

Természetesen amennyiben a keresőkérdések, vagy a tárolt dokumentumok tematikája nem ennyire homogén, akkor az átlagos találati arány ennél csak kisebb mértékben növekszik, azonban a teljesítménynövekedés minden esetben megfigyelhető.

Érdemes még megemlítenünk, hogy a szemantikus protokoll által elérhető elméleti maximális találati arány kiszámítására egy matematikai modellt alkottunk. A SemPeer protokoll finomítása során elértük, hogy ezt az elméleti határt minél inkább megközelítsük.

A [12] referencia alatt található cikkünkben összevetettük a modell által jelzett találati arányt a szimulált eredményekkel különböző helyzetekben (különböző érdeklődési területtel rendelkező csomópontok, érdeklődési körbe nem tartozó tárolt dokumentumokkal, keresőkérdésekkel stb.) Azt tapasztaltuk, hogy a kialakított protokoll a dokumentumok rendelkezésre állása esetén képes volt minimális eltéréssel megközelíteni az ideális értéket.




*2. ábra A Symella architektúrája*

## 6. A Symella architektúrája

A következőkben bemutatjuk azt az alkalmazás-arhitektúrát, amely a fent részletezett felismert követelményeknek és a Symbian operációs rendszer elveinek leginkább megfelel *(2. ábra)*. A Symella motorja felelős a kapcsolatok kiépítéséért és karbantartásáért, a keresési kérések feldolgozásáért, illetve a letöltések felügyeletéért. A Gnutella csomópontok adatait egy intelligens Host cache tárolja, mely folyamatosan gyűjti és osztályozza a különböző forrásokból beérkező címeket, hogy a Connection manager mindig a legoptimálisabb kapcsolatokat tudja kiépíteni. Az osztályozás alapja a kapcsolatok kiépítéséhez szükséges idő, ezen kívül a hosszú ideje aktív, megbízható kapcsolatokat külön eltárolja a rendszer. Az üzenetek feldolgozásáért és küldéséért a Message parser felelős.

A letöltésnél a sávszélesség minél jobb kihasználása kiemelt fontosságú szempont volt a tervezésénél, ezért is döntöttünk a többszálú letöltés támogatása mellett. Ennek lényege, hogy a letöltés elkezdése előtt a fájlt logikailag több apró darabra bontjuk, ezt végzi el a Download fragmenter (daraboló) modul, majd ezeket a darabokat több különböző csomóponttól, egymással párhuzamosan töltjük le. A letöltési alrendszer másik feladata a keresési találatok begyűjtése és a megegyező fájlokra hivatkozó válaszok összevonása.

A Symbian csak alapját képezi a mai készülékek operációs rendszerének. Erre ráépül még a felhasználói felület (User Interface, UI) rétege, amely készüléktípusonként nagyon különböző lehet. Egy széles képernyővel és teljes QWERTY billentyűzettel rendelkező Nokia Communicator felhasználói felületét teljesen máshogyan kell programozni, mint mondjuk egy Sony Ericsson P900-ét. Mivel egyik rendszer mellett sem szerettük volna elkötelezi magunkat, így a fejlesztés folyamán fontos szempont volt a platformfüggő és -független részek szétválasztása. Az alkalmazás jelenleg két UI-t is támogat: a legelterjedtebb S60-at, melyet a hagyományos mobiltelefon-billentyűzettel és kis képernyővel rendelkező készülékek használnak, illetve a Series 80-at, mely szabványa teljes alfanumerikus billentyűzetet és nagyobb kijelzőt ír elő.

A megfelelő architektúrának köszönhetően a mobil szoftver válaszideje igen jó lett. A *3. ábrán* összehasonlítottuk a Symella fontosabb válaszidejeit az elterjedtebb rendszerek megfelelő adataival.

## 7. Összefoglalás

Az általunk tervezett és implementált szoftver az első Symbian operációs rendszer alatt futó, natív nyelven írt Peer-to-Peer fájlcserélő alkalmazás. Jelenlegi formájában már most több ezren használják szerte a világon, a regisztrált letöltések száma pedig folyamatosan növekszik.

A tartalomszolgáltatók szemszögéből nézve, egy decentralizált, elosztott módon működő információcserélő rendszer sok lehetséges jövőbeli felhasználási módot rejt. Az adatok szabadon tárolhatók asztali, illetve mobil eszközökön, a mobil kliens az egységes protokoll ré-

*3. ábra A Symella és néhány ismertebb Gnutella kliens válaszidejének összehasonlítása*

| | Symella | BearShare | LimeWire | Gnucleus |
|---|---|---|---|---|
| Első induláskor a kapcsolódásig eltelt idő | 10 mp | 25 mp | 7 mp | 79 mp |
| További indulásokkor ugyanez | 3 mp | 7 mp | 3 mp | 15 mp |
| Első keresésre a találatok beérkezési ideje | 45 mp | 35 mp | 30 mp | 29 mp |
| További keresésekre ugyanez | 40 mp | 26 mp | 30 mp | 30 mp |

vén képes bármelyikkel kapcsolatba lépni. A mobil készülékek erőforrás-kapacitása jelentősen kisebb, mint egy asztali számítógépé, de sokkal nagyobb számban állnak rendelkezésre, így a redundancia révén nagy megbízhatóságú hálózatok építhetők ki belőlük.

A kliens jelenleg is megbízhatóan, és a mobilkészülékek képességeihez mérten viszonylag gyorsan működik, továbbfejlesztési lehetőségekben azonban nincsen hiány. A Gnutella szabvány számos olyan funkciót tartalmaz, amelyek implementálása tovább növelheti a kliens működésének hatékonyságát. Ilyen például az üzenetek tartalmának tömörítése, amely így jóval kisebb üzenetméretet és nagyobb szabad sávszélességet eredményez. A szemantikus réteg továbbfejlesztésével pedig nagyságrenddel csökkenthetjük a keresések lefutási idejét.

Az alapszoftver forráskódját a GNU GPL licenc keretében bárki szabadon használhatja [13]. A kutatási eredmények összefoglalása után a SemPeer protokollkiegészítés szintén publikusan elérhető lesz.

**Köszönetnyilvánítás**

A cikkben bemutatott projekt egyes részei a Mobil Innovációs Központ támogatásával valósultak meg.

**Irodalom**

[1] J2ME hivatalos weboldal,
http://java.sun.com/javame/index.jsp
[2] Windows CE hivatalos weboldal,
http://msdn.microsoft.com/embedded/
windowsce/default.aspx
[3] Montavista Mobilinux weboldal,
http://www.mvista.com/products/ mobilinux/
[4] Gartner Research
(Ben Wood, Carolina Milanesi, Ann Liang, Hugues J. De La Vergne, Tuong Huy Nguyen, Kobita Desai, Sauk-Hun Song, Nahoko Mitsuyama):
Market Share: Mobile Terminals,
Worldwide, 1Q05 (2005. május 24.)
[5] Symbian hivatalos weboldala,
http://www.symbian.com
[6] Mark S. Granovetter,
"The Strength of Weak Ties",
American Journal of Sociology, 78 (1973),
pp.1360–1380.
[7] H. Assadi,
"Construction of a Regional Ontology from Text and Its Use within a Documentary System International Conference on Formal Ontology and Information Systems", FOIS-98,
IOS Press, Amsterdam (WebDB-2000),
Springer-Verlag, Berlin, 2000,
pp.60–71.
[8] J.-U. Kietz, A. Maedche and R. Volz,
"Semi-Automatic Ontology Acquisition from a Corporate Intranet",
Proc. Learning Language in Logic Workshop (LLL),
ACL, New Brunswick, N.J., 2000,
pp.31–43.
[9] Resnik, P.,
"Semantic similarity in taxonomy:
An information-based measure and its application problems of ambiguity in natural language",
Journal of Artificial Intelligence Research, 11 (1999),
pp.95–130.
[10] B. Forstner, R. Kereskényi, Dr. H. Charaf,
"Eliminating Clustering in the Propagation Tree of Semantic Peer-to-Peer Networks",
IASTED Conference on Parallel and Distributed Computing And Networks, Innsbruck, Austria,
February 14-16, 2006.
[11] B. Forstner, G. Csúcs, K. Marossy, H. Charaf,
"Evaluating Performance of Peer-To-Peer Protocols with an Advanced Simulator",
Conference on Parallel And Distributed Computing And Networks, Innsbruck, Austria,
February 15-17, 2005.
[12] B. Forstner,
"An Analytic Model for Peer-to-Peer Systems with Semantic Overlay Network", AACS'06 Workshop,
2006, Budapest, Hungary
[13] A Symella elérhetősége:
http://symella.aut.bme.hu

# Linux a távközlésben

DEIM ÁGOSTON

*Linux Support Center*
*ago@lsc.hu*

***Kulcsszavak: Asterisk, VoIP, Linux, soft-phone, PBX, nyílt forráskód, SIP, IAX, H.323, ATA***

*A cikk célja a távközlés és a nyílt forráskódú szoftverek kapcsolatáról informálni az olvasót, különös tekintettel a Linux-ra. Elsőként bemutatjuk a Linux eredményeit a szolgáltatói szintű rendszerekben, a második részben pedig a nyílt forráskódú, Linux-alapú PBX és VoIP megoldásokra térünk ki, bemutatva a kisirodai alkalmazásokra szánt PBX-építőelemeket is.*

## 1. Bevezetés

A távközlés és a telefónia világában nem is olyan régen még csak zárt – bár szabványoknak megfelelő – rendszereket találhattak a hozzáértők és érdeklődők, pár éve azonban a szabad szoftverek is betörtek a távközlési piacra. A cikkben bemutatjuk mind a szolgáltatói (carrier-grade) mind a kisvállalati rendszerekben található lehetőségeket, remélve, hogy minnél többen kapnak kedvet a szabad szoftverek használatához.

Felvetődhet a kérdés, hogy miért éppen a szabad szoftver tört előre, van-e egyátalán létjogosultsága a szabad és nyílt forráskódú rendszereknek. A válasz a szabad szoftverek természetében rejlik: nyílt szabványok használata, szabadon hozzáférhető és módosítható forráskód. És ez miért előny? A gyártók szempontjából előnyös, mert a szabad kód miatt nincsenek kiszolgáltatva egy szoftverfejlesztő cégnek, másrészt kevesebb erőforrást kell áldozniuk saját fejlesztésű rendszerekre.

És miért éppen a Linux-alapú rendszerek a legelterjedtebb szabad szoftverek a távközlésben? Ennek fejlesztői és szellemi tulajdonvédelmi oka is van. Szabad szoftver és szabad szoftver között is van különbség. A Linux-kernelben, ha módosítás történik, azt mindenki által hozzáférhetővé kell tenni. Így, ha egy gyártó hozzáadta saját kódját a rendszerhez, egy másik gyártó nem tudja megtenni, hogy bezárja a kódot és saját módosításait nem adja közre. Ezzel biztosítva van a szellemi tulajdon védelme – bár sokan mást próbálnak elhitetni a világgal. Fejlesztők szempontjából pedig fontos, hogy egy kiterjedt felhasználói réteggel rendelkező, saját maguk által teljesen átlátható rendszerrel dolgozhatnak és csak a saját funkciók, kiegészítések fejlesztésével kell törődniük, ezzel időt és pénzt spórolnak meg a gyártók.

## 2. A Carrier grade Linux (CGL)

A távközlésben kiemelt szerepe van a magas rendelkezésre állásnak, így itt már öt-kilences rendelkezésre állás szükséges, tehát az éves állásidő nem lehet több, mint 5 perc.

Több gyártó, felismerve a GNU/Linux rendszerekben rejlő fejlesztési lehetőségeket, az OSDL (Open Source Development Lab) szárnyai alatt 2002 áprilisában létrehozta a Carrier Grade Linux munkacsoportot (CGLWG), melynek feladata, hogy felkészítse az operációs rendszert az ilyen kritikus működésű rendszerek üzemeltetésére. A gyártók között olyan ismert nevek találhatók, mint az Intel, IBM, Nokia, Siemens, Alcatel stb.

A fejlesztések itt is a követelményrendszerek és az elérendő célok kiírásával, megtervezésével kezdődtek, legutóbb 2005 júniusában jelentették meg a CGL követelményeinek legutolsó verzióját, a CGL 3.1-et, melyre még fel kell még készíteni a rendszereket. Ez tehát nem egy saját GNU/Linux kiadás, hanem követelményrendszer, megfelelő munkával bármely Linux disztribúció felkészíthető a telekommunikációban történő felhasználásra. Legutóbb a Hewlett-Packard készítette fel a Debian GNU/Linux 3.1-es verzióját Carrier grade szerverein történő futtatásra, jelenleg még a 2.0.2-es követelményrendszernek megfelelő módon. A célok „egyszerűek”: magas rendelkezésre állás, klaszterezhetőség, szervizelhetőség, teljesítményorientáció, magas rendelkezésre álláshoz szükséges hardverek támogatása, biztonság, nyílt szabványok és alkalmazásfejlesztő segédeszközök létrehozása. A CGL célja nem csak egy terület lefedése, hanem, hogy minél hatékonyabb jelzésrendszer (signaling) -kiszolgáló és -átjáró is legyen, valamint alkalmas legyen a kommunikációhoz tartozó egyéb feladatok ellátására, mint a forgalomszámlálás, számlázás, vagy a felhasználói adatok menedzsmentje. A Linux-szal lefedhető a teljes telekommunikációs terület igénye.

És, hogy mennyire megbízható egy ilyen nyílt megoldás? Az NEC termékpalettáján CGL alapú GGSN és SGSN is megtalálható, melyekkel már több, mint egy éve éles üzemben működik több hálózat is Japánban.

## 3. Linux alapú telefonközpont

Ezt a szekciót két részre kell osztani: egyrészt léteznek már GNU/Linux alapú késztermékek, mint az Alcatel OmniPCX Office, de egy bármilyen szaküzletben kap-

ható alkatrészekből akár magunk is összeállíthatunk saját alközpontot.

A gyártótól származó megoldások mindegyike általános GNU/Linux disztribúciót rejt magában, de ezekben az eszközöben saját fejlesztésű zárt rendszereiket árulják a gyártók. Azonban szívesen használják fel az egyéb nyílt forrású megoldásokat, így például a fentebb említett Alcatel terméket is felkészíthetjük és beállíthatjuk hálózati útvonalválasztónak, csomagszűrő tűzfalnak, cache proxynak és akár e-mail kiszolgálónak is.

A másik, izgalmasabb lehetőség egy saját alközpont létrehozása. Hogy mi kell ehhez? Egy átlagos PC, egy kiegészítő kártya és egy alkalmazás. Kiegészítő kártyából többfélét is találhatunk és többet is fel tudunk használni rendszerünkben, hiszen lehet, hogy szükségünk lesz FXS és FXO portra, de ugyanígy előfordulhat, hogy egy BRI vagy PRI ISDN vonalat kell kezelni az analóg vonalak mellett. Természetesen, ha „vonalas” telefonhálózatra nincs szükségünk, csak VoIP szolgáltatásokra, akkor még kiegészítő kártya sem kell.

## Rövidítések, fogalmak magyarázata

- **Linux kernel:** Szabad forráskódú, bárki által tanulmányozható, változtatható, szabadon felhasználható operációs rendszer mag. A saját változtatásokat elérhetővé kell tenni mindenki számára.
- **GNU/Linux:** Unix-szerű, szabadkódú operációs rendszer, mely a kernelből és a szabad szoftver felhasználói programokból áll.
- **Linux disztribució:** Programcsomag, ami tartalmazza a GNU/LINUX operációs rendszer elemeit, valamint tartalmazhat még tetszőleges, adott feladat elvégzésére alkalmas programokat.
- **CGL:** *Carrier Grade Linux* – Szolgáltatói szintű LINUX, azaz megfelel bizonyos szigorú minőségi, menedzselhetőségi, skálázhatósági követelményeknek.
- **OSDL:** *Open Source Developement Lab* – A Linux kernel és a kapcsolódó alkalmazások fejlesztését, kutatását végzi, a legnagyobb gyártók és cégek támogatásával.
- **SGSN:** *Serving GPRS Support Node* – Mobil távközlőhálózatok része, az úgynevezett csomagkapcsolt szolgáltatások (GPRS) kezelésére.
- **GGSN:** *Gateway GPRS Support Node* – Mobil távközlőhálózatok része, az úgynevezett csomagkapcsolt szolgáltatások (GPRS) kezelésére.
- **FXS:** *Foreign Exchange Station* – Analóg (POTS) telefonszolgáltatás biztosítása a feladata, leggyakoribb előfordulása a fali telefoncsatlakozó, de használható VoIP átjáróba szerelve hívástovábbításra is. POTS környezetben az FXS biztosítja a csengetési feszültséget és a vonalat, amely segítségével az analóg készülékeket „megszólíthatjuk”.
- **FXO:** *Foreign Exchange Office* – Az FXS-től fogadja a jeleket, leggyakoribb alkalmazása a hagyományos telefonkészülék, illetve a VoIP átjárók és a hagyományos telefonközpontok portjai, melyek fogadják a bejövő analóg vonalakat.
- **BRI ISDN:** *Basic Rate Interface ISDN* – ISDN előfizetői hozzáférés 64 kbps sebességen.
- **PRI ISDN:** *Pimary Rate Interface ISDN* – ISDN trönk hozzáférés 2 Mbps sebességgel.
- **POTS:** *Plain Old Telephone Service* – Hagyományos telefonszolgáltatás.
- **DTMF:** *Dual-tone multifrequency* – Telefon készülékekben használt jelzés, a korábban általános tárcsázást, vagyis a vonal szaggatással kiadott jelzését váltotta fel.
- **VoIP:** *Voice over IP* – Telefonálás Internet hálózaton keresztül.
- **SIP:** *Session Initiation Protocol* – VoIP jelzésrendszer
- **H.323:** Az ITU-T által szabványosított VoIP jelzésrendszer
- **IAX2:** *Inter-Asterisk eXchange* – Jól használható, könnyen NATolható VoIP protokoll az Asterisk fejlesztőitől. Támogatja a trönkölést és multiplexelést.
- **MGCP:** *Media Gateway Control Protocol*
- **RTP:** *Real-time Transport Protocol*
- **UDP:** *User Datagram Protocol* – Megengedi a csomagvesztést, ezért a csomagvesztésre kevésbé, ám a valósidejűségre sokkal inkább érzékeny VoIP-rendszerekben szívesen használják.

Megfelelő alkalmazásból már nem olyan nagy a választék, de nekünk csak egyetlen egyre lesz szükségünk, melynek neve: Asterisk. Véleményünk szerint az Asterisk a nyílt forrású alközpont-szoftverek megkoronázott királya. Használatával nemcsak a hagyományos, hanem a VoIP-rendszerek lehetőségei is elérhetővé válnak.

### 3.1. A PBX szoftver: Asterisk

Egyedi fejlesztésnek indult, de azóta olyannyira kinőtte magát a projekt, hogy külön cég szerveződött köré és mi sem mutatja jobban erejét, hogy külön nemzetközi konferenciasorozaton (AstriCon) is bemutatják. A cég úgy tud sikeres lenni, hogy megtartotta a szabad forráskódú szoftvert. Természetesen a hardver eladásokból realizálják a legtöbb bevételt, de akinek igénye van arra, hogy hivatalos támogatást kapjon az Asteriskhez, annak lehetősége van megvásárolni a szoftvert. Ezen kívül lehetőség van hivatalos Asterisk szakértőnek, fejlesztőnek is elismertetni magunkat, amennyiben sikeresen levizsgázunk. Látható tehát, hogy egy működő ökoszisztéma alakult ki a cég körül és sikeresnek lehet lenni nyílt forráskóddal is. Akinek nincsen lehetősége tanfolyamot meglátogatni (erre egyelőre csak az USA-ban és Nyugat-Európában van erre lehetőség) az válogathat több könyv közül, de a hivatalos O'Reilly kiadó által megjelenetett könyvet PDF formátumban ingyen is le lehet tölteni.

Az online „Biblia": *www.voip-info.org*. Ezen az oldalon szinte minden ott van, ami a VoIP kommunikációval kapcsolatban létezik. Nézzük ezek után a tényeket, mit támogat az Asterisk, milyen lehetőségeink vannak.

Röviden összefoglalva: a lehetőségek határtalanok. Nem csak azért, mert szabad szoftver lévén bármit belefejleszthetünk vagy fejlesztethetünk, hanem mert már most is rengeteg lehetőséggel bír. Az olyan alapfunkciókon kívül, mint a hívástovábbítás, feketelisták, „Do Not Distrub"-mód, DTMF támogatás, hívásvárakoztatás stb., olyan lehetőségei is vannak, mint hangposta vagy az SMS kezelés.

A támogatott VoIP-protokollok közé tartozik a SIP, a mai VoIP szolgáltatások alapköve, a H.323, az IAX2, Cisco Skinny (SCCP) és az MGCP protokoll.

Itt térjünk ki röviden az IAX2-esre is. Ez egy Asteriskhez fejlesztett hatékony protokoll, mellyel össze lehet kötni különböző Asterisk-kompatíbilis VoIP-telefonokat, ATA-kat és Asterisk-szervereket. Aki foglalkozik hálózati határvédelemmel, az tudja értékelni a SIP és az IAX közötti különséget, ha a szűrhetőségről van szó. Aki nem foglalkozik határvédelemmel, annak röviden annyit, hogy míg az IAX2-es esetében elég egy portot szűrni, addig a SIP esetében ez egy 10000 portos nyitott tartományt (10-20000-es UDP portok) jelent, melyet be kell engedni a hálózatba és ráadásul még az is dinamikus, előre nem meghatározható, hogy mit szeretnének használni a kommunikáló felek az kommunikáció RTP-n megvalósuló részében. Még egy apró különbség: a IAX2-nek nem jelent problémát a NAT (hálózati címfordítás), ami nem mondható el a SIP-ről. (A NAT – Network Address Translation – a tűzfalak és routerek által leggyakrabban használt technika arra, hogy a LAN-on levő eszközöknek magán IP-címeket adjuk, és ezáltal osztozzanak egy nyilvános IP-címen.)

Kodekek tekintetében támogatja a legelterjedtebb kódolási formákat, de akár – igaz, külön pénzért – hozzájuthatunk G.729-es kodekhez is, de nem ismeretlen számára a GSM kódolás sem. A támogatott kodekek a következők: ADPCM, G.711, G.723.1, G.726, G.729, GSM, iLBC, Linear, LPC-10, Speex.

### 3.2. Interfész kártya lehetőségek

A szabad szoftvereknél ugyanúgy igaz, mint minden más szoftvernél, hogy az alkalmazott hardvereszközökkel kompatíbilisnek kell lenni. Szerencsére az Asterisk rendkívűl jó támogatottsággal rendelkezik, ami nem kis mértékében köszönhető annak, hogy az Asterisket útjára indító Digium önmaga is foglalkozik analóg, valamint ISDN kártyák tervezésével és gyártásával.

A kártyák közül most csak az ajánlottakkal foglalkozunk, de ez a lista nem jelenti azt, hogy ne lenne támogatott több gyártó kártyája, de érdemes az Asterisk weboldalán utánanézni, vajon támogatott-e.

*Digium kártyák*

A legteljesebb támogatást természetesen ezek a kártyák élvezik, hiszen a szoftver és hardver gyártója megegyezik (ennek néha ellentmond az élet, de itt történetesen igaz). A kártyák között találhatunk analóg (POTS) és PRI ISDN kártyákat is, de nem foglalkoznak BRI ISDN kártyákkal. A PRI ISDN minden típusa támogatott (E1/T1/J1) és vannak kártyák, melyek rendelkeznek visszhang-elnyomással (echo cancellation).

*Billion kártyák*

Ha BRI ISDN-re van szükségünk, akkor ennek a cégnek a kártyái megfelelő működést fognak biztosítani. A kínálatban nem csak egy vagy két, hanem négy és nyolc (!) porttal rendelkező kártyát is találhatunk. Külön meghajtóprogram szükséges hozzá, de az is szabad forráskódú és a kártya 100%-ban kompatíbilis az Asteriskkel.

### 3.3. A végfelhasználó kapcsolata: telefonkészülékek

Több lehetőségünk közül is választhatunk, nem vagyunk adott gyártó rendszerkészülékeihez kötve, mint kereskedelmi rendszereknél. Egyrészt használhatunk „hardveres" telefonokat és használhatunk soft-phoneokat. A szoftveres megvalósításokról később ejtünk pár szót.

A készülékes telefonoknál gyakran felmerül a kérdés, hogy mihez is kezdjünk régi jól bevált analóg készülékünkkel. Nos, erre is van megoldás, hiszen a piacon létezik mind a SIP-et, mind az IAX2-t is ismerő ATA, tehát nem kell kidobni régi készülékeinket. Egyszerűen csak csatlakoztatunk egy ilyen készüléket a telefonhoz, sőt, ezeket az ATA-kat általában távolról is menedzselhetjük. Ilyen ATA készülék a szintén a Digium által gyártott IAXy S100I, mely természetesen az IAX2-es proto-

kollon keresztül kommunikál. Amennyiben SIP-es ATA-ra vágyunk, sokkal bővebb a kínálat, gyakorlatilag majdnem minden hálózati eszköz gyártó rendelkezik SIP képes ATA-val az alacsonyabb árkategóriában is, és itthon is kaphatók ezek az eszközök, például a Linksys és a Grandstream eszközei.

Másik megoldás, hogy egy VoIP-képes telefonkészüléket veszünk, melyet közvetlenül a struktúrált hálózatba kötve, a megszokott módon tudunk telefonálni. Az ilyen készülékek kínálata rendkívül széles, az egyik legolcsóbb és legelterjedtebb megoldás a Grandstream BT 101-es és 102-es típusa, valamint egy kevésbé ismert termék az 1Tek. Utóbbi készülék újabb firmwarerel még az IAX2-es protokollt is ismeri, így még jobban integrálható az Asteriskkel.

A másik lehetőség a szoftveres „készülékek" használata. Itt nincsen szükség ugyan külön hardverre – a mikrofonon és fejhallgatón kívül – de itt is fontos, hogy milyen alkalmazást használjunk. Kedvencek az Ekiga és az X-Lite. Míg az előbbi leginkább csak szabad operációs rendszerek alatt működik, addig az utóbbinak van Windows, Linux és MacOS X-en futtatható verziója is.

Ezek az alkalmazások a SIP-et támogatják, de rendkívül kellemesen használható és stabil szoftverek. Az X-Lite nem szabad szoftver ugyan, de ingyenes. Rendkívül sokrétű, támogatja az hang- és videókonferenciákat is, rögzíthetjük a beszélgetéseket, valamint rendelkezik – szoftveresen megvalósított – visszhang elnyomással is. Ezenkívül támogatja a SIP-et NAT-on keresztüli használatra alkalmassá tévő STUN lehetőséget is. Létezik egy többet tudó, kereskedelmi változata is az X-Lite-nak, ez az eyeBeam.

A kereskedelmi verzióban további lehetőségként Outlook integrációs lehetőséget is találunk, valamint hat vonalat képes kezelni – az X-Lite kettőt – és tíz különböző SIP azonosítót is beállíthatunk a szoftverben. A továbbiak csak a kereskedelmi verzióban megtalálható lehetőségek: a titkosított RTP csatorna és az IPV6 támogatás.

#### 3.4. Gyors kezdés: Trixbox

Korábbi nevén Asterisk@Home. Ez egy előre összeállított, CD-re írható Linux-disztribúció, amit kiegészítettek olyan adminisztrációs szoftverekkel, melyek megkönnyítik az Asterisk-kel történő megismerkedést. Segítségével egy szinte azonnal működő rendszerhez jutunk, de teszt céljából is kiváló lehetőség a használata. Megfelelően erős géppel rendelkező felhasználók virtuális gépre is telepíthetik a rendszert (például VMWare vagy QEMU alá).

A projekt azzal dicsekszik, hogy segítségével egy órán belül össze lehet állítani egy működő rendszert. Valóban elég lehet ennyi idő telepítéssel együtt, ha már előzetesen tudjuk, mit szeretnénk és mit kell beállítani. A rendszer használata magától értetődő, a telepítéskor megadott jelszóval be kell jelentkezni root-felhasználónéven (Unix-szerű rendszereken, mint amilyen a Linux is, a root a rendszergazda) és a rendszer kiírja a böngészőből elérhető adminisztrációs felület elérését.

A projekt honlapja és fájljai a SourceForge oldalán találhatók meg, nagyon sok szabad szoftveres projekthez hasonlóan. Apró nehezítés, hogy jelen pillanatban a legújabb verzió helyett az oldalon közvetlenül csak egy régire történik hivatkozás, de a linkek között megadott helyen elérhető az utolsó verzió, csak ki kell választani a megjelenő oldalon a legszimpatikusabb tükörszervert, de megadtunk egy direkt linket is.

## 4. Összefoglalás

A szerző reméli, hogy a cikk, ha csak bevezetésként is, de meggyőzte az olvasókat, hogy a Linux és a szabad szoftverek mind a nagyvállalati, mind a kisvállalati, illetve otthoni felhasználásban megállják helyüket.

**Irodalom**


[1] A CGLWG munkacsoport tagjainak listája: http://groups.osdl.org/workgroups/cgl/
[2] http://www.digium.com
[3] http://www.asterisk.org
[4] Asterisk könyv: http://www.nufone.net/downloads/asteriskdocs/AsteriskTFOT.zip
[5] http://voip-info.org/wiki/
[6] X-Lite: http://www.xten.com/index.php?menu=download
[7] www.trixbox.org
[8] Trixbox CD image: http://prdownloads.sourceforge.net/asteriskathome/trixbox-1.1.iso?download illetve egy direkt link: http://switch.dl.sourceforge.net/sourceforge/asteriskathome/trixbox-1.1.iso

# SDL–UML társulás: UML2.0

MEDVE ANNA

*Pannon Egyetem, Műszaki Informatikai Kar, Információs Rendszerek Tanszék, Veszprém*
*medve@almos.vein.hu*

***Kulcsszavak: SDL, UML, modellezés***

*Az egyre összetettebb informatikai rendszerek szoftverfolyamatában elfogadottá vált a tervezés és a modellezés fázisok fontossága. Mindez meghatározta a szoftverfolyamat kezdeti szakaszait támogató nyelvek fejlődését. Az eredetileg dokumentálásra szánt specifikáló és leíró nyelvek alkalmazását napjainkban a validáló, verifikáló és szimulációs eszközök tárháza segíti. Az automatikus tesztgenerálás és szimulációs módszerek lehetővé teszik a hibák felfedezését már a fejlesztés korai stádiumában. Ugyanakkor, a támogató eszközök fejlesztéséhez szükséges, hogy a támogatott nyelv kellően kifejező és egyértelműen definiált legyen. Az SDL (Specification and Description Language) és az UML (Unified Modelling Language) a legelterjedtebben használt modellező nyelvek az iparban. Az SDL kezdetektől, az UML a 2. verziójától formális nyelv. Cikkünkben röviden bemutatjuk mindkét nyelvet, komplemens jellegüket ismertetve alkalmazásuk variálhatóságát, továbbá a távközlés fejlődéséből eredő azon jellemzőiket, amelyek a két nyelv konvergenciájával új modellezési irányzatok magvalósításához vezettek.*

## 1. Bevezetés

*„Navigare necesse est…"*

Az egyre bonyolultabb rendszerek megvalósításához a szoftverrendszerek fejlesztésben bevált gyakorlat a dolgok összetettségére a komponensek fejlesztése, a történések összetettségére pedig a komponensek kommunikációs alapokra helyezése. A szoftverrendszerek fejlesztésében manapság vitathatatlanul bekövetkezett a modern fizika történetében tapasztalt formalizálás szükségessége; elkülönített technikák és módszerek kellenek a dolgok és összefüggéseik elvonatkoztatására, valamint technológiák a dolgok megvalósítására. Ennek szellemét hordozza alapjaiban úgy az UML mint az SDL, előbbi a dolgok és összefüggéseik, utóbbi a dolgok és kapcsolatukban való történéseik leírására alkalmasabb.

A formális nyelvek Z. szabványcsaládját [2] az ITU (International Telecommunication Union – Telecommunications Standardization Sector) a kommunikáló protokollok és kommunikációs szolgáltatások [1] modellezésére fejlesztette ki és bővíti folyamatosan. A szabványcsalád több eleme az MDA képességű specifikáló és leíró nyelv, az SDL köré csoportosul. Az UML az egységesített modellező nyelv, az OMG [6] fejlesztésében szabványosított a szoftverrendszerek fejlesztésének elemzés és tervezés szakaszára, főként az információs rendszerek fejlesztésében terjedt el.

Az OMG és az ITU-T munkacsoportjai létrehozták az UML2.0 szabványt az UML, az SDL és az MSC szabványainak összetársításával. Így az UML2.0 az MDA modellezést gyakorlattá tevő általános modellező nyelv szabványa lett.

A szabványok nem kínálnak módszertant a nyelvek alkalmazására, ezért célszerű figyelembe venni a társulásba beemelt nyelvek fejlődésének korábbi szakaszából általánosan és az egyes célterületek problémáira nyert technológiákat és technikákat. Az alábbiakban röviden bemutatjuk az SDL és UML nyelveket. A komplemens jellegüket ismertetve mutatjuk meg alkalmazásuk variálhatóságát, a távközlés fejlődésének hatásait a nyelvek alkalmasságának oksági összefüggéseiben hivatkozzuk, mindezek előtt a modellezés természetének és ebben a formális nyelvek szerepének felvillantásával a két nyelv növekvő szerepét hangsúlyozzuk.

## 2. A modellezés természete a szoftverfolyamatban

*„A számítástechnika nincs szorosabb kapcsolatban a számítógépekkel, mint a csillagászat a távcsövekkel."*
*Dijkstra*

A modellezés alkotó tevékenység, amelynek elengedhetetlen tartozéka az alkotást lehetővé tevő módszerek és eszközök rendszere. A modellelemek átnézése, egybevetése, kombinálása közben mentális folyamatokkal új felismerésekhez jutunk, amelyek kifejezéséhez szükséges formalizmus nélkül nem teljes az ábrázolás. A modell jósága ily módon függ *a modellező nyelv kifejezőerejétől.*

A szoftverfolyamat egyik fontos követelménye a piackövető szoftvertermékek előállítása, amely teljesülni látszik a dolgok és összefüggéseik, valamint a megvalósításukhoz és működtetésükhöz szükséges technológiák elválasztásával. A szoftver közvetve vagy közvetetten piaci termékké változott, sorozatgyártása előtt prototípust célszerű megadni: *modellezéssel és automatizálással,* mennél gyorsabban és olcsóbban.

A modellezés előnye a szoftverfejlesztésben a szoftver azon tulajdonságából adódik, hogy a szoftver természeti törvényekkel nem határolható be, ezért a működéskori valóságra vonatkoztatott hipotéziseket és pre-

dikciókat is modellezéssel állítjuk elő, az úgynevezett követelménymodellt, majd ezek transzformációi adják a megvalósítani kívánt rendszermodellt. Manapság elvárás szintű a fejlesztőkörnyezetektől az automatizált modell-transzformációs támogatás.

A modell megvalósításának automatizálhatósága függ attól, mennyire jól definiált a modellező nyelv,. A jól definiált modellező nyelv a formális nyelv, amellyel a modellezés eredménye a formális specifikáció. A formális specifikáció a bemenete megfelelő társítású transzformációs eszköztárnak, amellyel automatizáljuk a szoftverfolyamat részeit [19,20,3]. A modelltranszformációk helyességéről meg kell győződni a hibátlannak tudott rendszermodellből generált kód előállítása előtt.

Az automatizált modellfejlesztésnek az egyik eszköze a formális modell, irányzata az MDA [21], a Model Driven Arcitecture, célja a hordozható, újrafelhasználható, platformfüggetlen szoftvertermék előállítása. A technológia-függetlenséghez el kell tudni választani a szoftvertermékre vonatkozó valós világbeli generikus tulajdonságokat a specifikus tulajdonságoktól már a modellalkotás szintjén.

Ily módon kell a platformfüggetlen modell (PIM), melyből további módszerekkel és eszközökkel állítható elő az a platformfüggő modell (PSM), amely tartalmazza a szükséges technológiai információkat a modell megvalósításához a kijelölt platformon. A PIM szerepei (és az ezt támogató modellező eszközök) a formális specifikációból generálható kóddal váltak jelentőssé az összetett piaci változások követésére.

## 3. Az SDL specifikáló és leíró nyelv

Az *SDL (Specification and Description Language)* nyelvet az ITU-T a Z.100 szabvány [10] ajánlásaiban definiálták komplex rendszerek specifikálására. A rendszer működésének leírását a konkurens módon diszkrét jelekkel kommunikáló, valósidejű és interaktív folyamatok eseményvezérelt ábrázolásai alkotják.

Az SDL fejlődése a távközlés fejlődésében történt [10-13], létrehozása egy 1968-as ITU tanulmányból indul ki a programvezérlésű kapcsolórendszerek kezelésére vonatkozóan, aminek eredményeként 1972-ben egyetértés születik arról, hogy nyelvek szükségesek a gépek és berendezések interakcióinak leírására és programozására. Az első SDL szabvány 1976-ban az alap grafikus leírónyelv, majd négyévenként további fejlesztéseket jelentettek meg. A Z.100-as szabvány jelenleg az SDL-2000 verziót jelenti, amelyben növelték az objektumorientáltságot, valamint bevezették az ágens-elvet, viszont ez utolsó verzió absztrakt gépe még nincs megvalósítva, így az elterjedt fejlesztőkörnyezetek az SDL-96 szabványát támogatják.

Az SDL nyelvet 1996 óta használják a távközlési iparon kívül is, elsősorban az orvosi felszerelések iparágában, az autó- és repülőgépiparban. Az SDL eszközök piaca 1996-2000 között jelentősen növekedett. A fejlesztőkörnyezetek generálnak programozási nyelvekre forráskódokat (általában C/C++, Java), amelyeket be lehet szerkeszteni a valósidejű rendszerek termékgyártásába. Az SDL alkalmazását megkönnyíti, hogy az egymásnak kölcsönösen megfeleltethető, grafikus és szöveges nyelvi implementációi, az SDL/GR és az SDL/PR vannak használatban specifikálásra, tervezésre, dokumentálásra, megvalósításra. A megvalósítás alapja a hibátlan formális specifikáció.

Az SDL matematikai alapja a kiterjesztett véges automata (EFSM) modell, amely a rendszer működését gerjesztés-válasz módon határozza meg a rendszert alkotó kommunikáló automaták állapotátmeneteinek halmazán. A rendszer több egymással és a rendszerkörnyezettel kommunikáló automatából áll, ahol a kommunikáció jelekkel történik a FIFO elven működő csatornákon és jelútakon. Az SDL rendszer struktúráját a kommunikáló részegységek, a rendszer dinamikus viselkedését a kommunikáló automaták állapotátmenet-szekvenciái adják.

Az ábrázolandó rendszer szerkezetét hierarchikusan a *rendszer alatti blokk, processz és eljárás* egységek egymásba ágyazásai alkotják. A processz a hierarchia levélszintjén maga a kommunikáló automata, az eljárás algoritmusokat és csoportosított automata állapotátmeneteket tartalmazhat. Ezáltal bármely célú változtatás a faszerkezetben egyértelműsíthető.

Az SDL specifikációk jóságához szükséges az *MSC (Message Sequence Charts)* [14] nyelv használata, melynek szerkezetei segítik a rendszerentitások hierarchiába szervezését, valamint a dinamikus viselkedés formális analízisét. Az MSC teljes szabványát beemelték az UML 2.0 verziójába.

Az adatok ábrázolására az SDL-ben a beépített típusgeneráló lehetőséget ad bármely adat leírására, előnyös a beépített *ASN.1(Abstract Syntax Notation One)* adatleíró nyelvet alkalmazni [15]. Ennek nagy előnye a nyílt rendszerek ábrázolásakor érvényesítődik az együttműködő képesség fokozásával.

Az SDL fő szabványa a Z.100, további szabványokkal terjesztik ki a nyelvet a komponensalapú és platformfüggetlen fejlesztés támogatására. A Z.105-ös szabvány megadja az ASN.1 modulok kapcsolását az SDL adatleírásokba direkt módon, a Z.107-es szabványban rögzítik az ASN.1 beágyazását az SDL nyelvbe. A Z. 109-es szabvány definiálja az SDL-nek megfelelő UML profilt, így hozzáadja az UML elv szerinti ábrázolást. A Z.120-as szabványcsalád az MSC nyelvet hasonlóan definiálja.

Az *SDL fejlesztőkörnyezetek* legtöbbjének közös funkciói a grafikus szerkesztő, a szöveges és grafikus konverzió, a statikus elemző, a kódgeneráló, a szimuláló és validáló dinamikus elemzés, az MSC-vel kombinált támogatás [4].

Az SDL nyelvről ismertetést angolul az [4] fórumán, magyarul a Híradástechnika 2005/10. számában olvashattunk [5]. Az SDL kutatásának hazai képviselőinek eredményei elsősorban a specifikáció-bázisú tesztelés és validálás terén az automatikus tesztsor generálásra és szelektálásra adott számos algoritmus [8], az SDL

alkalmazása a hardver-szoftver együttes tervezésére [7], a konformancia teszteléssel összefüggésben [1], valamint formális módszertani összefüggésben [3].

## 4. Az UML egységes modellező nyelv

Az *UML (Unified Modelling Language)* nyelvben [21] a rendszermodell többféle modelltípusból tevődik össze, amelyek kötelező részletezettsége függ a modellezés céljától. Szimulációhoz vagy kódgeneráláshoz teljes és konzisztens rendszermodellt kell szerkeszteni.

Az egyes modelltípusokat különböző diagramokkal ábrázoljuk, amelyek közötti szemantikai összefüggések jóságát és helyességét modellverifikálással és validálással kapjuk meg. Az UML rendszermodellt alkotó modelltípusok és a modelltípus ábrázolásához szükséges diagramtípusok a következők:

- *A rendszerhasználat eseteinek modellezése a használati eset-, csomag- és osztály diagramokkal.*
  Ebben a modellben döntjük el a rendszer, alrendszer, komponens vagy osztály környezetét. Használati eset diagramban a kapcsolatok megadásával a használati esetek közötti általánosítást, magában foglalást, kiterjesztést, függőséget adjuk meg, míg a kollaborációval illusztrálhatjuk, hogy hányfajta különböző kombinációja lehet a szereplő interakcióinak más használati esetekkel vagy a rendszer többi részeivel (a tárgyakkal).
- *A rendszer objektumainak és ezek strukturális kapcsolódásainak modellezése a csomag-, osztály- és architektúra (kompozíciós) diagramokkal.*
  Ebben a modellben a nagy rendszerek leírásánál csomagokat szerkesztünk miután a rendszer elemeit azonosítottuk és osztályba foglaltuk. Egy osztálydiagram modell elemei az osztály, attribútum, operáció, (port, interfész, jel, jellista, időzítő csak a 2.0 verzióban), adattípus, választás, állapotgép és kapcsolat. Azokat az objektumokat, amelyek ugyanazt a tulajdonságot, viselkedést, és más objektumokkal ugyanazt a kapcsolatot mutatják, egybefogjuk, és annak az osztálynak az objektumaként modellezzük. Egy osztálydiagram az objektumtípusok közötti strukturális, és viselkedési kapcsolatokat mutatja meg, valamint a kompozíciókkal az aktív osztály kommunikációs jellemzőit strukturáljuk.
- *Rendszerviselkedések modellezése a tevékenység-, szekvencia- és állapotgép diagramokkal.*
  Ennek a modellezésnek a feladata, hogy megmutassa a viselkedést azzal, hogy ezt kis viselkedésegységekre bontja, leírva a vezérlő és adatfolyamokat ezen egységek között.
- *A rendszer elosztásának (komponenseinek) modellezése a komponesdiagrammal.*
  A komponensmodellezés feladata, hogy azonosítsuk a rendszer komponenseit, és modellezzük az interfészeiket, és kapcsolatukat. A rendszer statikus nézetét mutatja meg, a komponenseken, a realizált interfészeken, és a szükséges interfészeken keresztül. Itt csak a komponens fogalmát kell ismertessük, ami nem más, mint a rendszer egy jól elkülöníthető része, ami jól leírható szolgáltatásokat nyújt. Az UML nem nagyon különbözteti meg a komponenset az osztálytól, mindent amit megtehetünk egy osztállyal, azt megtehetjük a komponenssel is. Egy komponensdiagramban kettő vagy több elem között lehet társítás, aggregáció, kompozíció, függőség, általánosítás, implementálás.

Az *UML szabványai* az UML1.x és a kommunikáló automata alapokra helyezett UML 2.0 filozófiájában eltérnek egymástól. Az UML2.0 szabvány kiindulópontja az ITU-T Z.109-es szabványa, amely definiálja az SDL-nek megfelelő UML profilt. Az UML és SDL házasságából a legnagyobb haszon a mindkét oldalról beemelt hozzátartozók fejlesztés-támogató eszköztára.

*Az UML 2.0 formális nyelv,* amely alkalmas a formális specifikáció előállítására.

Egyszerűsítve mondhatjuk, hogy az UML2.0 szekvencia diagramja lett a teljes MSC-2000 nyelv, valamint az UML2.0 állapotgép diagramja lett az SDL processzdiagramja, az osztálydiagram elemei közé létrehozták az kommunikáció egyértelműsítéséhez szükséges port, interfész, jel, jellista, időzítés elemeket. Mindez maga után

*1. táblázat*

*Az SDL elemek UML megfelelői*

| *SDL* | *UML* |
|---|---|
| Rendszerdiagram | Osztálydiagram |
| Blokkdiagramban | Architektúra diagram |
| Processzleírás | Állapotdiagram |
| Rendszer | Csomag az osztálydiagramon, vagy tárgy a Használati eset diagramon |
| Blokk | Aktív osztály |
| Processz | Egy aktív osztály operációja |
| Csatorna | Konnektor az Osztálydiagramon, vagy Architektúra diagramon |
| Jelút | Konnektor az Osztálydiagramon, vagy Architektúra diagramon |
| Környezet | A Használati eset diagram szereplői |

„húzta” a TTCN-3 [16] szabványát az ITU berkeiből. A nyereség azonnal adódott: a validáló és szimulációs eszközök, a tesztelés támogatása, az automatikus kódgenerálás.

Az UM2.0 2003/2004-re megújult négy nagy strukturális részei láttatják alkalmazhatóságának sokrétűségét:

– *„Infrastructure”:* az UML nyelvi specifikációjának alapját adó szabványokat azonosítja,
– *„Superstructure”:* az UML nyelvi specifikációja,
– *„Diagram Interchange Model”:* egyfajta diagramkapcsolati interfész a többféle nyelv és fejlesztőkörnyezet közötti átjárásra,
– Az *OCL (Object Constraint Language)* objektumspecifikációs nyelv, amely utasításszerű specifikációs eszköze az UML megvalósításokhoz szükséges pontosabb feltételrendszer megadásának. Az OCL által az UML2.0 követelménytervezés leírása algoritmikussá válik, pontosított adatbázistervet tudunk származtatni.

*Az UML profilok sajátja* a különböző fő iparágak számára beépített UML sztereotípusokkal megadott szakterületi fogalmak és műveletek rendszere, amelyben az ábrázoló eszköztárhoz kifejlesztik az adott szakterületre jellemző, követelmény-elemzést támogató fejlesztőkörnyezetet. A szakterület-specifikus fejlesztéstámogató eszköztárakkal az UML nyelv az úgynevezett domén-specifikus nyelvek felé tolódik el, általában új nevet is kap az átszabott nyelv. UML profilok jellemzően az üzleti modellezés, az autóipar, repülőgépipar, a monitorozó- és vezérlőrendszerek iparában születnek. Bővebben a [21], magyarul a [22] irodalmak tájékoztatnak.

## 5. Az SDL és az UML egymás komplemensei

Mindkét nyelv az inkrementális fejlesztést nyújtja – nem dobunk el köztes ábrázolásokat – az SDL és UML nyelvek diagramszemlélete közötti nagy különbség ellenére mindkettőben jól hangolható a spirális fejlesztés, ami nagyfokú eloszthatóságot jelent időben és földrajzi egységben. Megjegyezzük, hogy ehhez vannak a modellező nyelvben nyelvi eszközök, a szoftverfolyamat módszertanát nem, csak eszközeit adja az UML, ugyanakkor az SDL nyelv kötöttségei diktálják a fejlesztés módszertanát is.

Az SDL és UML elemek megfeleltetését az *1. táblázatban,* az UML és SDL elemek megfeleltetését pedig a *2.táblázatban* mutatjuk be.

Az UML a jelek útjai között nem tesz olyan megkülönböztetést, mint például az SDL, ami jelutat és csa-

| *UML* | *SDL* |
|---|---|
| Használati eset (Use case) diagram | Rendszerdiagram |
| Szekvencia diagram | MSC leírás, az SDL állapotátmenetekből generálható |
| Osztálydiagram | Rendszerdiagram |
| Arhitektúra diagram, Kompozíció | Blokkdiagramban |
| Állapotdiagram , Állapotgép | Processz leírás |
| Szövegdiagram | Nincs konkrét megfelelője |
| Használati eset | Blokkok vagy processzek halmaza |
| Szereplő | Környezet |
| Tárgy | Rendszer |
| Csomag | Blokk |
| Osztály | Blokk |
| Operáció | Blokk egy processze |
| Aktív osztály | Blokk egy processze |
| Port | Konnektor |
| Interfész | Csatorna irányonként, |
| Jel | Jel |
| Rész | Processz |
| Konnektor | Jelút |
| Viselkedési port | Jelút |

*2. táblázat*

*Az UML elemek SDL megfelelelői*

tornát használ, ugyanakkor az UML, a bemenő és kimenő interfészekbe különítéssel egyértelműsíti az SDL-beli csatorna irányoknak megfelelő jelcsoportosítást, többletfeltételekkel megerősítve elérésüket az egyes interfész-csoportosító portokkal, ilymódon a tesztesetek és ellenőrzésük egyértelműbbé válik. Az UML interfészek definiálása bonyolultabb kommunikáció leírását teszi egyértelművé azáltal, hogy osztályszinten tudjuk megadni a jellistát és az operációkat a küldő/fogadó osztályra.

Ezeket az interfészeket az osztályok blokkjaihoz kapcsoljuk, így ezek együtt felfoghatók az SDL-beli csatornáknak, vagy jelutaknak, és azok egyfajta kiterjesztéseként, mintha csatorna-alrendszert deklarálnánk SDL-ben. A jelek halmazát mindkét nyelvben a jellistával (signallist) definiálhatjuk, de UML-ben lehetőség van arra is, hogy összefogott attribútumok halmazát egy jelnek tekintsünk.

Az SDL rendszerdiagram környezet fogalma is másképpen jelenik meg az UML-ben azáltal, hogy a struktúrák ábrázolása nem szigorúan hierarchikus, és a társítások foka ábrázolható. Ezért látványosabb az alsóbb szintű struktúrák konkrét környezete, azaz viselkedés szempontból is konkrétan megnevezhetők a környezeti elemek. Az SDL-ben alkalmazott automata egyedi azonosító az UML-ben típusleírással és feltételekkel megerősítve új eszköz lett az eseményvezérelt folyamatok komponenseinek fejlesztésére.

Összehasonlítva a két nyelvet láthatjuk, hogy nagy, komplex rendszereknél kifizetődőbb UML-ben modellezni, mint SDL-ben. Mivel eléggé hasonló módon lehet létrehozni az UML osztály- és állapotdiagramját és az SDL rendszer-, blokk-, és proceszdiagramját, kijelenthetjük, hogy azoknak is megéri áttérni erre a nyelvre, akik eddig SDL-ben fejlesztettek. Az UML nyelvben abból a szempontból is kifizetődőbb a fejlesztés, hogy rendszeralternatívákat a fejlesztés bármely fázisában tudunk viszonylag kevés munkaráfordítással megadni a megrendelők felé, ami SDL esetén már újratervezést jelent, ellenben könnyebb a feladat a hierarchiából adódóan.

SDL-ben fejleszteni kifizetődőbb a protokollfejlesztésekben, ahol a szabványokban rögzített az ellenőrzött követelményterv, amely, általában MSC-, gyakran SDL specifikációkat is tartalmaz az egyértelmű olvasatukhoz. Arra a kérdésre keresve a választ, hogy megéri-e esetleg először SDL-ben létrehozni az egyes diagrammokat, majd azt UML-be importálni, kijelenthetjük, hogy nem, kivéve, ha az SDL fejlesztő az áttérés stádiumában, UML módszertani ismeretek hiányában, SDL-ből képezi le az UML modellnézeteket.

A két nyelv között az automatikus átjárás mindkét irányban lehetséges a Telelogic fejlesztőkörnyezetein, manuálisan a diagramok összevetése és együttes használata is gyorsítja a modellezés folyamatait. Az SDL modellek újratervezését érdemes UML-ben megadni: a meglévő SDL rendszer importálásából megkapjuk a modellelemeket és UML-ben fejleszthetjük az újratervezést.

## 6. Az SDL és az UML konvergenciájának MDA jellegű hozamai

Az UML egységes modellező nyelv rendszerspecifikálásra és architektúra-szintű tervezésre alkalmas, kiterjeszteni a nyelv képességét a szerkesztett modell implementálására a sztereotípusok beépítésével lehetséges. Az UML2.0 szabvány támogatja a modellvezérelt paradigmát (MDA), az UML profilokban az automatizált transzformációkkal és egyéb szakterület-specifikus környezettámogatásokkal. Elvárt gyakorlat a diagramcserék átjárhatóságát biztosítani a fejlesztés különböző szakaszaiban, a különféle fejlesztőkörnyezetek között.

Az SDL komponensek üzenetcsere alapú kommunikációja valamint a komponens elvű fejlesztés eredendően tartalmazza az SDL fejlesztések MDA jellegét. Ennek megfelelő UML2.0 elemek az objektumtervezésben kapnak hangsúlyt az aktív osztályokhoz tartozó attribútumok és a metódusokhoz tartozó interfész-, jel- és idő- osztályok szerepében.

Ugyanis a spirális fejlesztéssel a részletes objektumterv előtti szimuláció kínálja a platformfüggetlen fokozatokat. További kérdéssé válik, hogy a fejlesztőkörnyezetek vizuális (egér-) programozású támogatásai előnyt vagy hátrányt szenvednek-e a részletes objektumterv spiráljaiban. Például a Telelogic Tau G2 2.6-os változatában a részletes objektumterv többlépcsősre bontásával (inkrementális és spirális szoftverfolyamat) előnyt veszítünk a modellelemek készletezéséből nyerhető egérhúzásos technikák terén.

Az ITU-T nyelvek fejlesztőcsoportja System Design Language néven az MDA paradigma mentén határozta meg a Z. szabványcsalád azon elemeit, amelyek a teljes fejlesztési folyamatot támogatják. Az ITU-T szabványkészlete és módszertana az URN-MSC-UML-ASN.1-SDL-TTCN szabványokra épülő inkrementális módszertant ajánlja [2].

A rendszer életciklusaiban alkalmazható formális nyelvek kapcsolatát szemlélteti az *1. ábra,* amelyben megmutatkozik az SDL nyelv kohéziós szerepe az implementációs jellegéből adódóan. Az SDL-Forum és az ITU-T munkacsoportjai dolgoznak a nyelvek egy fejlesztőkörnyezetbe integrálásán.

A Z. 150-153. szabványok *User Requirements Notation (URN)* szabványa két formális nyelvet, a *Use Case Map (UCM)* és *Goal Requirements Language (GRL)* formális nyelveket tartalmazza, rendre a funkcionális követelmények és nem-funkcionális követelmények ábrázolására.

A modellezés folyamatában a lépések az UCM leképezése MSC diagramokra, amelyek többféle módon építhetők be az UML modellbe, (osztály-, szekvencia-, tevékenység diagramként), az SDL modellbe áttranszformálhatók a statikus és dinamikus leírásba. Az ASN.1 deklarációk által többféle módon gyorsul és egyszerűsödik a *Test and Test Control Notation (TTCN-3)* modulok alkalmazása a validálás és teszteset generálás folyamataiban. Az SDL specifikáció az alapja a TTCN tesztgenerálásnak és a forráskód készítésének C++

vagy JavaScript nyelven. A Deployment and Configuration Language (DCL) telepítő és konfigurálás leíró nyelv a kidolgozás folyamatában van, elvei és elemei az objektum (eODL) és interfész leíró (IDL) nyelvekhez kapcsolódnak.

## 7. Összefoglalás

Az SDL és az UML a legelterjedtebben használt szabványosított modellező nyelv, profiljaikat számos területen alkalmazzák [6].

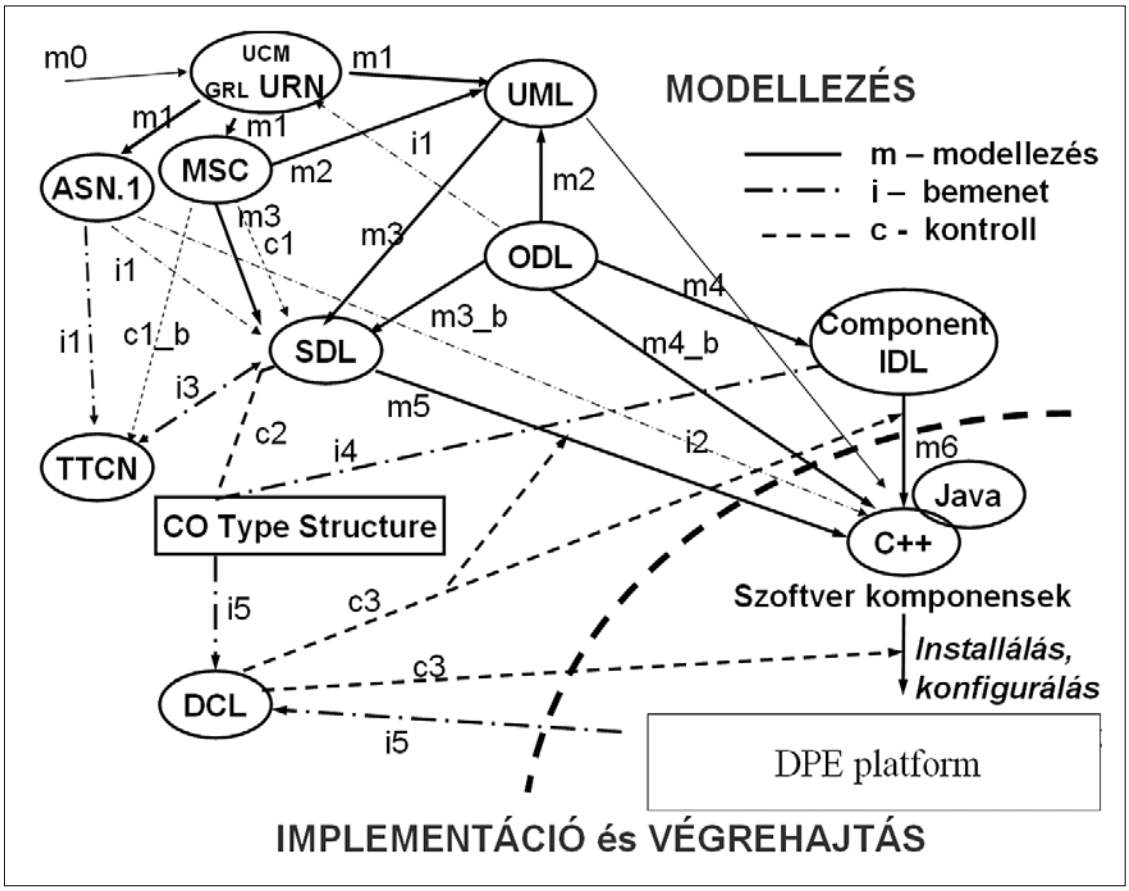
A SDL, a kommunikáló protokollok fejlesztésére létrehozott nyelvként, a távközlés fejlődésének hatására tartalmazza mindazon programozásnyelvi elemeket, amelyek alkalmassá teszik a rétegződés elvén szervezett és együttműködő-képességet fokozó architektúrák megvalósítására. A formálissága és adatdefiníciós eszköztára alkalmassá teszi bármely rendszer modellezésére. Több évtized alatt kifejlesztett validációs és verifikációs technikái beépültek a komponens-elvű szoftverfejlesztés technikáiba. Az UML objektumábrázoló elvein megerősített SDL specifikáló ereje fokozza a modellező nyelv terjedését az ipari hálózatok és beágyazott rendszerek fejlesztésében.

Az UML létrejötte és fejlődése, az OO programozástechnológiák elterjedésének hatására vált szükségessé, fejlődésének fő katalizálója a teljes fejlesztési ciklus lefedésének igénye. Az UML2.0 szabványba az MSC és SDL nyelvek beemelésével az MSC által a dinamikus elemzés, az SDL által az implementálás fázisok támogatása erősödött. Ugyanakkor, az SDL specifikációkra épített technológiák mind beemelhetők a fejlesztési ciklusba, így a távközlés világára kidolgozott tesztelési eljárások és tesztkörnyezetek is, megerősítve az UML nyújtotta követelményvalidálás automatizálásával. Ennek hozadéka a követelménytervezésből derivált tesztelési terv, indirekt módon pedig a szoftverfolyamat költségeinek csökkentése.

30 éves szerepével a távközlés és a modellezés fejlődésében, az SDL specifikáló és leíró nyelve napjainkra az ipari automatizálás egyik modellező eszközévé vált SDL-RT és ezSDL néven, jelenlétével az UML2.0-ban az úgynevezett beágyazott rendszerek fejlesztésére ad technikákat, a hálózattechnológiák és szolgáltatások összefonódásával szerepet kap az üzleti folyamatok és ügymenetek modellezésében is.

Az UML további fejlődését követhetjük az OMG keretében. Az UML modellező nyelv megerősítése, a kommunikáló automatákra alapozott formális specifikációval, felgyorsította a komponensalapú és eseményvezérelt programozástechnológiák alkalmazását, valamint a rendszerfolyamatok platformfüggetlen ábrázolásával az MDA paradigma kutatását és a SysML nyelv fejlesztését.

Az SDL további fejlesztését követhetjük az ITU-T és az SDL Forum keretében megvalósuló bővítésekkel, amelyek a (NGN) következő generációs hálózatok és a beágyazott hálózatok modellezésére súlyozottak, a folyamatok időbeliségének pontosabb ábrázolásával és tesztelésével.




*1. ábra*

*Az SDL kohéziós szerepet kap a távközlő rendszerek fejlesztésére szabványosított ITU-T formális nyelvek és az OMG UML szabvány egymásra épülésében [18]*

**Köszönetnyilvánítás**

Hálával tartozom Dr. Németh Gézának az elmúlt 25 év eszmecseréivel megvilágított szakmai irányvonalakért a programozástechnológiák és hálózattechnológiák világában, az „ezt kellene olvassad” kísérőszavú Ashton-Tate, Wirth, Kernigham-Ritchie, Angszter, Dijskra könyvekért és CEBIT-beszámolókért.

Ugyancsak sok köszönettel adózom többek között Dr. Kozmann György tanításaiért és támogatásáért, valamint a Híradástechnika főszerkesztőjének tanácsaiért és segítségéért a cikk véglegesítésében.

A téma kutatását a PE MIK és az IRIT Toulouse intézmények együttműködésében, a TÉT Alapítvány F-16/04 magyar-francia kormányközi szerződése támogatja.

**Irodalom**

[1] Tarnay K.
Kommunikációs protokollok modellezése és konformancia vizsgálata.
Doktori értekezés, 1990.

[2] www.itu.int.org/recommendation/zseries/languages

[3] Pataricza A.,
Formális módszerek az informatikában,
Typotex, Budapest 2004.

[4] http://www.sdl-forum.org; www.sdl-forum.org/tools

[5] Medve A.,
SDL – a kommunikációs folyamatmodellek egyik szabványosított implementációs eszköze,
Híradástechnika 2005/10.

[6] OMG.org, ISO.org, ITU.int.org., IEC.org, IEEE.org, CEN.org, WHO.org

[7] Gy. Csopaki, K. J. Turner,
Modelling Digital Logic in SDL.
In Proc. Formal Description Techniques X/Protocol Specification, Testing and Verification XVII,
Chapman and Hall, London, UK, November 1997.

[8] S. Dibuz
Testing protocols,
Application of Formal Description Methods,
Publ. VEAB,Veszprém, chapter 5.; 2001.

[9] www.telelogic.com, SysML.org

[10] ITU-T Recommendation Z.100 (1995)
“Specification and Description Language (SDL)”

[11] ITU-T Recommendation Z.100 (1999)
“Specification and Description Language (SDL)”

[12] ITU-T Recommendation Z.105 (1995)
“SDL Combined with ASN.1 (SDL/ASN.1)”

[13] ITU-T Recommendation Z.109 (1999)
“SDL Combined with UML (SDL/UML)”

[14] ITU-T Recommendation Z.120 (1999),
Message Sequence Chart (MSC)

[15] ITU-T Recommendation X.680 (1994)
“Data Networks and open System communications – OSI networking and system aspect –
Abstract syntax Notation One (ASN.1)”

[16] ITU-T Recommendation X.292 (1998),
Z.140-Z.149 (2003):
“OSI conformance testing methodologhy and framework for protocol Recommendation for ITU-T applications – The Tree and Tabular Combined Notation (TTCN)”

[17] ITU-T Recommendation Z.150-153.(2003),
“User Requirements Notation” (URN),
“Use Case Map” (UCM),”
Goa-Requirements Language” (GRL).

[18] Medve A.,
A formális módszerek szerepe a távközlési szoftverek fejlesztésében,
Networkshop 2001, www.niif.hu/networkshop

[19] D. Latella, I. Majzik, and M. Massink,
Automatic verification of UML statechart diagrams using the SPIN model-checker.
Formal Aspects of Computing, 11(6):637–664, 1999.

[20] D. Varró,
Automated formal verification of visual modeling langauges by model checking.
Software Systems Modelling, 3:85–113, 2004.

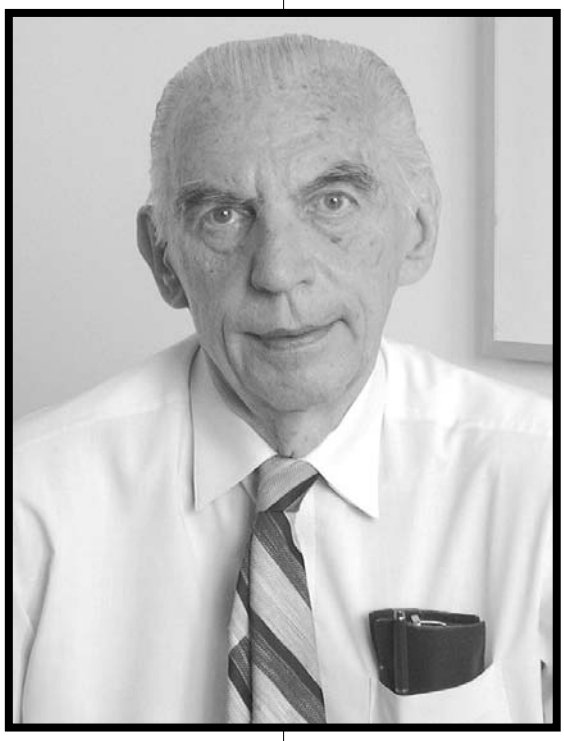
[21] www.omg.org/mda ,
www.omg.org/uml

[22] Raffai Mária,
UML 2 Modellező nyelvi kézikönyv,
Objektumtechnológia sorozat 4. kötet,
Palatia Nyomda és Kiadó, 2005.

# Búcsú Géher Károlytól (1929-2006)

*Szomorú szívvel búcsúzunk Dr. Géher Károly Professor Emeritus-tól, sokunk kedves Karcsijától, aki egy nappal 77. születésnapja után, 2006. augusztus 14-én elhunyt. Nem hagyott itt bennünket – hisz sokunkban tovább él, de meg kell tanulnunk azt, hogy hogyan kérjük ki, értsük, fogadjuk és köszönjük meg annak a tanácsait, aki már soha nem szól a megszokott szabatosságával hozzánk, s hogyan mutassuk ki tiszteletünket és szeretetünket annak, aki soha nem néz már ránk jóindulatot sugárzó szemeivel...*

Az angolul és németül folyékonyan beszélő Géher Károly 1947-ben iratkozott be a Budapesti Műszaki Egyetemre és 1952-ben szerzett villamosmérnöki oklevelet. Tudományos pályája páratlanul gyorsan ívelt felfelé: 1962-ben már a tudományok kandidátusa (mikrohullámú rendszerek csoport-futási ideje témájú disszertációval), erre alapozva 1963-ban a Dr. univ. cím birtokosa és 1973-ban a lineáris hálózatok érzékenysége témában írt disszertációjával a tudományok doktora. A Magyar Tudományos Akadémia 1993-ban az Eötvös József Koszorú adományozásával örökös véleménynyilvánításra kérte fel a műszaki tudományok minden dolgában.



Mindezek a tudományos teljesítmények és elismerések elválaszthatatlanok Dr. Géher Károly kimagaslóan érdemdús és eredményes egyetemi oktatói tevékenységétől.

Diplomájának megszerzése után a tanárfejedelem, Dr. Simonyi Károly tanszékén kezdte meg oktatói pályafutását, majd 1957-ben jelenlegi munkahelye (BME, Távközlési és Médiainformatikai Tanszék) jogelődjére kerül és a „Lineáris hálózatok” általa klasszikussá tett tantárgyával új rendszerezési elvet és ezen keresztül új pedagógiai módszert vezet be. Egy új világot teremt témájában mintegy 40 évre, műgondjában és szerkesztésében – azt hiszem örökre.

Mintegy két évtizedes kapcsolata az iparral, elsősorban a Távközlési Kutató Intézettel, ahol 1957 és 1967 között mellékállásban tevékenykedett, segítette Őt abban, hogy ne csak rendszerező és/vagy alkotó elefántcsonttoronybeli tudósként, hanem a gyakorlati alkalmazásokra is figyelmet fordítva művelje szakmai érdeklődésének tárgyát. Sokan – közülük azóta elismertté vált tudósok is –, a rendszerezés atyamesterét tisztelik benne. És persze a kitűnő előadót. Kiváló kutatói-oktatói teljesítményét több országos elismerés is jelzi, így például a „Kiváló pedagógus” kitüntetés 1991-ben, vagy a Szent-Györgyi Albert díj 1998-ban.

Ő a soha el nem évülő érdemű hármas egyik tagjaként részese a Microcoll magyar színhelyű konferenciasorozat koncepcionálásának, 1959-es elindításának, s ezzel Magyarország visszahelyezésének a II. világháború utáni híradástechnika világtérképére. Több mint harminc éven át kimagaslóan sokat tett e konferencia-sorozat tartós sikeréért.

Amikor a klasszikus hálózatelmélet rendszerezését befejezte, felismerte a hálózatelmélet új kihívását: a tolerancia analízist. Ezt a kérdéskört tudományos igénnyel világviszonylatban először Dr. Géher Károly állította középpontba és az első leglényegesebb válaszokat is Ő fogalmazta meg három idegen nyelven is megjelent könyvében. Máig a téma első és legelismertebb „új klasszikusaként” tartják számon világszerte.

Munkatársait mindig szelíd erőszakkal ösztönözte a tudományos előrehaladásra, melyhez mindenkinek minden segítséget megadott.

Egy ilyen tudós híre – hála Istennek – nem marad meg országhatárok között. Ezt illusztrálja az is, hogy az Institute of Electrical and Electronics Engineers (IEEE) Fellow-ja, az URSI (Union Radio Science International) Magyar Nemzeti Bizottságának 1966 és 1988 között tagja, titkára majd elnöke, s az URSI nemzetközi szer-

vezetében más funkciók mellett a szakmai C szekciónak (Signals and Systems) is elnöke volt.

1959 óta tagja volt a Hírközlési és Informatikai Tudományos Egyesületnek (HTE) is. Tagja, de nem „mindennapi" tagja. Tanúságul álljanak itt legfontosabb egyesületi díjai: a Pollák-Virág díj (1961), a Puskás Tivadar díj (1967 és 1997), a HTE Aranyérem (1989) és a HTE 50 éves jubileumi díj.

A fenti díjak a HTE-ben és azon kívül kifejtett tudományos-szakmai teljesítményeit egyaránt elismerik. Általános tájékozottságával és hallatlanul magas erkölcsiségével nagy tiszteletet kiváltóan munkálkodott a HTE nem szakma-specifikus, hanem általános értelemben vett értékeinek gyarapításán is. Közmegbecsülést kiérdemlően volt a Külügyi Bizottság, a Díjbizottság és az Etikai Bizottság elnöke, s haláláig az Elnökség választott tagjaként segítette az egyesület tevékenységét.

A fenti tisztségekben kifejtett feddhetetlen és konstruktív szerepe mellett országos jelentőségű feladatot is vállalt. Az 1992-es távközlési törvény létrehozta a Távközlési Mérnöki Minősítő Bizottságot, amelynek az előkészítésében elévülhetetlen érdemeket szerzett. E bizottságnak miniszteri kinevezéssel a lehetséges két cikluson át, 1993. július 1-től 1999. június 30-ig első elnökeként tevékenykedett.

1998-ban a sokoldalú, és minden irányban kimagaslót alkotó tudós mérnököt a Széchenyi díjjal tüntették ki.

Dr. Géher Károly Professor Emeritus-t a Budapesti Műszaki és Gazdaságtudományi Egyetem saját halottjának tekinti.

Az a megtiszteltetés ért, hogy temetésén én is elbúcsúzhattam Tőle. Személyes, irányában érzett mély tisztelem és őszinte szeretetem kifejezésére a temetési búcsúztatáson elmondottak néhány gondolatával zárom e nekrológot:

*Nagyon szeretett, de viszontszeretni már nem tudó, őszintén tisztelt, de ezt a Tőled megszokott visszafogott megelégedettséggel nyugtázni már nem tudó kedves Karcsi!*

*A jelenlévők közül talán én vagyok az a szerencsés, aki tanítványodként és munkatársadként a leghosszabb időt töltöttem Veled, ezért most szavaimat Te hozzád intézem.*

*Egy közös barátunkat megkérdeztem, mit javasol, mire térjek ki búcsúztatómban. Ő Shakespeare-t idézve tanácsolta, hogy mit tegyek: „Temetni jöttem, nem dícsérni". Dicsérnek tetteid, a könyvtárak, a nekrológok, visszaemlékezések. Mindez azonban nem teljes, mert csak a ráció hangján beszél. Az élet pedig a ráció és emóció síkjain zajlik. Te emóciót is ébresztettél munkatársaidban, mégpedig sokszor és mindig pozitívat és mi is Benned olykori – segítségeddel elért – sikereinkkel, vagy éppenséggel az elvárásodhoz képesti lassúságainkkal.*

*Kedves Karcsi! Örökre hiányozni fog emelkedett higgadtságod, ösztönzésed kifinomultsága, jóságod, emberséged.*

*1957-ben két alkalommal vizsgáztam Nálad. Emlékem a vizsgáról igazolta híredet, miszerint, hogy „a Géher jó szándékú". Ma már tudom, hogy ehhez szív is kell. És ez a szív is mindíg megvolt Benned.*

*Életed alatt világégések, forradalmak, rendszerváltások írták a történelmet. Nagyon sokan tiszteljük és próbáljuk magunkévá tenni azt a páratlanul pártatlan higgadtságot, amellyel Te mindezt követted.*

*Kedves Karcsi! El kell temetnünk Téged. De jelen vagy és jelen leszel kisugárzásoddal. Példát adtál emberségből, jóindulatból, kolosszális rendszerező képességből, oktatói rátermettségből, iskolateremtésből, csapatépítési tehetségből, diákjaid és munkatársaid előrehaladásának fáradhatatlan és diszkrétségével hatékony szorgalmazásából.*

*Példát adtál új diszciplinát teremtő tehetségedből, munkatársaid egymást tisztelő, egymást segítő, vidám hangulatot keltő összejöveteinek szervezéséből és a mindezek mögött álló – általad kinyilvánítani nem szándékolt, de a mi számunkra félreismerhetetlen – mély házastársi szeretetből.*

*Kedves tanítóm, munkatársam, kedves Karcsi! Kifejezem sokunk mély részvétét a munkádban mindvégig észrevehetetlenségre törekvő, de mégis észrevehető, számodra biztos családi hátteret teremtő kedves hitvesednek, Judit Asszonynak.*

*Kedves Karcsi! Nem frázis, színtiszta valóság, hogy akik ismertünk, életed tanulságait magunkba zártuk, és általad nemesedtünk. Ha bennünket tisztelnek, Téged is tisztelnek, ha nem –, mi rontottuk el.*

*Büszkék vagyunk Rád!*

*Nyugodj békében.*

*Gordos Géza*

**Trends in modern software development: integrated testing**
*Keywords: software development, testing, framework, TITAN, TTCN-3*
In this paper, we give an overview of a typical development process of complex software systems and define the requirements toward test solutions that should improve the efficiency of the development. Afterwards we analyse to what extent TTCN-3 meets these requirements.

**Testing in telecommunications**
*Keywords: testing, TTCN, CSP, standardized test*
Checking, control and testing functions can often be seen in everyday life. Communication between machines has not reached this level, yet. For this, a highly developed Artificial Intelligence is needed. Technical innovations of the future might realize Self-Adaptive systems operating with AI. But now, we have only one chance to work our systems in this area and it is the standardized test. Telecommunication uses a lot kind of test. Some of those are for example the conformance, the performance and the interoperability test. TTCN can solve the problem of test and it is the only one standardized test tool in this area at the present time.

**Testing telecommunication software**
*Keywords: software development life cycle, functional testing, performance testing, test automation*
The paper summarizes the test activities that are done during the telecommunications software development. It describes what kinds of tests is necessary to be done and how testing is contributing to the better quality of the software product. The resource needs of the different test types are also discussed and a brief summary of test automation is included with motivations on its usage.

**Overview of conformance and interoperability testing**
*Keywords: interoperability test, conformance test, ROHC*
The main goal of this paper is to present the nature and types of interoperability testing. Besides, it also mentions conformance testing because both types of testing are important and it is easier to understand the identical and different parts of them. Moreover, it is not possible to get a full picture of a product or protocol implementation based on just only one type of testing methods, both interoperability and conformance testing are needed.

**Load testing using distributed test components**
*Keywords: TTCN-3, load test, parallel test components*
An evaluation method is presented that can be applied on Testing and Test Control Notation version 3 (TTCN-3) based performance and load test software components. Our aim is to aid the development of load tests by predicting the performance critical behavior of test components in the early design phases. We, mainly focus on the race conditions of the queues underlying the implemented test components as well as on other TTCN-3 specific issues that may contribute to possible delays in the test system.

**TITAN, TTCN-3 test execution environment**
*Keywords: testing, TTCN-3, test system implementation*
This paper presents the TTCN-3 test execution environment of Ericsson, called TITAN. We show the internal details and the operation of the toolset. Unique TITAN features and differences from other commercial TTCN-3 tools are discussed. As a result of our development TTCN-3 and TITAN has become a widely used test solution within Ericsson and we have the possibility to contribute the TTCN-3 standardization work within ETSI.

**Model based testing of component systems**
*Keywords: domain specific modeling, component based system, Deployment Tool*
The growing demand of the telecommunication market for complex systems cannot be easily satisfied without new development paradigms. Model based development with Domain Specific Modeling Languages (DSML) offers a good way to achieve the desired complexity management and the transformation to our component based platform (ErlCOM) provides a good example to show all the capabilities of this method. In this paper we summarize the different testing methods and present the Deployment Tool, our model based component testing solution.

**Mobile Peer-to-Peer client software based on a semantic protocol**
*Keywords: semantic Peer-to-Peer information retrieval, software development for mobile devices*
With the spread of broadband wireless communication, the network applications move from desktop computers to mobile devices. Such an important application is the fully distributed information sharing client, the Peer-to-Peer application, because of its versatile usability. However, the relative high cost of wireless communication requires the use of advanced protocols with small generated network traffic which tolerate the strong transient property of these networks. In this paper, we will introduce the first Symbian operating system-based Peer-to-Peer client software, along with the semantic protocol developed for this environment.

**Linux in telecommunications**
*Keywords: Asterisk, VoIP, carrier-grade Linux, soft-phone, PBX, open source, SIP, H.323, ATA*
The goal of the article is providing information about the relationship of telecommication and open source software, especially Linux. While first part introduces the achievements of Linux in the field of carrier-grade systems the second part focuses on the open source PBX and VoIP solutions, Linux is able to offer. The second part also introduces the building blocks of a SOHO PBX and a PBX specific, out-of-the-box Linux PBX distribution.

**SDL–UML cooperation: UML2.0**
*Keywords: SDL, UML, software modelling*
The importance of analysis and modelling phases in the software process of the complex IT system's development is generally known nowadays. These movements influence the evolution of the languages that support the early stage of a software process. Originally intended for documentation, specification languages have become more and more descriptive and the tools that support them with validation, verification and simulation are augmented in number and in their role. The automatic generation of test cases, simulation techniques make the detection of the model's faults possible in a very early stage of the development. We can develop the supporting tools only to an expressive and a well-defined supported langage. SDL (Specification and Description Language) and UML (Unified Modelling Language) are the most used modeling languages by the major part of industry design. These languages belong to the category of formal methods, SDL since its introduction, UML since its UML2.0 version. In this paper, we briefly introduce these two languages, we also present the variability of their applicability along showing their complementary aspects, furthermore we show the languages properties derived from the evolution of the telecommunications, that summarized into the convergence of these languages helping the support of new modelling paradigms.

**Summaries** • of the papers published in this issue

GLOBAL SUCCESS FROM LOCAL WISDOM

# Triple-play Drives Network Transformation

**1.Triple-play can promote telecom operators to obtain new profit.**

Recent years, the deep end in fixed-line communication market has emerged gradually because of the intensifying challenges from Internet and Mobile. As the fig.1, both of operators and suppliers have to face the same problem about how to keep from profit declining under the condition of saturated or being saturated fixed-line voice market.

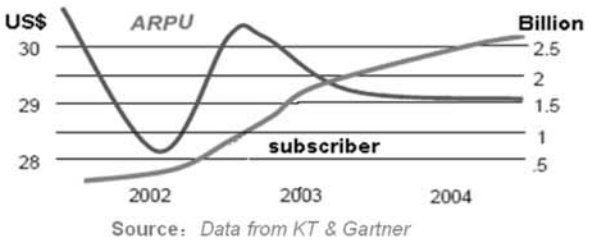



**Fig.1 Problem of fixed operators**

As we known, in the past 10 years, the gain mode of Telecom is correspondingly simple with its glancing management only pursuing subscriber increase because the cash flow of operators is plenitudinous. While the subscriber increase became slower, the first thing most Telecom operators considered is to seek the finance-driven increase. That means to pursue the growing scale and range based on the existing market to reduce cost and enhance growing speed. However, the basic gain mode of Telecom can not be changed in essence only through the way because it is just one layer of transformation. However, what is the essential change? More and more research shows that the transformation on service and operation mode might be the essential change which could bring new gain.

Nowadays, voice communication isn't only function for telephone, new multimedia services have brought a serial industry chains that relating to more providers besides the telecom operators. Then, by sharing the profit with other providers on the different nodes on the industry chain, the potential and covert operation scale might be magnified greatly with new growing.

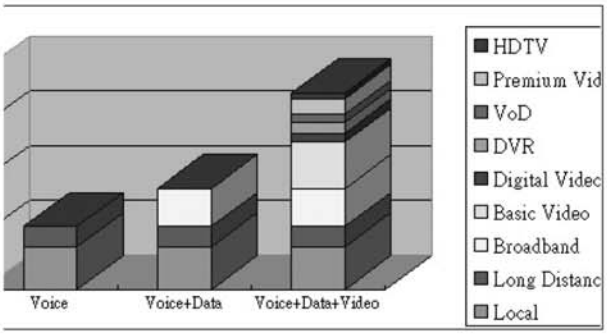



**Fig.2 Relative services of telecom**

However, another question is which kind of service provision mode will be the better way to form the larger industry chains for fixed-line operators? Triple-play mode might be an appropriate choice. Triple-play is becoming a popular topic since 2004. Triple-play means a service-converged experience for subscribers, which is integrated with VoIP(Voice over IP), Video, Data services features. As Fig2., the application research cases in US. show that Triple-play services will enhance ARPU (Average Revenue Per Unit), promote user loyalty and reduce 50% user losing ratio than VoIP only services, with differentiated and customized multi-service combination. Moreover, Triple-play services also bring better effect on ROI (Return on Investment) aspect. For example, for DSL network, the ROI is usually 5-6 years if only traditional services were provided, while it might be 2-3 years if video services were provided additionally.

Generally, Triple-play industry chain is composed of six parties: Content Providers, Content integration Providers, Network Operators, Platform Providers/Service Providers, Equipment Suppliers and terminal users. The Triple-play services refer to three branch-chain including Voice services, Video/IPTV services, broadband internet access services, the nodes on each branch are linked and related with other branch, that means more covert branch-chain are included in Triple-play mode, and it is more great and complex than any operation modes before. So Triple-play based operation mode might bring larger business scale and profit sharing space for fixed-line operators.

**2.Network requirement of Triple-play services**

Even though the advantages of Triple-play and more opportunities for the fixed-line operators, for the operators, it is still a question about how to make the various combinations of three kinds of services to meet the differentiated requirement from different user groups. And it is ales an important factor for successful deploying Triple-play to provide differentiated services from existing Cable/TV broadcast services. Moreover, traditional fixed network environment is still difficult to provide Triple-play services quickly and conveniently for its old architecture, un-opened service provision mechanism and limited bandwidth resource etc. That means, in order to provide Triple-play services, network transformation is inevitable to suffer for the operators.

Therefore, the network technology developing is necessary for Triple-play services provision. And following points might be the key to deploy Triple-play services:

Converged and opened service platform: For the Triple-play operators, following features are very important to increase ARPU and reduce operation cost:

1.Integrated Subscribers management and authentication
2.Open services provision to third part
3.Unified service management and service security
4.Unified Charging and Billing
5.Control-enable end to end QoS
6.Multiple service blending and fast deployment

As the bottleneck of whole network, following features are necessary for Triple-play user access:

1.Broadband features: As the most important service feature, video-related services (especially, IPTV service) will use up largest bandwidth resource than other twos. On the equipment side, enough uplink bandwidth and user bandwidth are necessary. And, on other aspect, it is also important to choose appropriate video codes which can occupy as less as possible bandwidth under definite signal quality.
2.Multicast features: For the IPTV large scale application, the controllable multicast capability of access network is mandatory to comply. And it also shall be considered to choose the appropriate position for control and replicate point.

**3.ZTE F3G total solution helps operators to realize network transformation**

In order to help fixed operators to transform successfully to Triple-play mode, ZTE launches F3G total solution focusing on how to provide Triple-play services effectively to users, how to resolve broadband access issues for Triple-play, how to build a unified and opened service provision platform and how to get to the target of constructing a converged network. The respective and optional sub-solution in different layers are contained in ZTE F3G solution to provide whole-network transformation consideration.

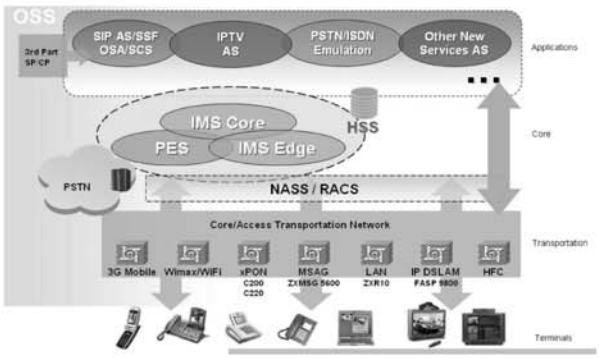



**Fig.3 ZTE F3G architecture**

As Fig.3, the architecture of ZTE F3G solution is composed of Applications, Core control modules, Transportation Network and terminals. The applications includes SIP AS(providing SIP based services), SSF(providing legacy Intelligent Network services), IPTV AS(providing IPTV services), PSTN/ISDN emulation service, and more services provided by third part CP(Content Provider)/SP(Service Provider). The core layer includes IMS(IP Multimedia Subsystem) and PES(PSTN Emulation Subsystem) that implement core session control functions for every service flow. The transportation network is to bear and switch all service flow between different accessed users, or between the Applications and terminals. It is a complete NGN architecture recommended by TISPAN.

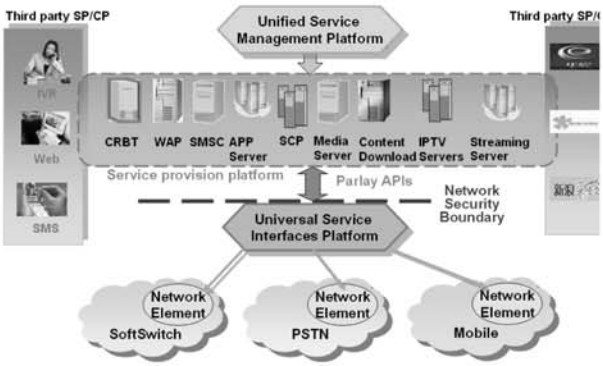



**Fig.4 Converged service platform of ZTE F3G**

The key technology features of ZTE F total solution are as following:

Converged service platform: As Fig 4., ZTE F3G solution provides a converged service platform to deliver the combined services including multimedia services, data services, streaming services, legacy voice/IN services, IPTV services, and 3G services. It is composed of unified service management, unified service provision, unified service interface and integrated user management platforms to support flexible and combined services deploying, unified service management, integrated authorization and charging etc. It is opened for third part CP(Content Provider)/SP(Service Provider) via Universal Service Interfaces Platform, and also is converged on CP/SP, Service, Content, and Subscriber management via Unified Service Management Platform.

Unified core control: It is fully complied with TISPAN IMS (IP Multimedia Subsystem) architecture that could provide unified session or non-session control for every service flow. It also provides complete QoS control for any kinds of service flow.

Multi-service broadband access technologies:

1."GE(Gigabit Ethernet) to slot" technique of ZTE access equipments make it is possible to provide GE-Level non-blocking architecture, at least 1 Gigabit Date Bus for per slot, multiple GE uplinks (4Gb/s), non-blocking more than 13 Mb/s per subscriber bandwidth.
2.Multicast control and replication capability of ZTE access equipments.
3.FTTx(Fiber to the x) via xPON(Passive Optical Network) sub-solution: Currently, FTTx networks via GPON/EPON techniques are becoming a trend of wire-line access networks for its larger bandwidth resource, longer transport distance, long use-period, higher QoS guarantee. ZTE FTTx via xPON sub-solution completely supports Triple-play services with its fully end-to-end QoS and security guaranteeing, as Fig. 5.
4.Hybrids solution: This method can optimize the access network, for the last passage existing twist pairs will keep being used, only the primary pairs will be replaced with fibre optical. This is the best option for most fixed-line operators

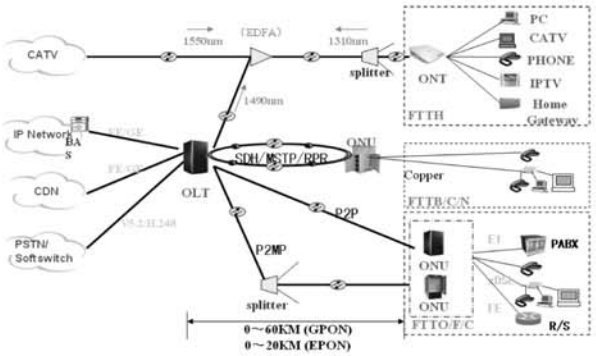



**Fig. 5 ZTE FTTx via xPON solutions**

Moreover, ZTE F3G solution also provide smooth migration proposal for fixed-line operators.

With ZTE F3G solution, it is expected that a perfect industry chain could be generated to establish multi-win relationship among operators, ZTE, and other cooperators.

# Contents



*Cover: An old telephone exchange (from the collection of Museum of Post and Telecommunications, Budapest)*

**Szerkesztőség**
HTE Budapest V., Kossuth L. tér 6-8.
Tel.: 353-1027, Fax: 353-0451, e-mail: info@hte.hu

**Hirdetési árak**
*1/1* (205x290 mm) 4C 120.000 Ft + áfa
*Borító 3* (205x290mm) 4 C 180.000 Ft + áfa
*Borító 4* (205x290mm) 4 C 240.000 Ft + áfa

**Cikkek eljuttathatók az alábbi címre is**
*Szabó A. Csaba,* BME Híradástechnikai Tanszék
Tel.: 463-3261, Fax: 463-3263
e-mail: szabo@hit.bme.hu

**Előfizetés**
HTE Budapest V., Kossuth L. tér 6-8.
Tel.: 353-1027, Fax: 353-0451
e-mail: info@hte.hu

**2006-os előfizetési díjak**
*Közületi előfizetők részére:* bruttó 30.450 Ft/év
*Hazai egyéni előfizetők részére:* bruttó 6.800 Ft/év
*HTE egyén tagok részére:* bruttó 3.400 Ft/év

**Subscription rates for foreign subscribers:**
12 issues 150 USD,
single copies 15 USD

www.hte.hu
Felelős kiadó: NAGY PÉTER
Lapmenedzser: DANKÓ ANDRÁS

HU ISSN 0018-2028
Layout: MATT DTP Bt. • Printed by: Regiszter Kft.