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.

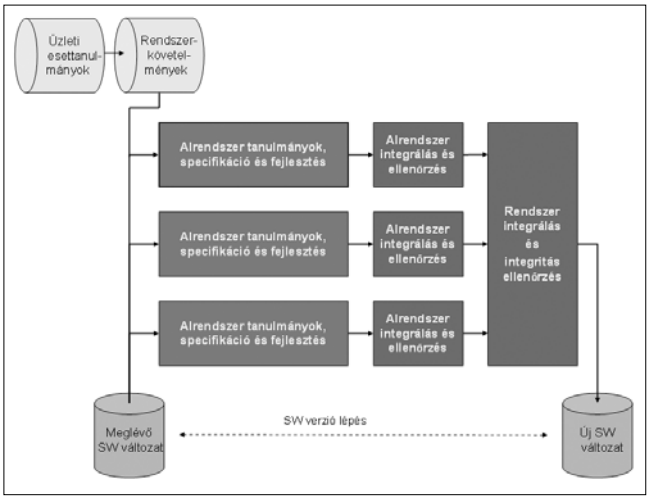
2. A szoftverfejlesztés folyamata

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.

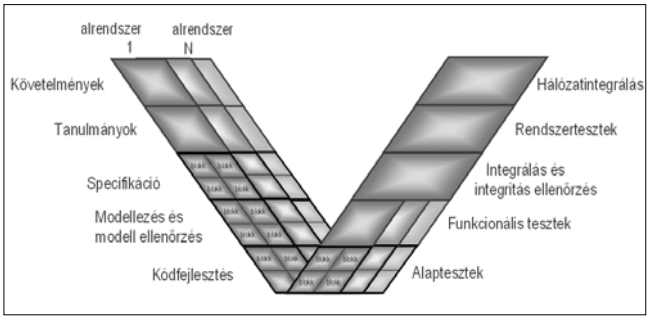
Ezért az üzleti esettanulmányokat először az egyes rendszerekre vonatkozó követelményekre kell lebontani, majd ezeket leképezni az érintett alrendszerek által teljesítendő követelményekre *(1. ábra)*. Ezután lehet az egyes alrendszerekben elvégezni a szükséges szoftverfejlesztéseket. Az egyes alrendszerek 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 Δ 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 Δ funkcionalitást megvalósítani egy működőképes közbenső szoftver változat (jelöljük pl. $SV_{n.a}$ -val) előállításához? Például, ha Δ 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 δ_j jelöli az egyes, önállóan megvalósítható részfunkciókat.

Természetszerűleg

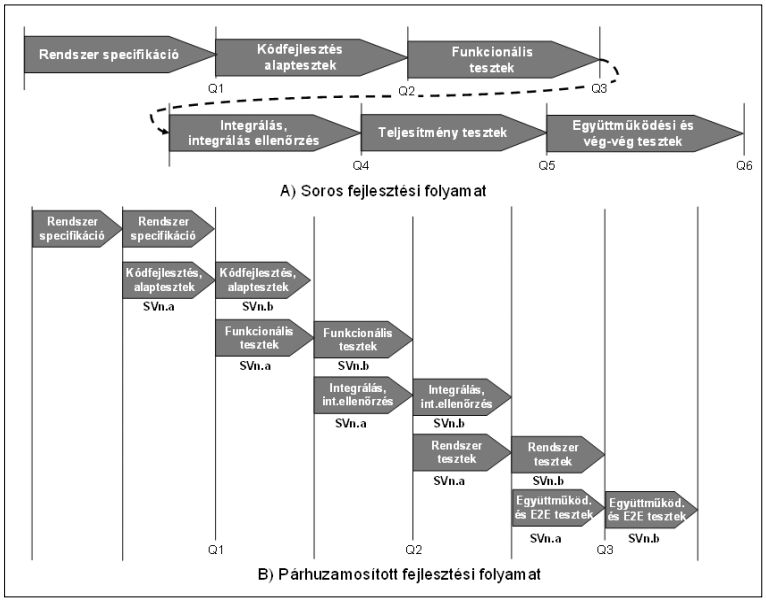
$$\Delta = \sum_{1}^{i} \delta_j \ .$$

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.

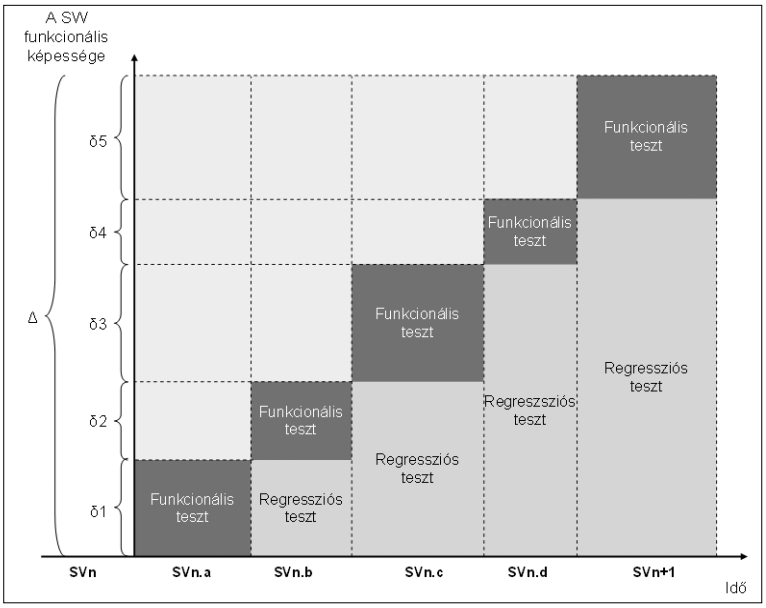
Az ICE alkalmazása azonban új probémákat is felvet. A gyakorlatban az ICE folyamat egyes ütemeiben fejlesztett funkciók, δ_k és δ_l nem teljesen függetlenek egymástól, a δ_k-hoz fejlesztett szoftver blokkok valamilyen mértékben módosításra szorulhatnak δ_l fejlesztése alatt. Így a későbbi fejlesztési 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 δ_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.

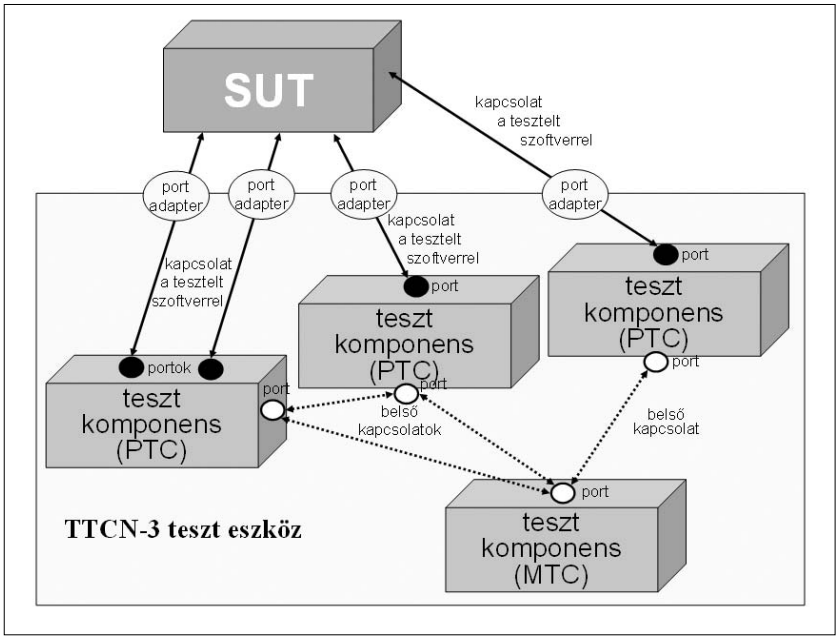
Itt kell megjegyeznem, hogy az Ericsson Magyarország a TTCN-3 teszt eszközök fejlesztésében a világon élenjáró eredményeket ért el. E műhelyből került ki a világ első működő TTCN-3-as teszteszköze, melyet TITAN névre kereszteltek. Ismereteink szerint a TTCN-3-at használó cégek között az Ericsson-ban 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.

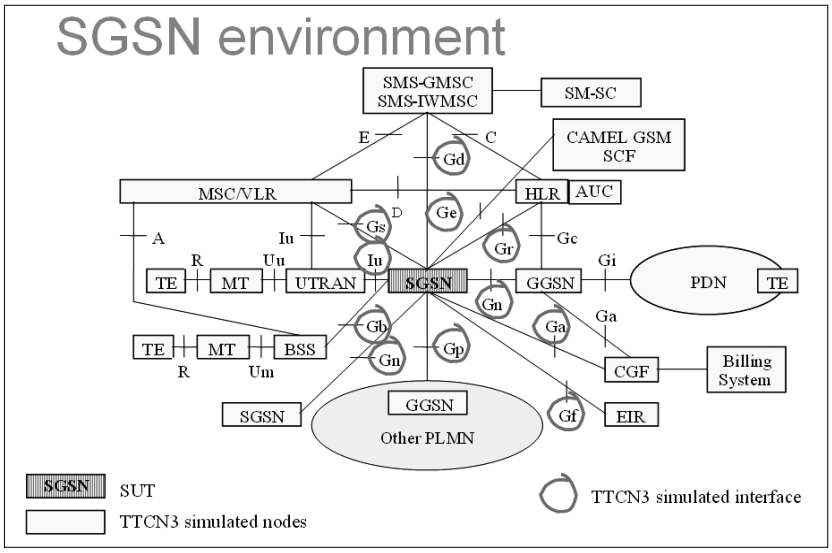
A TTCN-3-at nem egyszerűen automatizált teszt végrehajtásra, hanem felügyelet nélküli automatizált tesztelésre tervezték. Ennek egyik példája, hogy a TTCN-3 tesztkészletekben nem csak a teszteseteket, de azok végrehajtásának módját is megadhatjuk. Vagyis például egy teszteset végrehajtásakor kapott ítélettől 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.

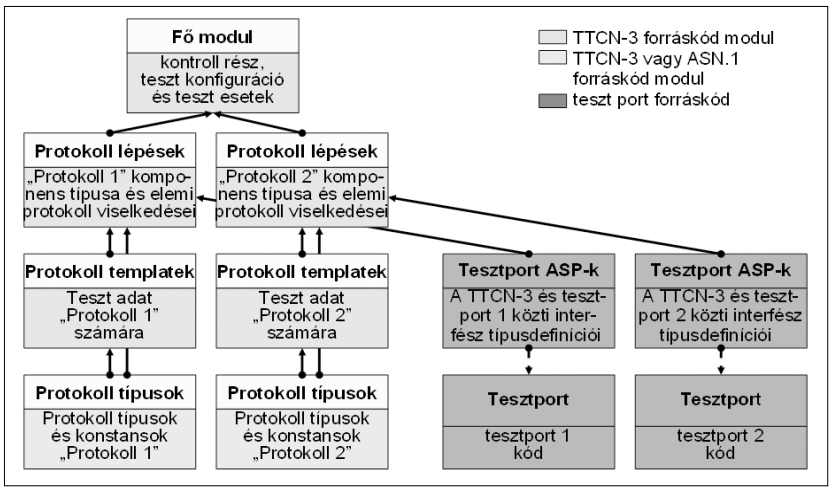
Mivel a TTCN-3 elsődleges formája ASCII szöveg, ennek tárolása, újrafordítása és futtatása valamint felhasználása más tesztkészletekben – a fájlformátum szempontjából – nem jelent problémát. A TTCN-3-ban végrehajtott funkcionális tesztek a regressziós vizsgálatok alatt gyorsan újrafuttathatók. Ezt segíti 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*.

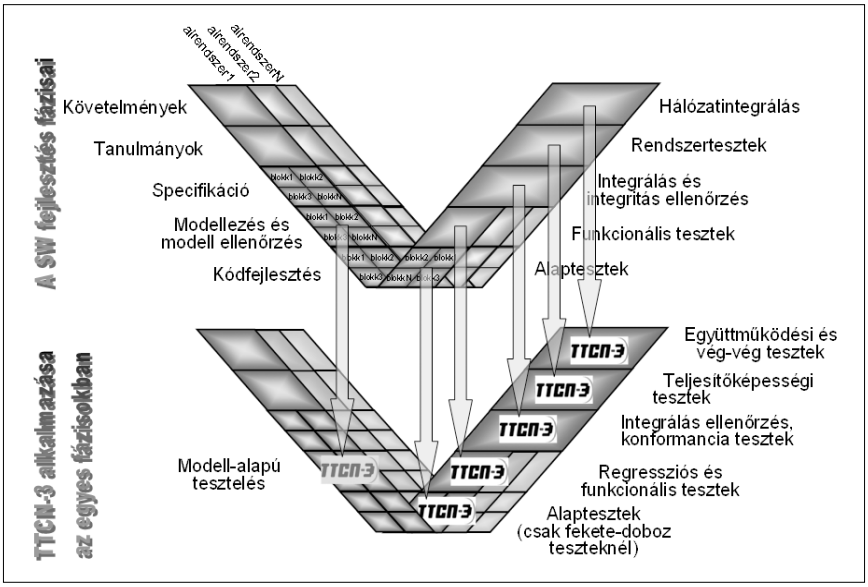
Mivel a tesztesetek rendszerenként eltérőek, tipikusan a tesztportok és protokoll modulok változtatása nélkül, a protokoll template és protokoll lépés modulok kisebb változtatásokkal újrahasználhatók.

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.