Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

AUTOSAR szoftvermodulok megfelelőségi tesztelése

DIPLOMATERV

Készítette
Csák Máté Gábor

Tanszéki konzulens
dr. Sujbert László
BME-MIT

Külső konzulens
dr. Pintér Gergely
ThyssenKrupp Presta Hungary Kft.

2012. december 7.

HALLGATÓI NYILATKOZAT

Alulírott *Csák Máté Gábor*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2012. december 7.

Csák Máté Gábor
hallgató

Tartalomjegyzék

Kivonat

Az autóipari beágyazott szoftverek fejlesztése terén az utóbbi évtized egyik legfontosabb törekvése a szoftverarchitektúrák egységesítése, szabványosítása. Ebből a célból jött létre 2003-ban az AUTOSAR konzorcium, amely egy világméretű együttműködés autógyártók és beszállítók között. Elsődleges célja, hogy specifikáljon egy alapvető szolgáltatásstruktúrát, szem előtt tartva a szoftver újrafelhasználhatóságának biztosítását az egyes modulok közötti interfészek szabványosítása révén.

Egy AUTOSAR szabvány alapján feljesztett szoftvermodul megfelelőségének tanúsításához elengedhetetlenül szükséges olyan tesztek elvégzése, amelyek igazolják, hogy a specifikációban megfogalmazott statikus és dinamikus követelményeknek eleget tesz. Ezen tesztek implementálására és futtatására számos lehetőség létezik, amelyek közül pénz- és erőforrás-takarékosság szempontjából ki kell választani az optimális módszert.

A diplomaterv bemutatja a kapcsolódó tesztelési módszereket és az elérhető megvalósításokat, valamint egy, az AUTOSAR által specifikált megfelelőségi teszt csomag és egy saját, C-alapú teszt implementáció futtatását egy AUTOSAR Basic Software modulon.

Ennek kapcsán összefoglalja a megfelelőségi tesztelés alapjait, tisztázza a kapcsolódó fogalmakat, bemutatja a tesztelendő modult, annak helyét és szerepét az AUTOSAR szoftverarchitektúrában, valamint a TTCN-3 nyelv azon sajátosságait, amelyek relevánsak a modul tesztelése szempontjából. Összehasonlít két, széles körben használt TTCN-alapú eszközt és bemutatja a CUnit nevű tesztkörnyezetet. A dolgozat ismerteti hogyan illeszthető a futtatókörnyezet a tesztelni kívánt modulhoz TTCN, illetve CUnit esetén, valamint részletezi a tesztelés folyamatát, elemzi és értékeli annak eredményeit, majd vázolja a további fejlesztési lehetőségeket.

Abstract

In the past decade, one of the main goals in automotive embedded software development was to unify and standardize software architectures. The AUTOSAR Consortium was founded on this purpose in 2003 as a partnership of automotive OEM's (Original Equipment Manufacturer) and suppliers. Its major objective is to specify a basic structure of services, enhancing software reusablity by the standardization of module interfaces.

In order to declare the conformance of a software module developed according to an AUTOSAR standard, it must be verified that it fulfills the static and dynamic requirements of the specification. There are several methods for implementing and executing conformance tests, from which the optimal solution shall be chosen, considering financial and resource-related terms.

In this Thesis the corresponding testing methods and the available tools are presented first, followed by the description of the conformance testing process of an AUTOSAR Basic Software module via two different methods: the official AUTOSAR Conformace Test Suite of the module, which is implemented in TTCN-3 language and another test suite, implemented in C.

In this context, conformance testing basics are summarized along with the corresponding definitions. The System Under Test, its place and function in the AUTOSAR software architecture are presented, as well as the relevant properties of TTCN-3 language. In the following, two widely-used TTCN tools are compared and the CUnit testing framework is presented. The adaptation of the runtime environment to the System Under Test is described for both TTCN-3 and CUnit, as well as the conformance testing process, its results and the further development options.

Bevezető

A járműipar minden ágazatában évek, sőt évtizedek óta megfigyelhető tendencia az elektronikus alkatrészek térhódítása. Ez a megállapítás fokozottan érvényes az autóiparra, azon belül is a személygépjárművekre. A modern személygépjárművek biztonságtechnikai és kényelmi funkcióinak megvalósításában, környezetvédelmi jellemzőinek javításában stb. központi szerep hárul a számítástechnikai megoldásokra. Manapság egy közép- vagy felső-közép kategóriás személyautóban is közel száz elektronikus vezérlőegység (ECU) található, ami miatt számos fedélzeti kommunikációs sínből kell kialakítani egy megbízhatóan működő elosztott rendszert, amely komoly algoritmus- és kommunikációtervezési kihívást jelent.

A növekvő igények kielégítése érdekében a gépjárművek számítógépes rendszereinek folyamatosan növekvő számú és bonyolultságú feladatot kell ellátniuk. Ennek eredményeképp nagymértékben megnövekedett a gépjárművekben található elektronikus és számítógépes hálózatok komplexitási foka, amely indokolttá tette, hogy az egyes feladatköröket (pl. kormányzás, fékrendszer, szellőzés) jól definiálható alfunkciókba partícionálják, amelyek ellátásáért elektronikus vezérlő egységek (ECU-k) felelősek. Ezen elektronikus vezérlőkből álló elosztott rendszerek kézbentartására kialakultak különféle szabványok, pl. a megbízható kommunikáció biztosítására a CAN és FlexRay sínek, a valósidejű feladatok futtatására az OSEK operációs rendszer vagy a futási idejű monitorozást támogató XCP protokollcsalád.

Az AUTOSAR konzorcium egy világméretű együttműködés autógyártók, valamint elektronikai, félvezetőipari és szoftver beszállítók között. Elsődleges célja, hogy specifikáljon egy alapvető *szolgáltatásstruktúrát*, amely eltakarja a hardver sajátosságait a magasabb rétegek elől és támogatja az alkalmazási szoftver portolhatóságát (hordozhatóságát). Fő alapelvei a *modularitás*, *skálázhatóság* és *újrafelhasználhatóság*, valamint az egyes modulok közötti *interfészek szabványosítása*. Ezen célok megvalósításának érdekében specifikál egy modellezési nyelvet az ECU-kon futó alkalmazási szoftver szabványos leírására, valamint egy futási idejű környezet (Runtime Environment - RTE), amely az AUTOSAR szoftverkomponensek számára megvalósítja a kommunikációs mechanizmusokat egy adott ECU architektúrán [1].

A konzorcium, bár jelentős hangsúlyt fektet az API-k szabványosítására, kifejezetten támogatja a versengést az egyes szolgáltatások megvalósításában („Cooperate on standards, compete on implementation”), megteremti továbbá az egyes modulok optimalizálásának lehetőségét a hozzájuk tartozó különböző konfigurációs beállítások segítségével [1].

A különböző gyártóktól származó BSW[1] modulok integrálhatósága érdekében szükség van arra, hogy az API *statikus* részein túl (pl. adattípusok, függvény szignatúrák, stb.) a modulok *dinamikus* viselkedése is megfeleljen a szabványban szövegesen vagy interakciós diagramokon bemutatott működésnek – ebből a szempontból pl. azt kell tehát ellenőrizni, hogy egy magas szintű funkciót megvalósító modul megfelelő sorrendben és paraméterekkel hívja az alacsonyabb szintű modulok szolgáltatásait, kezeli azok válaszait stb.

Mivel a szabvány szöveges leírása önmagában értelmezési félreértésekre adhat lehetőséget, a szabványhoz csatolt mellékletek számos modulhoz definiálnak végrehajtható megfelelőségi teszteket (conformance testing) TTCN-3 specifikációs nyelven. A TTCN-3 nem közvetlenül futtatható programozási nyelv, önmagában nem tud kapcsolódni egy AUTOSAR modul megvalósításhoz, ezért végrehajtásához ki kell alakítani egy tesztelő környezetet és meg kell írni a szükséges (adatkonverziót, nyelvi átjárást stb. biztosító) illesztő rutinokat (glue code).

[1]Basic Software - Szabványosított szoftver réteg, amely a futási idejű környezet alatt helyezkedik el és az AUTOSAR szoftver komponensek számára biztosítja a funkcionális szoftver futtatásához szükséges szolgáltatásokat.

1. fejezet

A megfelelőségi tesztelés alapjai

A mérnöki gyakorlatban használt tesztelés kifejezés számos, többé-kevésbé hasonló értelmezésének és definíciójának lényege tömör, általános formában úgy ragadható meg, ha a tesztelést egy olyan folyamatnak tekintjük, amelynek segítségével igazolható, hogy egy termék vagy folyamat a teszt által specifikált körülmények (bemenetek, gerjesztés) esetén bizonyos előírásoknak eleget tesz, megvalósít egy adott funkciót, valamint amelynek segítségével a hibák, hiányosságok felderíthetők. A tesztelési alapvető szintjei az alábbiak [2]:

- **Komponens teszt**: a rendszer legkisebb önállóan tesztelhető egységeit egymástól elkülönítve vizsgálja.
- **Integrációs teszt**: célja az integrált egységek közötti interfészekben, illetve köcsönhatásokban lévő hibák megtalálása.
- **Rendszerteszt**: az integrált rendszer tesztje abból a célból, hogy ellenőrizzük a követelményeknek való megfelelőséget.
- **Átvételi (acceptance) teszt**: a felhasználó vagy megrendelő által a végtermékén végzett feketedoboz teszt, amely azt hivatott eldönteni, hogy megfelel-e a termék a megfogalmazott elvárásoknak.

A diplomaterv egy AUTOSAR szoftvermodul *komponens* tesztjét mutatja be. A nemzetközi szakirodalom a tesztelés alatt álló termékre a tesztelés szintjétől és céljától függetlenül általában egységesen az absztrakt System Under Test (SUT) kifejezéssel hivatkozik, amely jelölhet akár egyetlen komponenst (pl. kommunikációs driver), akár egy összetett rendszert (pl. AUTOSAR Basic Software stack).

A megfelelőségi vagy teljesítési teszt (conformance testing) célja a SUT azon képességének vizsgálata, hogy megfelelő funkciókat biztosítson meghatározott feladatokhoz és felhasználói célokhoz, egy adott követelményrendszer előírásainak betartá-

sa mellett. A követelményrendszert az esetek többségében valamelyik nagy szabványalkotó testület specifikálja, mint pl. az IEEE (Institute of Electrical and Electronics Engineers), ETSI (European Telecommunications Standards Institute) vagy jelen esetben az AUTOSAR Consortium.

Alkalmazási területtől függetlenül a megfelelőségi tesztek szinte minden esetben feketedoboz tesztek, amelyek során a tesztelendő rendszernek vagy komponensnek a környezete felé mutatott viselkedése áll a vizsgálódás középpontjában. A teszt sikerességének verifikálásához egyáltalán nem, vagy csak minimális mértékben szükséges a tesztelt hardver- vagy szoftverkomponens belső realizációjának ismerete, a kérdés csupán az, hogy adott bemeneti gerjesztésre az elvárt kimenetet, viselkedést valósítja-e meg.

A tesztkörnyezet a teszt végrehajtásához szükséges meghajtókból (*driver*) és csonkokból (*stub*) áll. A meghajtó egy szoftver komponens vagy teszt eszköz, amely kiváltja azt a komponenst, amely a rendszer vezérlését végzi, vagyis parancsokat ad a SUT-nak annak érdekében, hogy megfigyelhessük az ezekre adott válaszokat. A csonk egy szoftver komponens speciális célú vagy részleges megvalósítása, amely helyettesíti a meghívott komponenst, vagyis közvetlenül megfigyelhetővé teszi a SUT-nak a környezetével végzett interakcióit a driver által generált gerjesztés hatására [2]. A csonk általában diagnosztikai funkción kívül nem lát el feladatot.

A tesztelés egyik kedvezőtlen sajátossága, hogy mindenre kiterjedő tesztelést – triviálisan egyszerű esetek kivételével – nem lehet végrehajtani, ugyanis egyrészt nem lehetséges minden bemenet és előfeltétel összes kombinációját előállítani a teszt során, másrészt a tesztelés arra alkalmas, hogy megmutassa a hibák létezését, nem pedig arra, hogy bizonyítsa azok hiányát. Ezért a tesztelés „jóságának” mérésére szokás különböző mérőszámokat, metrikákat használni (pl. kódfedettség, döntési feltétel fedettség). Ezek bemutatása előtt azonban célszerű áttekinteni a hozzájuk kapcsolódó fogalmakat:

- **Utasítás**: a programozási nyelvek egy entitása, ami tipikusan a futtatás legkisebb oszthatatlan egysége.
- **Feltétel**: olyan logikai kifejezés, amely igaz vagy hamis értékeket vehet fel (pl. A == B).
- **Döntés**: a progam olyan pontja, ahol a vezérlési folyamnak két vagy több alternatív útvonala van.
- **Elágazás**: a program egy logikai feltételtől függő útvonala (pl. if - else if - else).

A legelterjedtebb, és a tesztelés kvantitatív jellemzésére leginkább alkalmas metrikák az úgynevezett *kódfedettségi metrikák*:

- **Utasítás lefedettség**: a tesztkészlet által végrehajtott futtatható utasítások és az összes utasítás aránya százalékban.
- **Feltétel lefedettség**: annak mérőszáma, hogy a teszt végrehajtása során milyen arányban kerülnek meghívásra az egyes feltétel eredmények (milyen arányban értékelődnek ki). 100%-os feltétel lefedettség eléréséhez minden döntés minden feltételét igaz és hamis értékre is tesztelni kell.
- **Döntési lefedettség**: a tesztkészlet végrehajtása során a döntési eredmények meghívásának százalékos végrehajtási aránya. 100%-os döntési lefedettség 100%-os elágazási lefedettséget és 100%-os utasítás lefedettséget jelent.
- **Elágazás lefedettség**: a tesztkészlet által meghívott elágazások százalékos aránya. 100% elágazás lefedettség 100% döntési lefedettséget és 100% utasítás lefedettséget jelent.
- **Döntési feltétel lefedettség**: a tesztkészlet végrehajtása során az összes feltétel eredmény és döntési eredmény meghívásának százalékos aránya. 100% döntési feltétel lefedettség 100%-os feltétel lefedettséget és 100%-os döntési lefedettséget jelent.
- **Módosított döntési feltétel lefedettség (MC/DC)**: annak mérőszáma, hogy a teszt végrehajtása során milyen arányban kerülnek meghívásra a döntési eredményeket függetlenül befolyásoló egyes feltétel eredmények. 100%-os módosított döntési feltétel lefedettség 100% döntési feltétel lefedettséget jelent.

1.1. Szerepe az autóiparban

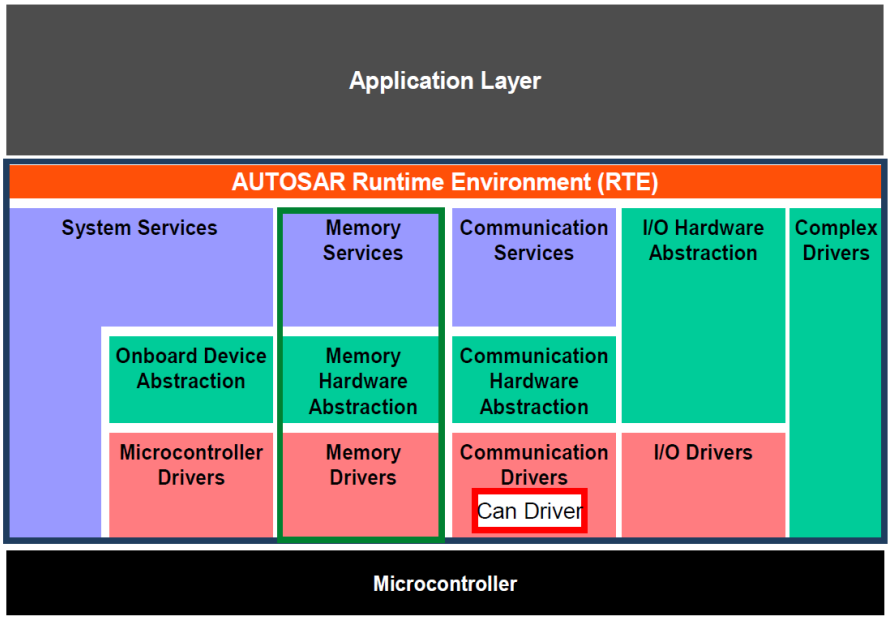
A teszteléshez hasonlóan a megfelelőségi tesztelés is igen általános fogalom, napjainkban a legkülönbözőbb területeken találkozhatunk vele mind mechanikai, mind elektronikai, mind szoftver komponensek kapcsán. Az autóiparban mindegyikre találhatunk példát szép számmal.

Számos megfelelőségi előírást specifikálnak a szabványok pl. az elektromos és elektronikus rendszerkomponensekre. Személygépjárművekben ezek tipikusan szenzorok (pl. abszolút- és relatív pozíciómeghatározók) és aktuátorok (pl. motorok), de ide sorolhatók a szórakoztató elektronikai eszközök (pl. autórádió, CD-lejátszó) és telekommunikációs eszközök (pl. beépített telefon, GPS) is.

A megfelelőségi követelményeket minden esetben teljesíteni kell, mielőtt a kérdéses eszköz integrálható lenne a rendszerbe, illetve mielőtt a rendszer akár tesztautóba kerülhetne. Különösen szigorú előírásoknak kell megfelelniük a biztonságkritikus rendszereknek (amilyen pl. a fék- vagy kormányrendszer) és általában a telekommunikációs rendszereknek: rádiófrekvenciás zavarvédettség és -kibocsátás, ESD (Electrostatic Discharge) elektrosztatikus kisülésekkel szembeni tűrés, burst feszültségcsúcsokkal (pl. DC motor feszültségtüskéi) szembeni tűrés stb.

Szoftverkomponensek esetén a vizsgálódás célja nagyon hasonló, módszerei azonban mások. A végső következtetés, amit egy önálló modul tesztelése után le kell tudnunk vonni az, hogy az adott szoftverkomponens a szabvány által előírt interfészeken keresztül az elvárt interakciókat bonyolítja-e le a szomszédos[1] modulokkal. Komplex rendszerek integrációs tesztelése esetén az előírások hasonlóak, bár máshogy kerülnek megfogalmazásra. Ebben az esetben integrációs-, rendszer- vagy átvételi (acceptance) tesztekről beszélhetünk (lásd 1. fejezet).

Egy adott modul implementációja és tesztelése szempontjából is fontos szerepet játszik a modulhoz specifikált *konfigurációs struktúra*. A szabvány a modulok megvalósítását egy *statikus* (kézzel írt, minden konfigurációban azonos) és egy *dinamikus* (konfigurációtól függő, tipikusan generált) részre bontással javasolja és megadja a konfigurációs adatok modelljét egy osztálydiagrammal.



1.1. ábra. *Implementációs megfelelőségi kategóriák [3].*

[1]Két AUTOSAR BSW szoftvermodult akkor nevezünk szomszédosnak vagy határosnak, ha van közös interfészük.

A megfelelőségi tesztek az 1.1. ábrán látható három implementációs kategóriába (ICC - Implementation Conformance Classes) sorolhatók [3]. Az ICC3 kategória (pl. a CAN Driver vagy I/O Drivers) az RTE-t és az összes BSW modult külön entitásként értelmezi, ez biztosítja a legfinomabb felbontást, ebbe a kategóriába tartoznak a modultesztek. Az ICC2 (pl. a Memory Services - Memory Hardware Abstraction - Memory Drivers együttes) a BSW-t stack-enként értelmezi, így pl. a memória stack vagy kommunikációs stack tesztjéről beszélhetünk, ez egy fokkal durvább felbontást tesz lehetővé, átmenetet képez a modultesztek és a teljes integrációs teszt között. Az ICC1 kínálja a legmagasabb integrációs szintet, az RTE-t és a BSW-t egy egységben kezeli. Az AUTOSAR által a BSW modulok számára definiált megfelelőségi tesztek (ICC3) alapvetően az alábbi két fő részből állnak:

Statikus tesztek az implementációs szabályok betartásának ellenőrzésére.

- **Forráskód-vizsgálat**: igazolni kell, hogy a SUT tartalmazza az összes specifikált forrás- és fejlécfájlt, a megfelelő típusdefiníciókat és exportálja az API-t. Ezek előfeltételei annak, hogy integrálni lehessen a tesztkörnyezetbe.
- **Konfiguráció ellenőrzés**: igazolni kell, hogy a SUT konfigurálható a specifikáció által előírt összes variánsra, az esetleges opcionális interfészekkel és működési módokkal, az aktuális konfigurációs elemtől függően. A BSW modulok tesztelését általában 4-8 különböző konfigurációs elemre kell elvégezni.
- **Interfész szignatúra ellenőrzés**: a megfelelőségi tesztelés során igazolni kell, hogy a SUT az általa biztosított (exportált) interfészeket a specifikációnak megfelelő formában valósítja meg, illetve, hogy az általa igényelt (importált) interfészeket a specifikációnak megfelelő formában éri el.

Dinamikus tesztek szolgálnak a szoftvermodul működésének ellenőrzésére.

- **Inicializáció-vizsgálat**: minden esetben igazolni kell, hogy a SUT inicializációja sikeresen lezajlott. Ez előfeltétele annak, hogy a viselkedési tesztek elindulhassanak.
- **Verzióinformáció ellenőrzés**: meg kell vizsgálni, hogy a SUT specifikus verziószámai megegyeznek-e a teszt esetben definiált verziószámokkal.
- **Viselkedési teszt esetek**: a valódi feketedoboz tesztek, feladatuk a modul által megvalósított funkcionalitás vizsgálata. Általuk igazolható, hogy a SUT adott „gerjesztés” esetén (ez általában valamely függvényének meghívását jelenti) a megfelelő módon reagál, pl. meghívja egy alacsonyabb modul egyik függvényét.

Látható, hogy még egy egyszerű modul megfelelőségének tanúsítása is igen összetett folyamat, amelynek során egyaránt szükség van a forráskód revíziójára és az összeállított teszt esetek lefuttatására, valamint az eredmények verifikálására. Hatalmas előnyt jelent, hogy az AUTOSAR konzorcium a 4.0 verzió 2-es revíziójáig bezárólag minden BSW modulhoz kibocsátott egy általuk specifikált és implementált megfelelőségi teszt csomagot (CTS - Conformance Test Suite) TTCN-3 nyelven.

A szabvány által specifikált megfelelőségi tesztek implementálását sok esetben a fejlesztőktől független társaságok (CTA - Conformance Test Agency) végzik, amelyek erre a feladatra szakosodtak. Erről bővebb információt az F.1. függelék tartalmaz. Azokban az esetekben, amikor a fejlesztés nem szigorúan egy jól specifikált szabvány mentén zajlik vagy elengedhetetlen a viselkedést az első release vagy upgrade kibocsátása előtt verifikálni, a fejlesztő cégek házon belül implementálhatnak, sőt specifikálhatnak megfelelőségi teszt-csomagokat [4].

1.2. Tesztleíró nyelvek

A téma alapjainak rövid áttekintése után célszerű megvizsgálni, hogy milyen lehetőségek állnak rendelkezésre a megfelelőségi tesztek elvégzésére, pl. egy AUTOSAR BSW modul esetén. Ez alapján pedig levonható a következtetés, hogy melyiket érdemes használni.

A legkézenfekvőbb megoldás természetesen az, hogy ugyanazon a nyelven írjunk teszteket, amin a szoftvert fejlesztettük – jelen esetben C nyelven. Ennek legfőbb előnye az egyszerűség, hiszen nem szükséges hozzá sem egy másik nyelv ismerete, sem másik fejlesztőeszköz, fordító stb. Erre tartalmaz néhány példát az 1.1. táblázat. A C-ben írt tesztek mellett szól még az, hogy segítségükkel nem csupán feketedoboz tesztek hajthatók végre, vagyis a modul belső viselkedését jobban ellenőrizhetjük (szürkedoboz teszt).

1.1. táblázat. *A C nyelvű modultesztelés legelterjedtebb eszközei.*

Standalone teszt	A tesztelés legegyszerűbb, egyben legkevésbé hatékony módja, amikor a kész szoftver köré valamilyen PC-n futtatható kódot írunk (a legtöbb esetben egy main függvényt), amelyben meghívjuk a tesztelni kívánt függvényeket, értéket adunk a megfelelő paramétereknek, kiíratjuk az eredményeket stb. Tipikusan .exe fájlt fordítunk belőle és így futtatjuk a teszteket.
Parasoft C/C++ Test	Integrált tesztkörnyezet, amelyben a függvényhívásokat stub függvények segítségével, az értékadásokat assertionök segítségével ellenőrizhetjük. Támogatja a stub generálást és paraméterezést, a hibakeresési (debug) módban történő tesztelést és az MC/DC tesztelést. Licenszdíjas.
CUnit	Működésének alapelve hasonló a Parasoft-hoz, azonban ez nem egy integrált környezet, hanem statikus könyvtárként linkelődik a felhasználói kódhoz. A tesztleírás stub függvényekre és assertion-ökre épül, többféle felhasználói interfész közül választhatunk. A különböző lefedettség-vizsgálatokat támogatja a Gcov révén (egy Eclipse plugin segítségével grafikus formában is). Nagy hátránya, hogy nem nyújt lehetőséget MC/DC számításra. Ingyenes.

Bizonyos szempontok miatt mégis érdemes lehet egy kimondottan erre a célra megalkotott tesztleíró nyelvet választani. Egyik legnagyobb előnyük, hogy sokkal formálisabb leírását biztosítják a teszt eseteknek, mint a C nyelv, egyúttal függetlenné is teszik őket az éppen használt fordítótól. Ezen felül további lehetőségeket is tartogatnak: adott esetben sokkal kényelmesebb az adatfolyam ellenőrzése egy ilyen tesztleíró nyelv használatával. A TTCN-3 pedig magában foglalja a platformszimulációs tesztek, valamint az időzítési tesztek lehetőségét is. Az 1.2. táblázat rövid áttekintést tartalmaz a leggyakrabban használt nyelvekről.

Hátrányaik között meg kell említeni, hogy elsajátításuk plusz időt és ráfordítást igényel, tehát ebből a szempontból eleinte bonyolultabb használni őket, mint C-ben írni teszt eseteket. Szintén a hátrányok közé sorolható, hogy használatukhoz általában külön fordító és futtatókörnyezet szükséges. Ezek a legtöbb esetben licenszdíjas eszközök, melyek ára igen széles skálán mozoghat, de általánosságban elmondható, hogy a jól használható eszközök jelentős beruházást igényelnek.

1.2. táblázat. *A legelterjedtebb tesztleíró nyelvek.*

TCDL (Test Case Description Language)	XML-alapú tesztleíró nyelv, amelyben a teljes tesztcsomagot különböző típusú előre definiált elemekből és a rájuk vonatkozó szabályokból építhetjük fel. Nehezen olvasható, de támogatja a lefedettség-vizsgálatot.
TestML (Software Testing Meta Language)	Metanyelv, amely a SUT implementációs nyelvétől független viselkedési tesztek leírására szolgál. Definiál egy adatpont-halmazt és egy programot, amely meghívja a tesztelt alkalmazást. Az adatpontokat transzformációs függvények konvertálják futási időben, majd ennek eredményét összehasonlítja a várt eredményekkel.
STIL (Standard Test Interface Language)	IEEE Std 1450.0-1999 szabvány, automatizálható „CAD-to-Test" eszköz, amely alacsonyszintű funckiók tesztelésére optimális: serial scan, partial/full boundary scan, időzítések vizsgálata, hullámforma vizsgálat.
TTCN-3 (Testing and Test Control Notation)	Az ETSI által fejlesztett szabvány tesztleíró nyelv, amely ideális feketedoboz tesztek végrehajtására. Tipikus alkalmazási területei a kommunikációs protokollok tesztelése, komponens-, integrációs- és rendszertesztelés, valamint a beágyazott és elosztott rendszerek tesztelése.

Az 1.2. táblázatban bemutatott tesztleíró nyelvek közül több nyomós érv is a TTCN-3 alkalmazása mellett szól:

- Kiválóan alkalmas a jelenleg elvégezni kívánt modulszintű feketedoboz tesztelésre.
- Szabványosított leíró nyelv, ezért egységes és jól dokumentált.
- Annak köszönhetően, hogy mára eléggé elterjedt, több jól használható fordító- és futtatóeszköz is rendelkezésre áll hozzá.
- Az AUTOSAR konzorcium ezen a nyelven bocsátotta ki a hivatalos megfelelőségi teszt csomagokat a BSW modulokhoz.

2. fejezet

A tesztelendő szoftvermodul

A megfelelőségi tesztelés folyamatának bemutatásához egy közepesen komplex AUTOSAR BSW modult választottam. A *FlexRay State Manager* a kommunikációs stack-en belül az egyik legalapvetőbb és legfontosabb feladatot látja el: a FlexRay sínre csatlakozó és a kommunikációban résztvevő ECU-k működési állapotainak beállítását. A modul komplexitása ideális ahhoz, hogy rajta keresztül a megfelelőségi tesztelést mélységeiben be lehessen mutatni, oly módon, hogy az illeszkedjen a dolgozat kereteibe.

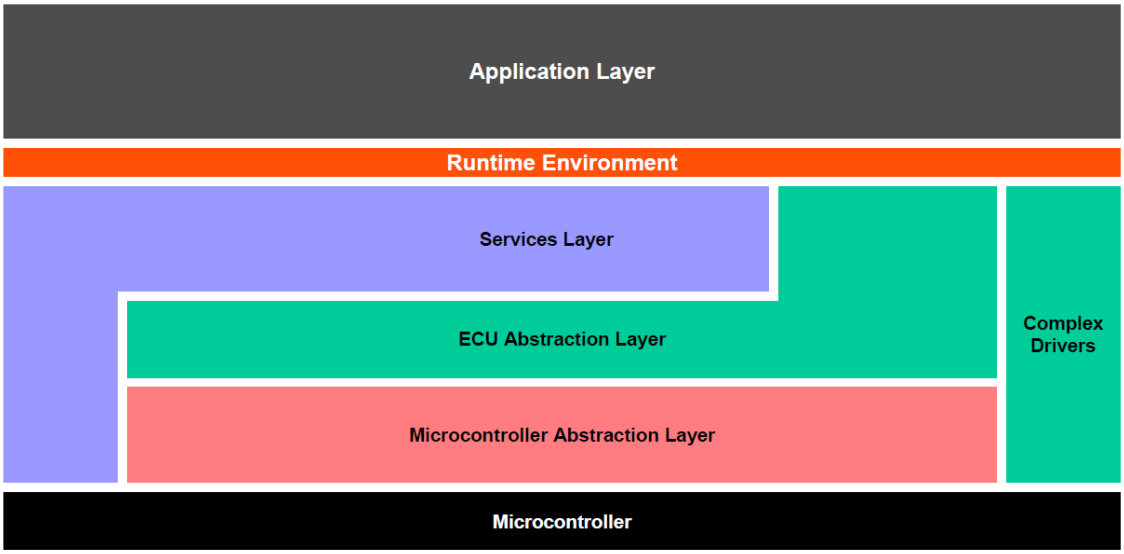
2.1. AUTOSAR szoftver architektúra

A szoftver architektúrára vonatkozó szabványok kialakítása során a szabványalkotók szem előtt tartották a felhasználási terület sajátosságait, nevezetesen azt, hogy az elkészült szoftverek szinte kivétel nélkül autóipari ECU-kon futnak. Ennek megfelelően a specifikált architektúra szorosan illeszkedik az ECU-k általános sajátosságaihoz, mint pl. a folyamatos interakció a környezettel és a hardverrel, a járműipari kommunkációs buszokra (CAN, FlexRay, LIN) való csatlakozás igénye vagy a valósidejű működéssel szemben támasztott követelmények.

Fontos megjegyezni, hogy az AUTOSAR terminológiában az ECU kifejezés egy mikrokontrollert, a hozzá tartozó perifériákat és szoftvert, valamint konfigurációt takarja. A 2.1. ábrán látható architektúra négy alapvető rétegre tagolható:

- Magas szintű alkalmazás réteg
- Runtime Environment (RTE)
- Basic Software (BSW)
- Mikrokontroller réteg

Ez nagyon hasonlít a beágyazott rendszerekben elterjedt általános hierarchikus felépítéshez. Az legjelentősebb különbség az *RTE-BSW tagolódás*, amely révén ez az architektúra a szokottnál nagyobb mértékű szabadságot biztosít a fejlesztők számára. A standard modulok funkcionalitása kiterjeszthető úgy, hogy illeszkedjenek a specifikált architektúrába és nem standard modulok is integrálhatók AUTOSAR-alapú rendszerekbe a Complex Drivers réteg által. További rétegek azonban nem adhatók hozzá.



2.1. ábra. *AUTOSAR szoftver rétegek áttekintése [5]*

A Runtime Environment réteg feladata a kommunikációs szolgáltatások biztosítása az alkalmazói szoftver (AUTOSAR szoftverkomponens és/vagy szenzor, illetve beavatkozó komponens) felé, valamint az, hogy az AUTOSAR szoftverkomponenseket függetlenné tegye az adott ECU-specifikus megvalósítástól. Az RTE réteg felett a szoftver architektúra már nem réteges, hanem komponens-alapú.

Ennek értelmében a szoftverkomponensek egymást és/vagy különböző szolgáltásokat ECU-n belüli és ECU-k közötti kommunikáció esetén is kizárólag az RTE-n keresztül érnek el. Az RTE kódját mindig specifikusan az adott ECU-ra generálják, a felsőbb rétegek felé nyújtott interfészei azonban teljesen függetlenek az ECU-tól.

2.1.1. Basic Software réteg

Az architektúra legösszetettebb rétege a Basic Software réteg, amelynek áttekintéséhez érdemes részletesen megvizsgálni annak tagolódását, az egyes alrétegek feladatát és egymáshoz való viszonyát [5].

- **Microcontroller Abstraction Layer**: Olyan meghajtókat tartalmaz, amelyek közvetlenül hozzáférnek a mikrokontrollerhez és annak belső perifériá-

hoz. Feladata, hogy az egyes mikrokontroller hardver sajátosságait elfedje a magasabb rétegek elől. Ennek megfelelően megvalósítása függ az alkalmazott hardvertől, felsőbb rétegek számára azonban standardizált, kontrollerfüggetlen interfészt biztosít.

- **ECU Abstraction Layer**: Interfészt képez a mikrokontroller absztrakciós réteg és a szolgáltatás réteg között, külső eszközökhöz tartozó driver-eket tartalmaz. Feladata, hogy egy olyan API-t biztosítson a szervíz réteg felé, amelyen keresztül elérhetők a perifériák és a belső eszközök attól függetlenül, hogy hol helyezkednek el és hogyan csatlakoznak a mikrokontrollerhez (hardver layout). Megvalósítása már nem függ a kontrollertől, függ azonban az ECU architektúrától, felsőbb rétegek felé ECU-független interfészt biztosít.

- **Services Layer**: Az alkalmazás szoftverhez legközelebb álló réteg. Feladata operációs rendszer szolgáltatások, hálózati kommunikáció és menedzsment, memória és diagnosztikai szolgáltatások biztosítása, valamint az ECU állapotainak és működési módjainak kezelése. Az alsóbb rétegeket néhány kivételtől eltekintve kizárólag az ECU absztrakciós rétegen keresztül éri el. Felsőbb rétegek számára teljes mértékben kontroller- és ECU-független interfészt biztosít.

- **Complex Drivers Layer**: Az egyetlen réteg, amely közvetlenül köti össze az RTE-t és a hardver réteget annak érdekében, hogy speciális funkcionalitást megvalósító driver-ek is integrálhatóak legyenek az architektúrába. Ezek olyan eszközök driver-ei lehetnek, amelyek nincsenek specifikálva az AUTOSAR szabványban, nagyon szigorú időzítési viszonyokkal rendelkeznek stb. Megvalósításuk és felfelé biztosított interfészük egyaránt függhet a mikrokontrollertől, az ECU-tól vagy éppen az alkalmazástól.

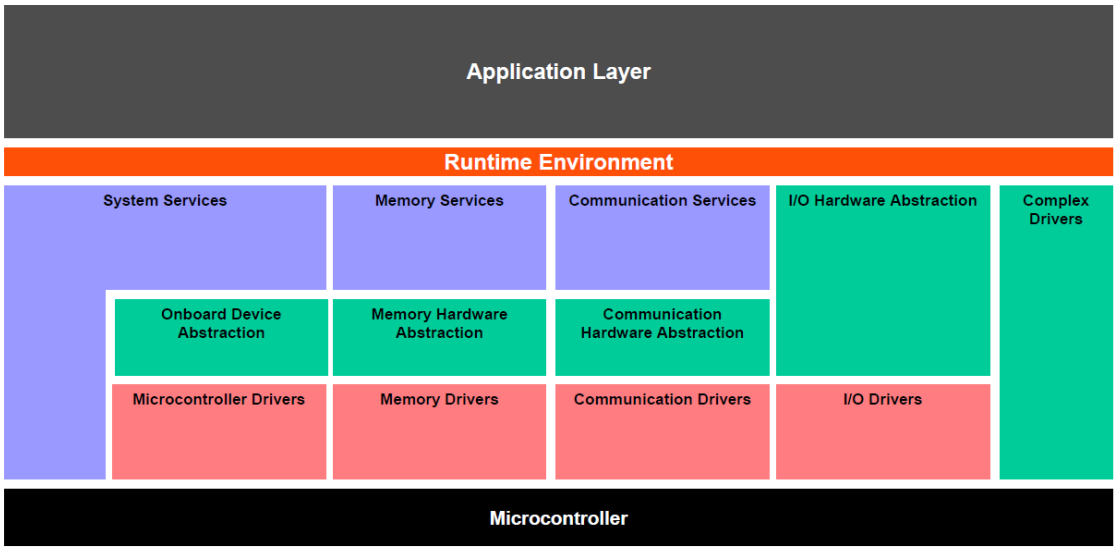
A fentiekben bemutatott *vízszintes* rétegződés mellett a Basic Software réteg *függőlegesen* is tagolódik, amint az a 2.2. ábrán látható. Az egyes rétegek átfogó szolgáltatáscsoportjai az alábbiak (az ábrán jobbról balra):

- **Input/Output**: Szabványos interfészt biztosít az alkalmazás réteg felé a szenzorokhoz, beavatkozókhoz és az ECU onboard perifériáihoz.

- **Communication**: A gépjármű kommunikációs hálózatához, ECU onboard kommunikációs rendszeréhez, valamint a belső szoftverhez biztosít szabványos hozzáférést.

- **Memory**: Külső és belső memóriához (RAM teszt) biztosít hozzáférést.

- **System**: Szabványosítható (operációs rendszer, időzítők, hibakezelés) és ECU-specifikus (állapot menedzsment, watchdog menedzsment) szolgáltatások és könyvtárak összessége.



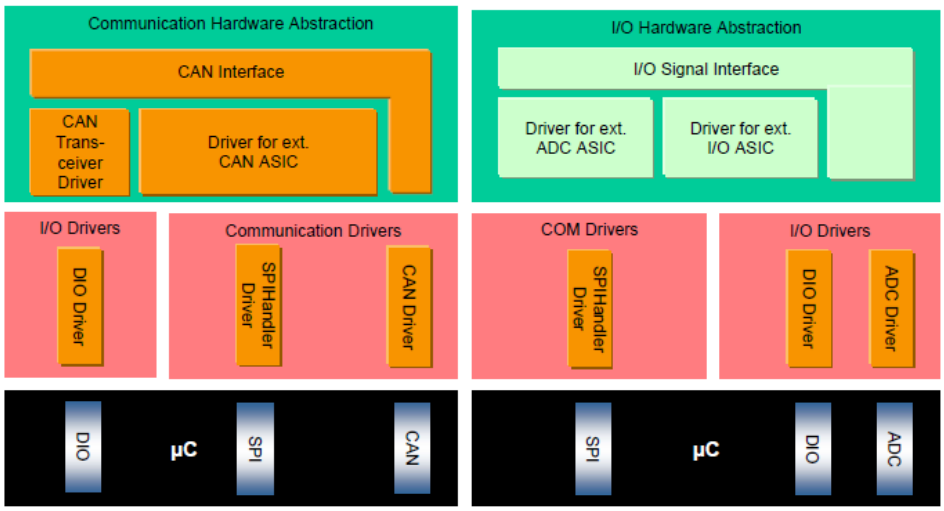
2.2. ábra. *AUTOSAR Basic Software réteg áttekintése [5]*

A FlexRay State Manager modulhoz közvetlenül kapcsolódó Communication Stack részletes bemutatása előtt, a teljesség érdekében érdemes röviden áttekinteni az egyes Basic Software rétegek tartalmát. A Microcontroller Abstraction réteg szerkezetében is megfigyelhető a függőleges tagolódás, ennek megfelelően az alábbi modulcsoportokat különböztethetjük meg:

- **I/O drivers**: Analóg és digitális input/output vonalak kezelése (ADC, PWM, GPIO stb).
- **Communication drivers**: az ECU onboard kommunikációs interfészeit (SPI, I2C) és a járművek buszaihoz (CAN, LIN, FlexRay) csatlakozó interfészeket kezelése.
- **Memory drivers**: belső, on-chip (Flash, EEPROM) és memóriába map-elt külső tárolók meghajtóit tartalmazza.
- **Microcontroller drivers**: a mikrokontroller belső perifériáihoz (Watchdog, General Purpose Timer) tartozó meghajtókat tartalmazza, valamint támogat olyan funkciókat, amelyek direkt módon kell, hogy elérjék a kontrollert (Core test).

Mind a mikrokontroller absztrakciós rétegre, mind az ECU absztrakciós rétegre igaz, hogy egyik sem értelmezhető igazán a másik nélkül, hiszen ketten együtt alkotják azt a réteget, aminek eredményeképp a szolgáltatás vagy szervíz réteg már

egy közel architektúra-független interfészt érhet el. Ennek megfelelően a 2.3. ábrán a kommunikációs és I/O ECU absztrakciós réteg moduljai a hozzájuk tartozó mikrokontroller absztrakciós rétegbeli modulokkal együtt láthatók.



2.3. ábra. *ECU és mikrokontroller absztrackiós réteg [5]*

Az I/O hardver absztrakciós réteg feladata, hogy a felsőbb rétegek számára egy olyan interfészen keresztül biztosítson hozzáférést az eszközmeghajtókhoz, amely független az adott periféria elhelyezkedésétől (on-chip, on-board) és az ECU hardver elrendezésétől. Nem független azonban a szenzorok/aktuátorok konkrét megvalósításától. Az egyes I/O eszközök egységesített, szabványos interfészen keresztül érhetők el, ami azonban függ a jeltípustól - ez az AUTOSAR szabvány szerint specifikált és implementált, így ennek megfelelően kell kezelni.

Hasonló feladatot lát el a kommunikációs ECU absztrakciós modul, amelyben külön almodulok látják el a különböző kommunikációs rendszerekhez való illesztést. Ha például adott az ECU-ban egy mikrokontroller, amelyhez tartozik 2 belső CAN csatorna és egy on-board ASIC 4 CAN vezérlővel, amely SPI porton keresztül csatlakozik a mikrokontrollerhez, akkor a kommunikációs absztrakciós rétegben CAN és SPI modult is meg kell valósítani. A kommunikációs driverek busz-specifikus interfészeken keresztül érhetők el. Ez a modul egységes, szabványos mechanizmust biztosít a kommunikációs busz csatornák elérésére, annak kialakításától és elhelyezkedésétől függetlenül.

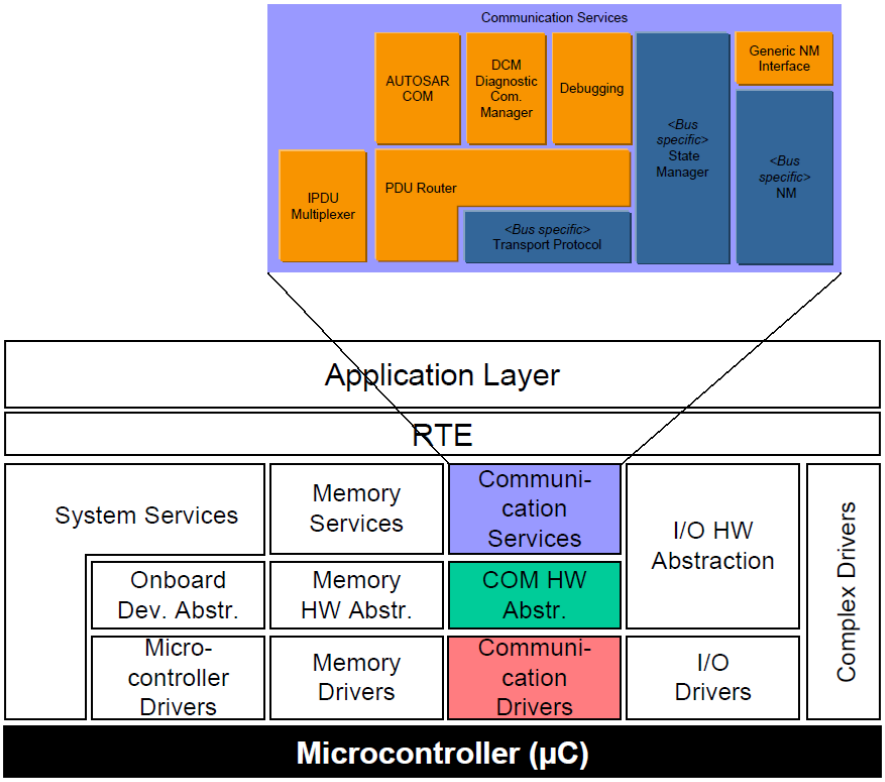
A memória hardver absztrakció elsődleges feladata, hogy egységesítse az on-chip és on-board memóriák elérését, valamint eltakarja a magasabb rétegek elől a különböző memória típusok (EEPROM, Flash) hozzáférésbeli sajátosságait. A memória driverek memória-specifikus absztrakciós/emulátor modulokon keresztül érhetők el.

Az on-board eszköz absztrakció olyan eszközök meghajtóit kezeli, amelyek nem szenzorként vagy aktuátorként látszanak a réteg számára (pl. belső vagy külső Watchdog).

A komplex meghajtók nem-standard funkcionalitást valósítanak meg a Basic Software stacken belül. Ilyen eset például, ha felmerül az igény arra, hogy speciális megszakítások révén közvetlenül elérjük a mikrokontrollert vagy komplex perifériáit használjuk, illetve ha speciális időzítési viszonyoknak kell eleget tennünk. Autóipari alkalmazásokban ez előfordulhat pl. befecskendezés-vezérlés, elektronikus szelepvezérlés vagy inkrementális pozíciómeghatározás esetén.

2.2. A Communication Stack felépítése

A kommunikációs driverek és a kommunikációs ECU absztrakciós réteg a kommunikációs szolgáltatásokkal (2.4. ábra) együtt alkotják a kommunikációs stacket. A kommunikációs stack tehát magában foglalja a kommunikációs modulok teljes láncát, amely összeköti a mikrokontroller réteget az RTE-vel.



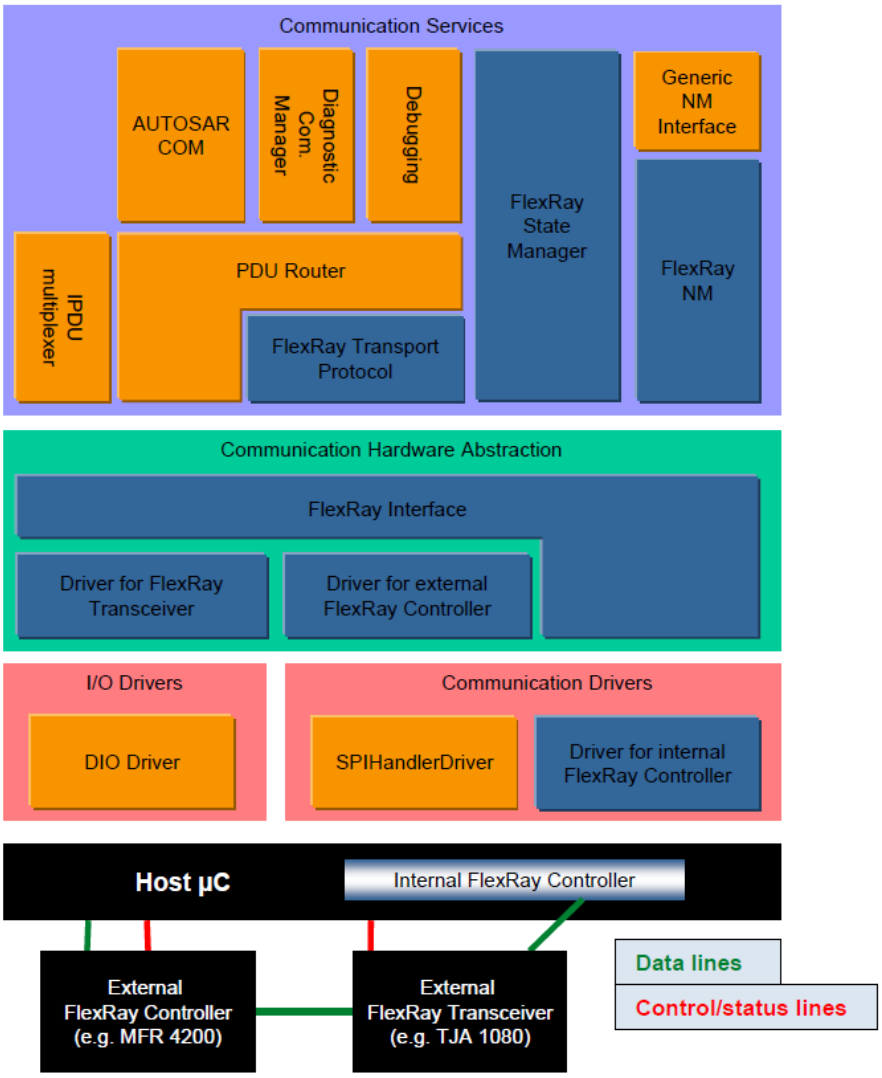
2.4. ábra. *Communication stack és a service modulok [5]*

A kommunikációs szolgáltatások rétege a jármű kommunikációs hálózatait a hardver absztrakción keresztül éri el, feladata pedig, hogy az alkalmazások felé *egységes interfészt* és magas szintű függvényeket biztosítson a szolgáltatásokhoz, ellássa a hálózatmenedzsment és diagnosztikai feladatokat és elfedje a protokoll és üzenetküldés sajátosságait az alkalmazás elől. Megvalósítása ennek megfelelően teljesen mikrokontroller- és ECU-független, függ azonban a kommunikációs busz típusától.

A fenti megállapítás igaz a FlexRay kommunikációs szolgáltatások moduljaira is, a 2.4. ábrán sárga színnel az általános célú modulok láthatók, amelyeknek egységesek a különböző kommunikációs buszok esetén, sötétkék színnel pedig a busz specifikus modulok:

- **State Manager**: feladata az adott kommunikációs buszra csatlakozó ECU-k állapotainak ütemezett és szinkronizált kezelése, azok bekapcsolása és felébresztése (lásd 2.3. fejezet).
- **Network Management**: a Communication Manager modul a NM révén szinkronizálja a kommunikációs preferenciákat/beállításokat a hálózaton keresztül, ez a modul felelős az adott kommunikációs busz absztrakt állapotainak beállításáért, a kommunikációban résztvevő node-ok felől érkező szavazatok alapján.
- **Transport Protocol**: feladata a felsőbb rétegekből érkező I-PDU-k (Internetwork Protocol Data Unit) szétbontása N-PDU-kra (Network Protocol Data Unit) és ezek továbbítása, valamint az alsóbb rétegből érkező N-PDU-k összeállítása I-PDU-kká és ezek továbbítása.

A megvalósítandó modul esetében a kommunikációs busz FlexRay busz, ennek megfelelően a hozzá tartozó kommunikációs stackből kell kiindulnunk a tervezés során. Amint az a 2.5. ábrán látható, a FlexRay State Manager az ECU absztrakciós réteg moduljai közül kizárólag a FlexRay Interface modult éri el, amely elfedi előle a legtöbb hardver-specifikus tulajdonságot.



2.5. ábra. *FlexRay communication stack vázlatos felépítése [6]*

2.3. FlexRay State Manager modul

A FlexRay hálózat bekapcsolása komplex folyamat, ami nagymértékben különbözik a többi hálózattól, így pl. a CAN-től. Amíg egy CAN hálózatban *bármely* üzenet felébresztheti a buszt, addig FlexRay hálózat esetén ehhez egy *speciális* wake-up minta szükséges. Annak érdekében, hogy a bekapcsolási folyamatot a lehető legmegbízhatóbbá tegyük, expilicit vezérlés és felügyelet szükséges, amelyet a BSW modul lát el. Mivel az AUTOSAR Communication Manager a buszt csak absztrakt formában látja az interfészén keresztül, ezért a FlexRay State Manager feladata, hogy ezt az absztrakt ábrázolást „lefordítsa” a POC (Procotol Operation Control) állapotaira és CHI (Controller Host Interface) parancsokra, amelyeket az állapotátmeneteket vezérlik [6].

A FlexRay State Manager fő feladata, hogy egy absztrakt interfészt biztosítson a Communication Manager modul számára, amelyen keresztül a kommunikáció be- és kikapcsolható a hálózaton (pontosabban egy clusteren[1]). A State Manager a specifikáció szerint alsóbb rétegekkel (FlexRay Communication Controller és FlexRay Transceiver) *kizárólag* a FlexRay Interface modulon keresztül kommunikálhat, direkt módon nem érheti el őket.

Az állapotgép belső működésének ismertetése a 2.3. fejezetben található. Ebből az állapotgépből a modul minden FlexRay cluster számára meg kell, hogy valósítson egyet, amelyben az állapotátmeneteket a cluster-specifikus main függvény FrSM_MainFunction_<Cluster Id> dolgozza fel. Kivételt képez ez alól néhány speciális állapotátmenet, amelyet az FrSM_RequestComMode függvényen belül kell végrehajtani a lekapcsolás determinisztikus voltának megőrzése érdekében.

Állapotok és állapotátmenetek

A State Manager állapotait részben a FlexRay Communication Controller POC állapotából, pontosabban a POC néhány eleméből származtatjuk. A modul felépítése, belső működése és a környezete felé megvalósított funkciói az alábbi elemek segítségével írhatók le:

- **Állapotok**: az állapotgép elemi állapotai (FRSM_READY, FRSM_WAKEUP stb.) ezekkel konkurens FRSM_ECU_ACTIVE / FRSM_ECU_PASSIVE állapot, valamint a BSW Manager felé jelentendő állapotok (az előző kettő alapján meghatározva).
- **Modul változók**: reqComMode, startupCounter, wakeupType, wakeupTransmitted, busTrafficDetected, wakeupCounter.
- **Konfigurációs paraméterek**: lehetővé teszik a szoftvermodul különböző variánsokra történő konfigurálhatóságát és skálázhatóságát.
- **Őrfeltételek**: futási időben értékelődnek ki minden cluster esetében (WUReason, AllChannelsAwake, wakeupFinished stb.)
- **Időzítők**: a modul az állapotátmenetek felügyeletére három szoftveres időzítőt használ működése során (t1, t2, t3).

[1]Clusternek nevezünk egy elosztott rendszert, amelynek csomópontjait (node-jait) legalább egy kommunikációs csatorna összeköti valamilyen topológiának megfelelően.

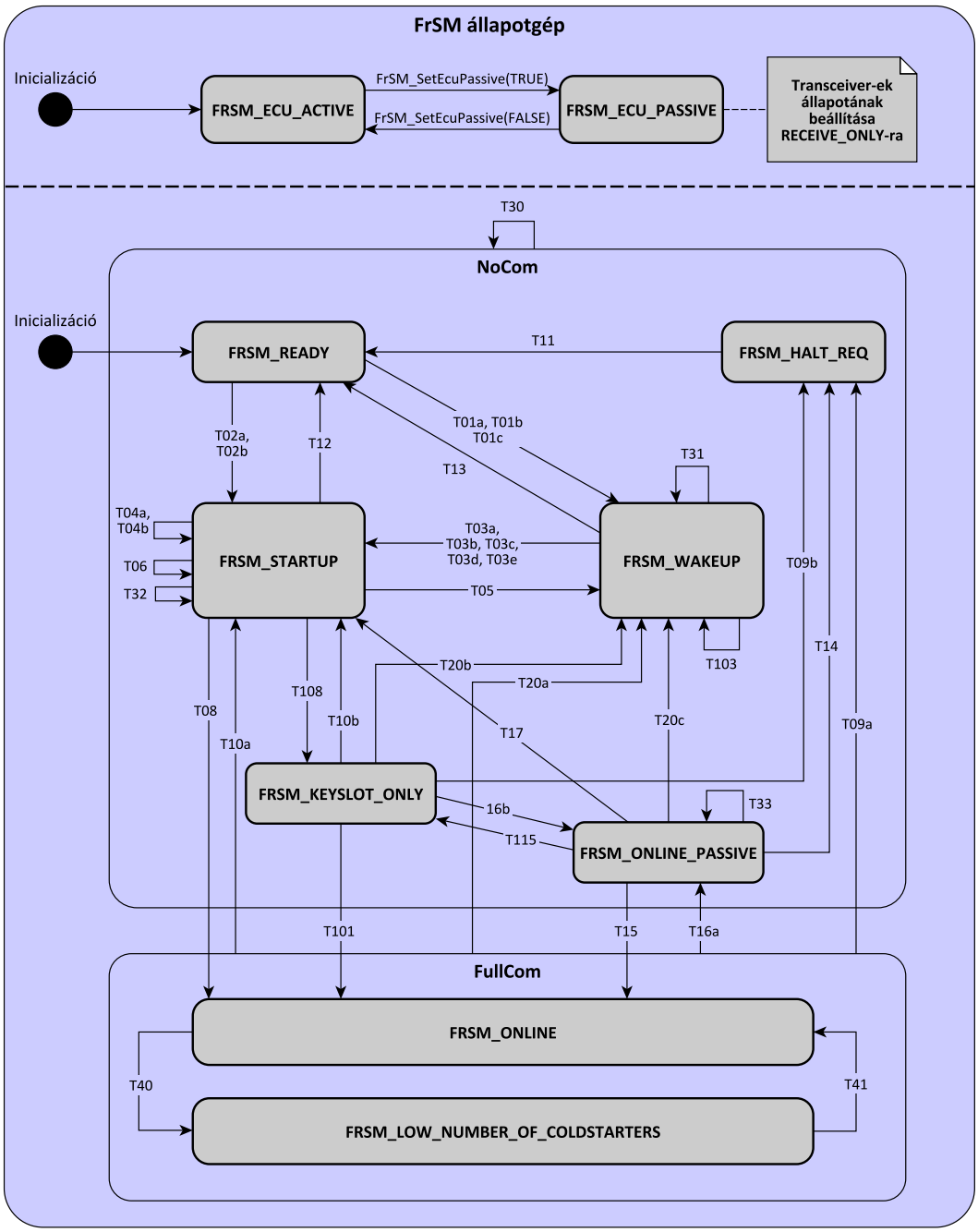
- **Aktivitások**: az egyes állapotátmenetek esetén végrehajtandó műveleteket a szabvány funkcionális elemcsoportokba partícionálja, amelyek általában néhány változó értékének beállítását és/vagy függvény meghívását foglalják magukban.

2.3.1. Állapottérkép

A State Manager belső működését, állapotait és állapotátmeneteit a 2.6. ábrán látható állapottérkép írja le. Fontos megjegyezni, hogy ezek a modul *belső állapotai*, amelyek nem képződnek le közvetlenül ECU állapotokra. Az ECU felébresztése, inicializálása vagy a kommunikáció elindítása ezen belső állapotok közötti meghatározott átmenetek eredménye.

A FlexRay State Managernek *minden* állapotváltást jelezni kell a BSW Manager felé a BswM_FrSM_CurrentState függvény segítségével. Egy állapotváltást mindig egy adott *őrfeltétel* teljesülése vált ki. Például ha a modul inicializálását követően FRSM_READY állapotban a Communication Manager FullCom állapotot kér, az ébresztés nem a buszról történt és az adott modul Wake-up ECU, valamint a FlexRay busz mindkét csatornájára csatlakozik, akkor bekövetkezik a T01(b) átmenet és a modul FRSM_WAKEUP állapotba kerül. Ebből az állapotból a megfelelő számú ébresztési kísérlet végrehajtása után, amennyiben még mindig FullCom a kért állapot, átkerülhet FRSM_STARTUP-ba vagy ha NoCom-ra vonatkozó kérés érkezett, visszakerül FRSM_READY-be.

Az időzítők is őrfeltételként szolgálnak egyes állapotváltások esetén. Bizonyos esetben azok időbeli lefutását felügyelik, pl. a t3 időzítő lejárta azt jelzi, hogy nem történt meg időben az átmenet FullCom állapotba és ez cluster startup hibajelzést indikál a Development Error Tracer és FlexRay Network Management modulok felé. FRSM_STARTUP állapotban a modul a startupCounter változó és a konfiguráció függvényében ismétli a bekapcsolási folyamatot. Ha ebbe az állapotba előzőleg az FRSM_WAKEUP-on keresztül jutott el (vagyis valamelyik T03 átmenettel), akkor a t2 időzítő lejárta után visszakerül ide, ellenkező esetben a t3 lejártáig marad FRSM_STARTUP-ban és folytatja a bekapcsolási kísérletek végrehajtását.



2.6. ábra. *A FlexRay State Manager működését leíró állapotgép*

2.3.2. API függvények

A FlexRay State Managerrel a többi modul az előre definiált API (Application Programming Interface) függvényeken keresztül léphet interakcióba. Az API magába foglalja a modulspecifikus típusokat (FrSM_ConfigType, FrSM_BswM_StateType), amelyeket a többi modulnak is látnia kell, valamint azokat a függvényeket, amelyeken keresztül az FrSM szolgáltatásai elérhetők.

FrSM_Init Elvégzi az összes FlexRay cluster inicializációját, FRSM_READY állapotba állítja az állapotgépeket (T00 állapotátmenet) és egy modulstatikus változóban eltárolja a bemenő paraméterként kapott konfigurációs struktúrára mutató pointert a többi függvény számára.

FrSM_RequestComMode Eltárolja a felsőbb modulok által igényelt kommunikációs módot a reqComMode változóban. Amennyiben a FullCom összetett állapotból a NoCom összetett állapotba kell lépni, azonnal végrehajtja az állapotátmenetet annak érdekében, hogy a NoCom igény biztosan megszüntesse az adott ECU részvételét a kommunikációban a FlexRay ciklus végéig.

FrSM_GetCurrentComMode A paraméterlistán visszaadja a cluster aktuális kommunikációs módját (eltárolja a paraméterként kapott memóriacímen).

FrSM_GetVersionInfo Segítségével lekérdezhetők az adott modul verzió információ paraméterei: moduleId, vendorId, gyártóspecifikus verziószámok.
Ez a függvény pre-compile időben konfigurálható az FrSMVersionInfoApi kapcsoló beállításával (ON / OFF).

FrSM_Allslots Segítségével hagyható el az ún. KeySlotOnlyMode, amely egy speciális működési mód (ekkor csak egy meghatározott kommunikációs szegmensben zajlik adás).

FrSM_SetEcuPassive Ha TRUE paraméterrel kerül meghívásra, az összes cluster transceiver-einek állapotát passzívba (csak vétel) állítja, egyébként aktívba. Ha egy cluster nem FRSM_READY állapotban tartózkodik, végrehajta rajta a TRCV_NORMAL aktivitást.

FrSM_MainFunction A modul generikus main függvénye, amely az állapottérképnek megfelelően feldolgozza az adott cluster által kapott eseményt, kezeli az időzítőket és végrehajtja az előírt interakciókat a külső modulokkal.

A specifikáció szerint a FlexRay State Manager modul által biztosított interfész csak FrSM_MainFunction_<ClusterId> formában megvalósított függvényeket tartalmaz, ezért a generikus main függvényt az alábbi módon map-eljük a szabványos formába:

```
void FrSM_MainFunction_<ClusterId>(void)
{
  FrSM_MainFunction(ClusterId);
}
```

2.3.3. Konfiguráció

Amint arról már a bevezetőben szó volt, az AUTOSAR szoftvermodulok optimalizálására a hozzájuk tartozó különböző *konfigurációs paraméterek* nyújtanak lehetőséget. A modul implementációjának generikus részeit ezek beállításával definiálhatjuk. Az egyes paraméterek a szoftverfejlesztés folyamatának különböző szakaszaiban kaphatnak értéket (ezek meghatározzák az egyes konfigurációs osztályokat és variánsokat):

- **Pre-compile time**: fordítási idő előtt kell, hogy értéket kapjanak, makrók formájában definiáljuk. Példa:

```
#define FRSM_DEV_ERROR_DETECT     STD_ON
#define FRSM_VERSION_INFO_API     STD_ON
```

- **Link time**: fordítási vagy linkelési időben kell, hogy eldőljön, tipikusan makrók vagy konstansok. Példa:

```
const FrSMClusterDemEventParameterRefsType FrSMClusterDemEventParameterRefs[];
```

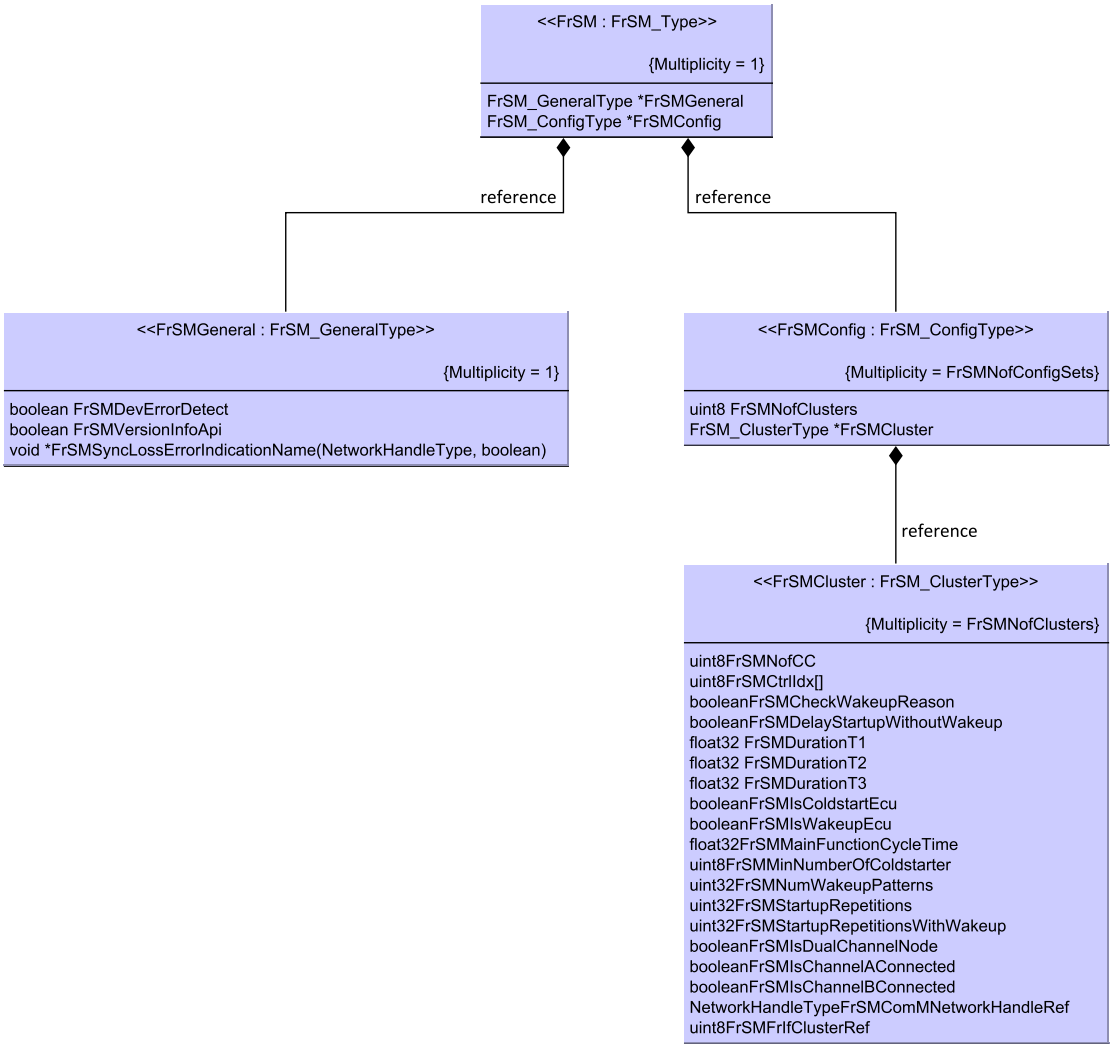
- **Post-build time**: futási időben is kaphatnak értéket, konstans vagy változó lehet. Példa:

```
const boolean FrSMCheckWakeupReason;
const float32 FrSMDurationT1;
```

Minden paraméter valamely konfigurációs containerbe tartozik (FrSMGlobal, FrSMConfig, FrSMCluster) ezek zárják egységbe a különböző osztályba tartozó konfigurációkat. A pre-compile time csoportba tartozó paramétereken kívül mindegyiket konstans formában realizáltam és a hozzá tartozó containernek megfelelő struktúrában tároltam.

A 2.7. ábrán látható UML osztálydiagram ábrázolja az egyes containerek hierarchiáját és egymáshoz való viszonyát. Egy cluster container multiplicitása azt jelenti, hogy az adott konfigurációs szettben (amit az FrSM_Init függvény kap paraméterként) hány darab FrSM cluster van jelen, a Config container multiplicitása pedig a rendelkezésre álló konfigurációs szettek számát adja meg. Utóbbinak a megfelelőségi tesztelés során lesz jelentősége.

A paraméterek közül külön meg kell említeni azokat, amelyek valamilyen fizikai mennyiséget reprezentálnak és dimenziójuk van. Ilyenek az időzítésekkel kapcsolatos konfigurációs értékek: az FrSMTimerDurationT1 stb. és az FrSMMainFunctionCycleTime, amelyek másodpercben értendők.



2.7. ábra. *A FlexRay State Manager konfigurációs struktúrája.*

3. fejezet

A TTCN-3 nyelv

A Testing and Test Control Notation nyelvet az ETSI (European Telecommunications Standards Institute) fejlesztette ki, eredetileg kimondottan kommunikációs protokoll alkalmazások tesztelésére. Ez a nyelv számos elemén megfigyelhető (komponensek, portok, üzenetek), habár az évek során renegeteg új elem került a nyelvbe, amelynek köszönhetően sok más alkalmazás tesztelésére is alkalmassá vált.

A fejezet célja a TTCN-3 nyelv szintaktikájának és szemantikájának ismertetése [7] alapján, valamint használatának bemutatása a FlexRay State Manager modulhoz tartozó CTS-ből származó példák segítségével. Az implementáció sarkallatos kérdése, hogy hogyan jelenik meg a tesztelt szoftver API-ja, valamint az általa használt interfészek a nyelvben és hogyan írja le az elvárt viselkedést, illetve hogyan modellezi az egyszerű és összetett adattípusokat.

3.1. Típusok és nyelvi elemek

A TTCN-3 nyelv számos jól használható beépített típust tartalmaz, amelyek közül több ismerős lehet más programozási nyelvekből (integer, float, enumerated). Ezeken az alaptípusokon kívül célszerű áttekinteni az FrSM modul teszt implementációjában használt egyszerű és összetett TTCN-specifikus adattípusokat, ugyanis az *adattípus-konverzió* egy igen fontos kérdése a tesztkörnyezet illesztésének a felhasználói kódhoz.

Egyszerű alap típusok		

integer	Egész típus, amely a pozitív és negatív egész számokat és a nullát foglalja magában.	Példa:
		`integer myIntVar1:=256;`
		`integer myIntVar2:=-273;`

float	Lebegőpontos típus, amely a számokat $mantissza \times bazis^{exponens}$ formában ábrázolja. Speciális értékei a ±infinity és a not_a_number.	Példa:
		`float myFloatVar1:=3.1415;`
		`float myFloatVar2:=-infinity;`

boolean	Kétértékű változó típus (igaz vagy hamis).	Példa:
		`boolean myBooleanVar1:=true;`
		`boolean myBooleanVar2:=false;`

verdicttype	A teszt eredmények reprezentálására szolgáló adattípus, lehetséges értékei a pass, fail, none és error.	Példa:
		`setverdict(pass);`
		`verdicttype myVerdict:=getverdict;`

Alap sztring típusok		

bitstring	0 és 1 karakterekből álló, tetszőleges hosszúságú karaktersorozat.	Példa:
		`bitstring myBitstring1:='01101'B;`
		`bitstring myBitstring2:='111001'B;`

octetstring	Hexadecimális karakterekből (0...9 és A...F) álló, nulla vagy páros hosszúságú karaktersorozat.	Példa:
		`octetstring myOctetstring1:='01'O;`
		`octetstring myOctetstring2:='88AF'O;`

charstring	Az ITU-T Recommendation T.50 szerint az International Reference Version-nek megfelelő karakterekből álló, tetszőleges hosszúságú karaktersorozat.	Példa:
		`charstring myCharstr1:="abcd";`
		`charstring myCharstr2:="efgh";`

Struktúrált típusok		

record	Rendezett struktúrált típus, a C-ből ismert struct TTCN-es megfelelője. Elemei alap típusúak vagy felhasználó által definiált típusúak lehetnek, rájuk a pont karakter segítségével hivatkozhatunk.	Példa:
		`type record MyRecordType {`
		`  integer field1,`
		`  float field2 optional,`
		`  charstring field3`
		`};`

record of	Olyan record, amelynek minden eleme ugyanolyan típusú. Elemeinek nincsenek egyedi azonosítói, adott típushoz rendelt, határozatlan elemszámú, rendezett tömbnek tekinthető.	Példa: `type record of boolean MyBooleanRecord;`
enumerated	A C-ből ismert enum TTCN-es megfelelője, olyan típusú változók modellezésére szolgál, amelyek csak meghatározott egész értékeket vehetnek fel. Minden enumerated értéknek egyedi azonosítóval kell rendelkeznie.	Példa: `type enumerated MyEnum { Monday, Tuesday, Wednesday };`
Speciális konfigurációs típusok		

address	A SUT egyes entitásainak egyedi címzésére szolgáló típus. A valódi adat reprezentáció a TTCN-3 specifikációtól függ, aminek segítségével az absztrakt teszt esetek a SUT valódi címzési mechanizmusától függetlenül specifikálhatók.	Példa:
		`/* Globalis address */`
		`type integer address;`
		`/* Port-specifikus */`
		`type port MyPortType message {`
		`  address MyAddress;`
		`  inout integer;`
		`}`

port	A portok teszik lehetővé a kommunikációt a teszt komponensek, illetve a teszt komponensek és a teszt rendszer interfész között. Két fő típusuk az üzenet-alapú és a procedúra-alapú port. Alapértelmezésként minden pont kétirányú, a valódi irány az in, inout vagy out kulcsszóval adható meg.	Példa:
		`/* Uzenet-alapu */`
		`type port MyMessagePort message {`
		`  in MsgType1;`
		`  out MsgType2;`
		`  inout integer`
		`}`
		`/* Procedura-alapu */`
		`type port MyProcedurePort procedure {`
		`  inout Proc1;`
		`  address integer`
		`}`

component	Definiálja, hogy mely portok és/-vagy attribútumok tartoznak egy adott komponenshez. A teszt komponenseknek háromféle szerepe lehet a teszt rendszerben: MTC (Main Test Component), PTC (Parallel Test Component) vagy TSI (Test System Interface). Az extend kulcsszó segítségével már definiált típusokból származtathatunk komponens típusokat, ezt nevezzük effektív típusdefiníciónak.	Példa: `type component MyCompType { var integer MyLocalInt; timer MyLocalTimer; port ProcPortType PC01; port MsgPortType PC02; }`

Egy TTCN-3 teszt csomag a nyelv definiált elemeiből építhető fel, amelyek a következők:

altstep A TTCN-3 nyelvben altstep-ek révén specifikálhatjuk és struktúrálhatjuk az egyes alt ágakban elvárt viselkedést.

```
/* altstep AltstepIdentifier(in Par1, inout Par2, ..) runs on ComponentType */
altstep a_FrIfAllSlots(inout boolean p_FrIfAllSlotsApiIsInvoked,
in Std_ReturnType_ p_StdReturn, inout integer p_NumTimesApiInvoked)
runs on PTCFrIf {
  []pt_FrIf.getcall(FrIf_AllSlots:{?}) -> param(v_CtrlIdx) {
    /* Az elvart viselkedes */
  }
[]pt_TestFrIf.getcall(TestFrIf_AllSlots:{?, ?, ?, ?}) ->
  param(v_VP, v_CtrlIdx, v_ApiExp, v_NumTimesApiInvoked) {
    /* Az elvart viselkedes */
  }
}
```

component Teszt komponens definíció (lásd 3.2. fejezet).

const Konstans definíció.

external A modulon kívül definiált nyelvi elem (external function, action, constant).

function Függvény definíció.

```
/* function MyFunction(in Par1, inout Par2, ..) runs on ComponentType
   return Type */
function f_ApiFrSMInit(in charstring p_VP) runs on MTCFrSM {
  /* Fuggvenytorzs */
}
```

group Azonos típusú nyelvi elemek csoportosítása adott szempont szerint.

import Definíciók importálása más modulokból.

module TTCN-3 modul definíció (lásd 3.4. fejezet).

port Kommunikációs port definíció (lásd 3.2. fejezet).

signature Szignatúra definíció, amely tartalmazza a SUT procedúra-alapú kommunikációs interfészeinek specifikációját.

template Adat vagy szignatúra template, amely valamilyen mintaillesztésre szolgál. Definiál teszt adatot és a hozzá tartozó minta értékeket vagy értékhatárokat, amelyek az illesztés kritériumaként szolgálnak.

testcase Teszt eset definíció, amely specifikálja a SUT-on végrehajtott műveleteket, vizsgálatokat, azok sorrendjét, valamint a hozzájuk kapcsolódó döntéseket.

timer Időzítő definíció. Az időzítők a modulvégrehajtás törzsében, teszt esetekben, függvényekben és altstep-ekben használhatók, tipikusan valamilyen időbeli őrfeltétel betartásának ellenőrzésére.

type Adattípus definíció, segítségével hozhatók létre a felhasználó által definiált egyszerű vagy összetett saját típusok.

var Érték tárolására szolgáló változó definíciója, amely a változó nevét a memóriában elfoglalt helyével társítja.

var template Template-ek tárolására szolgáló változó definíciója, amelynek tartalma a mintaillesztési mechanizmusnak megfelelően korlátozható.

Az ún. *paraméterezett objektumok* olyan TTCN-3 nyelvi elemek (modulok, struktúrált típusok, template-ek, függvények, altstep-ek vagy teszt esetek), amelyek definíciója nem teljes, azaz a felsőbb objektumok által használt néhány entitásuk (változók, portok, timerek stb.) nem értékelődik ki az objektum definíciójakor. Azokat a formális entitásokat, amelyek a ki nem értékelt entitásokat helyettesítik, *formális paramétereknek* nevezzük. Irányuk háromféle lehet:

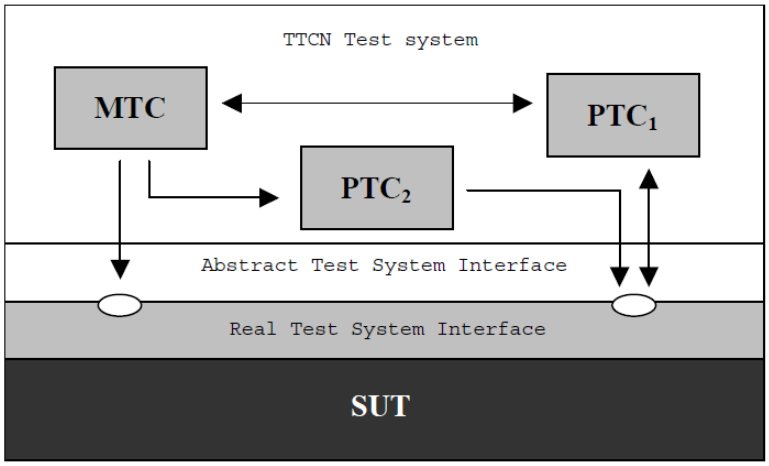
in Az argumentum kiértékelése már a hívás előtt megtörténik, értéke átadódik az objektumnak, az objektumon belüli változások azonban semmilyen hatással nincsenek az argumentumra a hívó oldalról nézve.

inout Az objektum direkt hozzáféréssel éri el az argumentumot, tehát minden, a formális paraméteren végrehajtott változtatás azonnal érvényre jut a valódi paraméteren.

out Az argumentum inicializálatlan a hívás előtt, amennyiben értéket kap az objektumon belül, úgy ezt az értéket a valódi paraméter örökli, egyébként változatlan marad.

3.2. Portok és komponensek

A TTCN-3 lehetővé teszi konkurens teszt konfigurációk dinamikus létrehozását. Egy konfiguráció egymással összeköttetésben álló teszt komponensből, az őket összekötő kommunikációs portokból és a *Test System Interface*-ből (TSI) áll, amely a tesztrendszer „határait" definiálja. Minden konfigurációban lennie kell egy MTC-nek (*Main Test Component*), amelyeket a rendszer automatikusan hoz létre a teszt végrehajtás kezdetekor, a végrehajtásnak pedig akkor van vége, amikor az MTC futása befejeződik. Az MTC-n kívül minden komponenst PTC-nek (*Parallel Test Component*) nevezünk. A PTC-k között nincs explicit hierarchia, végrehajtásuk és futásuk befejezése semmilyen hatással nincs sem a többi PTC-re, sem az MTC-re.



3.1. ábra. *Egy tipikus teszt konfiguráció portjai és komponensei [7].*

A teszt komponensek közötti, valamint a teszt komponensek és a TSI közötti kommunikáció a portokon keresztül válik lehetővé (lásd 3.1. ábra). A valódi komponens konfiguráció és az összeköttetések a teszt eset leírásában valósíthatók meg, create és connect műveletek segítségével. A komponensek portjait a map művelettel lehet összekötni a TSI-vel.

Minden port egy végtelen mélységű FIFO-val modellezhető, amely mindaddig tárolja a bejövő üzeneteket és procedúra hívásokat, amíg azok feldolgozásra nem kerülnek. A teszt esetek és a SUT közötti kommunikáció egy absztrakt teszt interfészen (TSI) keresztül valósítható meg, amely a valódi rendszerben megtalálható kommunikációs csatornákat hivatott helyettesíteni.

Ez tulajdonképpen nem más, mint egy komponens, amely statikus módon definiálja a teszt futása során a SUT-hoz csatlakozó portok számát és típusát. A TSI portjai a map és unmap műveletekkel köthetők be és ki egy komponensbe. Az alábbiakban a FlexRay State Manager teszt csomagjából származó példák láthatók port és komponens definíciókra:

```
module FrSMTestArchitecture {
  /* Procedura-alapu port definicioja */
  type port FrSMPort procedure {
    out FrSM_AllSlots,
    FrSM_GetCurrentComMode,
    FrSM_GetVersionInfo,
    FrSM_Init,
    FrSM_MainFunction_CFGIF,
    FrSM_RequestComMode,
    FrSM_SetEcuPassive
  }
  /* A FlexRay Interface modulhoz tartozo PTC definicioja */
  type component PTCFrIf {
  port TestFrIfPort pt_TestFrIf;
  port FrIfPort pt_FrIf;
  port TestMemoryAccessPort pt_TestFrIfMemoryAccess;

  var CTFrSMSetTransceiver_ vc_FrSMSetTransceiver[255];
  .
  .
  var CTFrSMAllSlots_ vc_FrSMAllSlots[255];
  }
  /* FlexRay State Manager MTC definicio */
  type component MTCFrSM {
  port FrSMPort pt_FrSM;
  port TestDemPort pt_TestDem;
  .
  .
  port TestMemoryAccessPort pt_TestMemoryAccess;

  var PTCFrIf vc_PTCFrIf;
  var PTCFrNm vc_PTCFrNm;
  var PTCComM vc_PTCComM;
  var PTCBswM vc_PTCBswM;
  }
} /* End module */
```

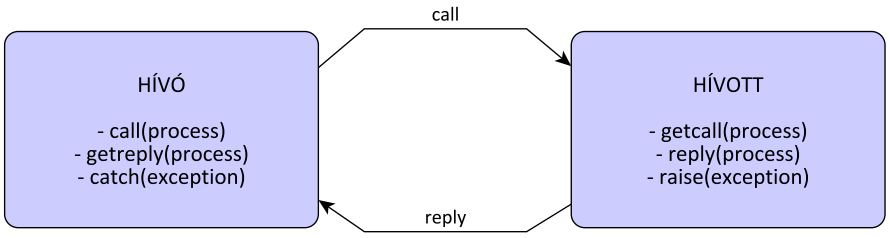
3.3. Procedúra-alapú kommunikáció

A TTCN-3 nyelv kétféle metódust specifikál a komponensek portokon keresztül történő interakciójára: a *procedúra-alapú* és az *üzenet-alapú* kommunikációt. Ez két alapjaiban különböző megközelítés, a közös bennük csupán annyi, hogy a kommunikációt kezdeményező komponens mindkét esetben kizárólag egy közös porton, előre definiált műveletek segítségével tud kapcsolatot teremteni célkomponenssel, illet-

ve visszajelzést fogadni tőle. A FlexRay State Manager modul teszt csomagjában kizárólag procedúra-alapú kommunikációra találunk példát, így csak ez kerül bemutatásra ebben a fejezetben.

A 3.2. ábrán látható kommunikációs mechanizmus alapja az, hogy a hívó a távoli entitás (hívott fél) valamely függvényét éri el az adott porton keresztül, amely lehet akár egy másik teszt komponens vagy a SUT része. Lehetőségünk van blokkoló és nem-blokkoló procedúra hívásra. Előbbi esetben a hívás a hívó és hívott oldalon is blokkolja a végrehajtást, utóbbi esetben csak a hívott oldalon:

- **Blokkoló hívás** esetén a hívó komponens a call művelet segítségével meghívja a hívott komponens valamely távoli függvényét. A hívott komponens a getcall művelet segítségével fogadja a hívást és visszajelzést küld a reply segítségével vagy kivételt jelez a raise művelet segítségével. A hívó fogadja a visszajelzést a getreply segítségével vagy kezeli a kivételt a catch meghívásával. Csak azután hajthat végre újabb műveletet, hogy ez megtörtént.
- **Nem-blokkoló hívás** esetén a hívó komponens a call művelet segítségével meghívja a hívott komponens valamely távoli függvényét, majd folytatja végrehajtást a soron következő művelettel, nem vár visszajelzésre vagy kivételre. A hívott komponens a getcall művelet segítségével fogadja a hívást és végrehajtja a megfelelő műveleteket. Sikertelen végrehajtás esetén kivételt jelez, amelyet a hívó egy alt blokkban a catch meghívásával kezelhet.



3.2. ábra. *TTCN-3 procedúra-alapú kommunikáció vázlat.*

A hívó és a hívott fél részéről elvégezhető műveletekre számos példát tartalmaz az FrSM modul hivatalos megfelelőségi teszt csomagja. Az ebben található példákon keresztül demonstrálható a procedúra-alapú kommunikáció mechanizmusa (a példák között nem szerepel a check és a raise művelet, mert ezeket nem használja a FlexRay State Manager CTS).

A hívó komponens által kezdeményezhető műveletek

Az 1. példában az MTC a call utasítással a pt_FrSM porton keresztül nem-blokkoló hívással (nowait kulcsszó) meghívja a SUT FrSM_SetEcuPassive függvényét a bemenő paraméterként kapott p_EcuState állapottal, majd ugyanezen a porton keresztül fogadja a procedúra válaszát a getreply parancs segítségével. A várt visszatérési érték esetén beállítja a teszt eredményt pass-ra, eltérő visszatérési érték esetén log-olja a hibát és a teszt eredményét fail-re állítja.

A 2. példában látható altstep hibakezelést valósít meg, ez meghatározza, hogy az MTC egyes alt ágaiban hogyan kell kezelni, ha valami kivétel adódik.

Példa 1:

```
/* A SUT FrSM_SetEcuPassive API fuggvenyenek tesztelesere szolgalo fuggveny */
function f_ApiFrSMSetEcuPassive(in charstring p_VP, in boolean p_EcuState,
    in Std_ReturnType_ p_ExpReturn) runs on MTCFrSM {
  /* Procedura hivas a pt_FrSM porton keresztul */
  pt_FrSM.call(FrSM_SetEcuPassive:{p_EcuState}, nowait);
  alt {
    /* Valasz fogadasa a vart visszateresi ertekkel */
    []pt_FrSM.getreply(FrSM_SetEcuPassive:{p_EcuState}
    value Std_ReturnType_: p_ExpReturn) {
      f_TestSetVerdict(pass);
    }
    /* Valasz fogadasa mas visszateresi ertekkel */
    []pt_FrSM.getreply(FrSM_SetEcuPassive:{p_EcuState}) {
      f_LogReturnValueFailure(p_VP, "FrSM_SetEcuPassive", p_ExpReturn);
      f_TestSetVerdict(fail);
    }
  } /* End alt */
} /* End f_ApiFrSMSetEcuPassive(...) */
```

Példa 2:

```
/* A felmerulo hibak kezelesenek definialasa */
altstep a_failureBehavior() runs on MTCFrSM {
  /* Egy fuggveny hivasa nem fejezodott be idoben */
  []any timer.timeout {
    f_LogTimeout("FrSM");
    f_TestSetVerdict(fail);
  }
  /* Nem vart valasz erkezik egy fuggvenytol */
  []any port.getreply {
    f_LogUnexpectedReply("FrSM");
    f_TestSetVerdict(fail);
  }
  /* Altalanos kivetelkezeles */
  []any port.catch {
    f_LogUnexpectedPort("FrSM");
    f_TestSetVerdict(fail);
  }
} /* End a_failureBehavior() */
```

A hívott komponens által kezdeményezhető műveletek

Procedúra hívás fogadása egy távoli entitástól a getcall parancs segítségével lehetséges. Az 1. példában a BswM PTC a pt_BswM porton keresztül fogadja a BswM_FrSM_CurrentState függvény hívását az adott paraméterekkel, beállítja a hívást jelző flag-et, majd visszajelzést küld a hívónak a reply művelet segítségével. Az altstep másik ágában a függvényhívás tesztelésére szolgáló pt_TestBswM porton fogad egy procedúrahívást, amely során ellenőrzi, hogy megtörtént-e a függvényhívás és ha igen, milyen paraméterekkel. Ezután visszaállítja a flag-et és visszajelzést küld a reply utasítással.

Példa 1:

```
/* A BswM_FrSM_CurrentState fuggveny hivasanak vizsgalata */
altstep a_BswMFrSMCurrentStateBehavior(inout NetworkHandleType_ p_Network,
    inout FrSM_BswM_StateType_ p_CurrentState,
    inout boolean p_BswMFrSMCurrentStateApiIsInvoked) runs on PTCBswM {
  /* Lokalis valtozok */
  var charstring v_VP;
  var NetworkHandleType_ v_Network;
  var FrSM_BswM_StateType_ v_ExpCurrentState;
  var boolean v_ApiExp;
  /* Procedura hivas fogadasa egy tavoli entitastol */
  []pt_BswM.getcall(BswM_FrSM_CurrentState:{?, ?}) -> param(p_Network,
  p_CurrentState) {
    /* Flag beallitasa es visszajelzes kuldese a hivo komponensnek */
    p_BswMFrSMCurrentStateApiIsInvoked := true;
    pt_BswM.reply(BswM_FrSM_CurrentState:{-, -});
    repeat;
  } /* End []pt_BswM.getcall(...) */
  /* Procedura hivas fogadasa az MTC-tol */
  []pt_TestBswM.getcall(TestBswM_FrSM_CurrentState:{?, ?, ?, ?}) ->
  param(v_VP, v_Network, v_ExpCurrentState, v_ApiExp) {
    /* API hivas vizsgalata */
    f_ValidateApiInvocation(v_VP, "BswM_FrSM_CurrentState", v_ApiExp,
    p_BswMFrSMCurrentStateApiIsInvoked);
    /* API hivas parametereinek vizsgalata */
    if(v_ApiExp) {
      f_ValidateIntegerParam(v_VP,"BswM_FrSM_CurrentState","CurrentState",
      enum2int(v_ExpCurrentState), enum2int(p_CurrentState));
      f_ValidateIntegerParam(v_VP,"BswM_FrSM_CurrentState","Network",
      v_Network, p_Network);
    }
    /* Flag beallitasa, visszajelzes kuldese */
    p_BswMFrSMCurrentStateApiIsInvoked := false;
    pt_TestBswM.reply(TestBswM_FrSM_CurrentState:{-, -, -, -});
    repeat;
  } /* End []pt_TestBswM.getcall(...) */
} /* End a_BswMFrSMCurrentStateBehavior(...) */
```

3.4. Modulok felépítése

A TTCN-3 nyelv legfelső egysége a *modul*. Egy modul nem bontható almodulokra, de importálhat definíciókat más modulokból, valamint *paraméterezhető*. A modulparaméterek lehetővé teszik a TTCN-3 modulok függetlenné tételét a SUT-tól, emellett a konfigurációhoz hasonlóan meghatározzák, hogy mely teszt esetek futtathatók, illetve értelmezhetők egy konkrét esetben (lásd 2.3.3. fejezet). Minden modul két fő részből áll: *definícióból* és *végrehajtásból*. Az alábbiakban a FlexRay State Manager teszt csomagjából származó példák láthatók TTCN-3 modulokra:

```
/* A konfiguracios parameter szettet tartalmazo modul */
module FrSMConfigParameters_1 {
  /* Importalas mas modulokbol */
  import from FrSMConfigTypes all;

  /* Modul parameterek */
  modulepar {
    Configuration_ CONF := {
      FrSM := {
        ...
        FrSMConfig := {
            FrSMCluster := {
              ...
            }, /* End FrSMCluster */
      FrSMCluster := {
              ...
            } /* End FrSMCluster */
        }, /* End FrSMConfig */
        FrSMGeneral := {
          ...
        } /* End general */
      } /* End FrSM */
    } /* End CONF */
  } /* End modulepar */
} /* End module */

/* Az FrSM teszt csomagjanak fo modulja */
module FrSMTestSuite {
  /* Importalas mas modulokbol */
  import from FrSMConfigFunctions all;
  .
  .
  import from FrSMStateTransition4TestCases all;

  /* Teszt esetek vegrehajtasa */
  control {
    execute(TC_FRSM_0078(), f_CfgCTFrSMTCMaxTimeout());
    .
    .
    execute(TC_FRSM_0100(), f_CfgCTFrSMTCMaxTimeout());
  } /* End control */
} /* End module */
```

4. fejezet

TTCN-alapú eszközök

Amint arról az 1.2. fejezetben szó volt, a TTCN-3 napjainkra széles körben használt tesztleíró nyelvvé vált, aminek köszönhetően számos fordító-, futtató- és fejlesztőkörnyezet jelent meg hozzá a piacon. Ingyenes eszközök után kutatva hamar nyilvánvalóvá válik, hogy teljesértékű, ingyenes TTCN-3 eszköz jelenleg nem érhető el. Annak ellenére sem, hogy a `http://www.ttcn-3.org/` oldal szerint több ilyen fejlesztés is zajlik vagy zajlott (köztük egy magyar cég neve is megtalálható). Ezek közül azonban sok nem is teszt eszköz (hanem pl. kodek- vagy dokumentációgenerátor), a többi pedig teljesértékűnek nem nevezhető (fordítót nem, csak futtatót tartalmazó), esetleg forráskód formájában, mintegy „továbbfejlesztésre" ingyenesen felkínált eszköz, de semmiképpen sem a célnak megfelelő tesztkörnyezet.

A licenszdíjas kereskedelmi eszközök között is több félbeszakadt projekt található, illetve sok olyan termék, amivel kapcsolatban – bár a gyártó honlapján nagyon tetszetős dolgok szerepelnek – komoly kétségek merülhetnek fel a funkcionalitást illetően. A kívánt alkalmazási területen használható, azaz modultesztelésre alkalmas eszközök a teljesség igénye nélkül az alábbiak [8]:

- **OpenTTCN Tester 2012**: az OpenTTCN (Finnország) nevű cég legsikeresebb terméke, amely számos nyelven támogatja a felhasználói kód illesztését: C, C++, C# és Java szoftverek tesztelésére is alkalmas és támogatja az ASN.1 formátumot[1].

- **Real Time Developer Studio**: a PragmaDev (Franciaország) által fejlesztett szoftver egy általános célú teszt eszköz, amelynek legújabb, 4.0-ás verziója támogatja a TTCN-3 nyelven írt teszt esetek futtatását is.

- **Telelogic Tester**: az IBM (USA) által felvásárolt Telelogic cég TTCN-3 fordítója, amely csak C nyelven írt kódhoz történő illesztést tesz lehetővé.

[1] Abstract Syntax Notation One - ISO szabvány adatábrázolási formátum, amely programozási nyelvtől, operációs rendszertől és hardvertől független reprezentációt tesz lehetővé.

- **TestCast**: az Elvior (Észtország) által fejlesztett TTCN-3 fejlesztőkörnyezet, amely támogatja a C, C++, C# és Java nyelven írt kódhoz történő illesztést.
- **TTCN-3 Express**: a Fraunhofer FIRST és Metarga GmbH (Németország) gondozásában készült TTCN-3 eszköz fordítót és teszt futtatót is tartalmaz, azonban kizárólag C# kód illeszthető hozzá.
- **TTCN-3 Toolbox**: a Devoteam Danet (Németország) által fejlesztett TTCN-3 fordító, amely támogatja az ASN.1 formátumot, valamint a C és C++ kódhoz való illesztést.
- **TTworkbench**: a Testing Technologies (Németország) terméke, amely számos extra funkciót kínál, azonban csak Java és C kódhoz illeszthető.

A fent ismertetett szoftverek közül a 4.1. fejezetben bemutatásra kerülő TTworkbench és a 4.2. fejezetben szereplő OpenTTCN került kipróbálásra. Előbbi elsősorban azért, mert a legtöbb szolgáltatást nyújtja, utóbbi pedig azért, mert a hivatalos AUTOSAR megfelelőségi teszt csomagokat ezzel a szoftverrel fordították és tesztelték.

4.1. Testing Technologies TTworkbench

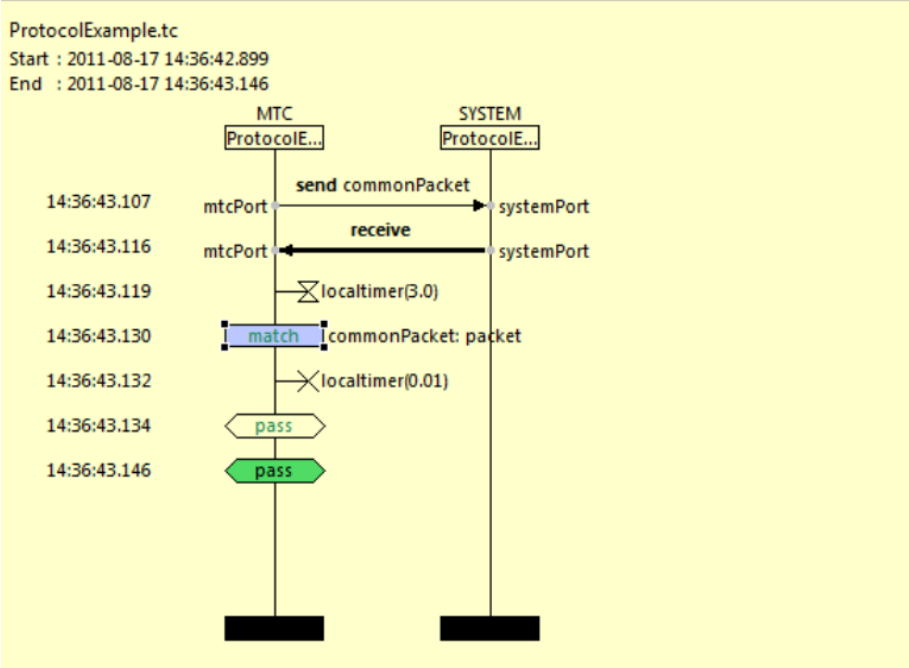
A TTworkbench egy Eclipse-alapú integrált fejlesztőkörnyezet és toolchain, amely három termékkategóriában (Express, Basic, Professional) érhető el. Az Express verzió kizárólag teszt futtatót (TTman) tartalmaz, a Basic ezen felül fordítót (TTthree), szerkesztőt és TTCN-3 szkriptek generálására alkalmas eszközt is. A Professional verzió tartalmaz még néhány szolgáltatást, ezek közül a legfontosabb a TTdebug, ami egy forráskód-szintű debugger. A támogatott operációs rendszerek a Microsoft Windows 7/Vista/XP, a SUSE Linux és a Fedora (32/64 bit).

A TTworkbench TTCN-3 nyelvi készlete a FlexRay State Manager modul teszt csomagjára nézve teljes, az abban használt összes kifejezést és nyelvi elemet támogatja. Néhány megkötést tesz ugyan, de ezek az AUTOSAR CTS-ek futtathatóságát nem befolyásolják: ilyenek pl. a template változókra történő hivatkozással vagy a karaktermintákban található metakarakterekkel kapcsolatos korlátozások. Szigorúbb, azonban az AUTOSAR-os teszt implementációt ugyancsak nem érintő korlátozás, hogy a TTworkbench az address, anytype, objid és universal charstring típusú értékeket nem tudja kezelni [9].

Az eszköz Java és C kódhoz illeszthető, utóbbihoz egy natív plugin segítségével. Az illesztő kód (glue code) az általános adattípus-konverziót és futási idejű illesztést (lásd 5.1 és 5.2. fejezet) kell, hogy megvalósítsa, akárcsak a többi TTCN-alapú

eszköz esetében. Implementációs támogatást a Testing Technologies egyrészt számos példakódon és template-en kersztül nyújt, másrészt a support team igen gyors és hatékony a felmerülő problémák megoldásában. Összességében elmondható, hogy bár a szoftver dokumentáltsága bizonyos esetekben hagy némi kívánnivalót maga után, a színvonalas online támogatás pótolja ezt a hiányosságot.

A TTworkbench a teszt eredményeket jól áttekinthető szekvenciadiagramon (lásd 4.1. ábra) jeleníti meg, ezzel elősegítve a gyors kiértékelést, a teszt futásának nyomonkövetését és az esetleges hibák megtalálását. A szekvenciadiagramon látható az összes komponens, amely a teszt futtatása során létrejött, közöttük nyilak jelölik a procedúra hívásokat, illetve üzeneteket és azok irányát. Minden értékvizsgálat esetén látható, hogy az eredmények egyeznek-e (match) vagy sem (mismatch) és hogy az adott teszt eset sikeres volt-e (pass) vagy sem.



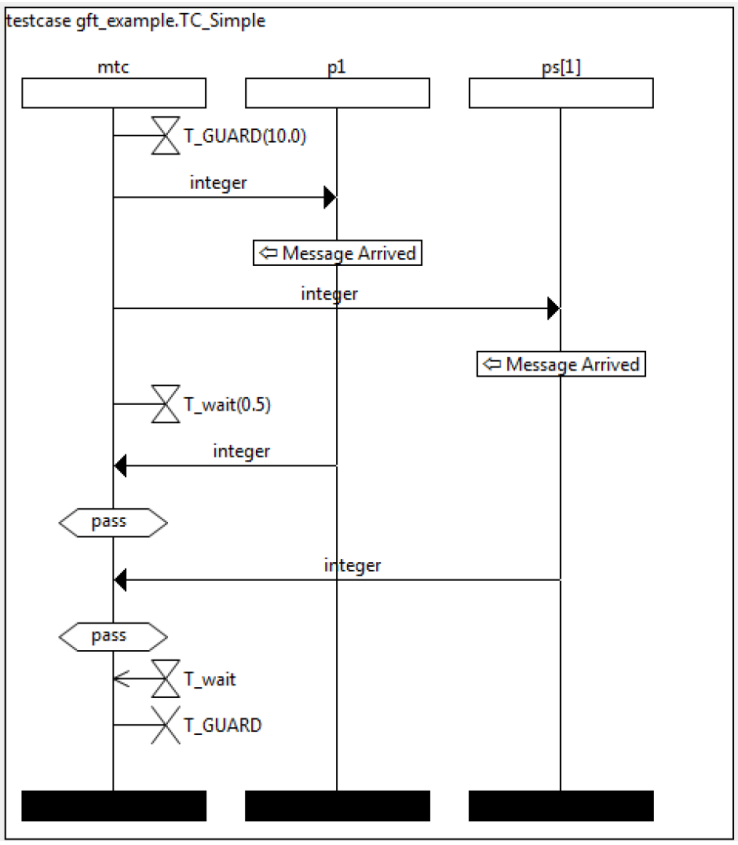
4.1. ábra. *A TTworkbench grafikus megjelenítője*

Esetünkben fontos szempont, hogy a tesztek futtatása lehetséges-e a GUI-n kívül valamilyen automatizált módszerrel (pl. parancssorból). A TTthree (fordító) és a TTman (futtató) összes funkciója elérhető konzol parancsok formájában, így a megfelelő makefile elkészítése után automatizáltan tesztelhető egy modul batch compiler-en.

4.2. OpenTTCN Tester 2012

Az OpenTTCN Ltd. nevű cég terméke szintén Eclipse-alapú fejlesztőkörnyezet és toolchain, amely a TTworkbench-hez hasonlóan három termékkategóriában (Standard, Professional, Enterprise) érhető el. A Standard Edition által kínált szolgáltatások elegendőek az AUTOSAR CTS-sel történő teszteléshez, a Professional ezen felül ASN.1-kompatibilitást és távoli adapterek használatának lehetőségét biztosítja. Az Enterprise TTCN-2 fordítót is tartalmaz, erre azonban nincs szükségünk. A támogatott operációs rendszerek a Microsoft Windows 7/Vista/XP és Debian Linux (32/64 bit).

Nyelvi készlete a FlexRay State Manager modul teszt csomagjára nézve teljes, ami nem meglepetés, hiszen mint arról már korábban szó volt, az AUTOSAR BSW megfelelőségi teszt csomagokat ezzel az eszközzel fordították. A hivatalos dokumentáció semmilyen egyéb megkötésről nem tesz említést [10].



4.2. ábra. *Az OpenTTCN grafikus megjelenítője*

Az OpenTTCN Tester 2012 ANSI C, C++, C# és Java szoftverhez illeszthető, utóbbihoz natív kód nélkül. A C nyelven írt modulok illesztéséhez lényegében ugyanazokat a funkciókat kell megvalósítani a glue code-ban, mint a TTworkbench esetén,

az OpenTTCN adapter és kodek könyvtáraiban található megvalósítások (HTTP, UDP, TCP) használhatók kiindulási alapként, illetve számos példa található a szoftverhez tartozó offline segédletben, valamint online súgóban. Ennek köszönhetően az OpenTTCN support szolgáltatására jóval kevesebb alkalommal volt szükség.

A megjelenítés OpenTTCN esetén is szekvenciadiagramon (lásd 4.2. ábra) történik, a leglátványosabb különbség a TTworkbench-hez képest, hogy itt az ábrák nem színesek és valamivel kevésbé informatívak. A szekvenciadiagramon itt is láthatóak a tesztkomponensek, a közöttük zajó interakciókat nyilak jelölik. Minden értékvizsgálat esetén látható, hogy az eredmények egyeznek-e (match) vagy sem (mismatch) és hogy az adott teszt eset sikeres volt-e (pass) vagy sem.

Az OpenTTCN szintén használható parancssorból, azonban kissé bonyolultabb módon, mint a TTworkbench. Míg utóbbi esetén gyakorlatilag egyetlen paranccsal fordítható, egy másikkal pedig futtatható a teszt csomag (a megfelelő opciók megadása mellett), addig előbbi esetén a tesztelés hét lépésből áll. Cserébe az OpenTTCN lehetőséget nyújt arra, hogy egy adott teszt csomagot szekvenciálisan több paraméter szettre (pl. több .par fájlra) is futtassunk.

4.3. A TTworkbench és az OpenTTCN összehasonlítása

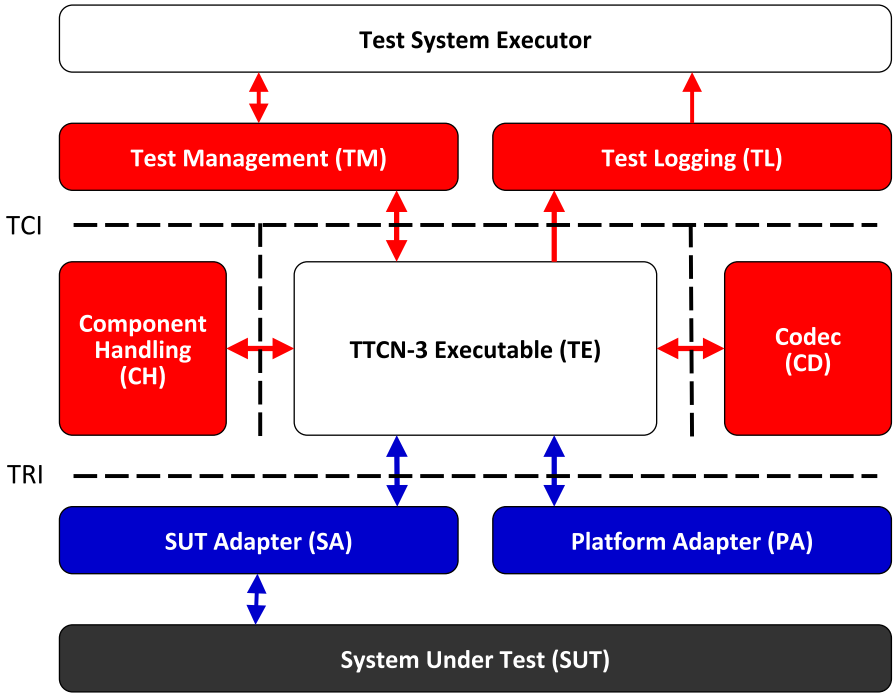
A két kipróbált TTCN-alapú fejlesztőeszköz értékelésében 1 jelenti az elégtelen, 5 pedig a kiváló eredményt egy adott kategóriában. Sok kategóriában közel azonos eredményt ért el a két szoftver, használhatóság szempontjából is egyenértékűek. A kritikus szempont, amely eldöntötte a versenyt az OpenTTCN Tester javára, a jóval kedvezőbb ár.

	TTworkbench Professional	**OpenTTCN Tester 2012**
Nyelvi korlátozások	Az AUTOSAR CTS szempontjából releváns korlátozás nincs. *Értékelés: 4*	Nincs ismert nyelvi korlátozás. *Értékelés: 5*
Illesztő kód	Implementálásához támogatás példakód és support formában. *Értékelés: 4*	Implementálásához támogatás beépített könyvtárakban. *Értékelés: 4*
Command line mód	Egyszerű, könnyen használható és jól automatizálható. *Értékelés: 5*	Kissé körülményes, több kódot igényel, nehezebben automatizálható. *Értékelés: 3*
Támogatott nyelvek	C és Java nyelvű kód illeszthető. *Értékelés: 3*	C, C++, C# és Java kód illeszthető. *Értékelés: 5*
Dokumentáltság	Néhány hiányosság tapasztalható (elsősorban C kód illesztés témakörben). *Értékelés: 3*	Alapvetően jónak mondható, részletes és áttekinthető. *Értékelés: 4*
Karbantartás és támogatás	Gyors, hatékony online support. *Értékelés: 5*	Ritkán volt rá szükség, késedelem előfordult. *Értékelés: 4*
Licensz	Volume licensz, lehetővé tesz egyidejűleg több hozzáférést. *Értékelés: 5*	Node-locked, floating és 12 hónapos előfizetéses licensz. *Értékelés: 4*
Ár	A kedvezőbb licenszfeltételek ellenére igen magas. *Értékelés: 2*	Több licensz vásárlása esetén is versenyképes. *Értékelés: 4*

5. fejezet

TTCN-3 tesztrendszer illesztése a SUT-hoz

A TTCN-3 nyelven implementált teszt csomagot és a futtatókörnyezetet minden esetben illeszteni kell a tesztelni kívánt modulhoz (SUT), amely valamilyen programozási nyelven – jelen esetben C-ben – áll rendelkezésre. Az illesztő kód vagy glue code egy része (pl. az adattípus-konverzió) egy adott programozási nyelvre nézve nagyrészt általános, míg másik része (pl. a SUT Adapter) minden teszt csomag és modul esetén egyedi. Szerencsére a glue code felépítése a második esetben is elég jól uniformizálható, így lehetőség nyílik annak generálására.



5.1. ábra. *TTCN-3 tesztrendszer általános felépítése.*

Az 5.1. ábrán látható egy TTCN-3 tesztrendszer általános felépítése, valamint komponenseinek kapcsolata. A rendszer egyes elemeinek feladata az alábbi [11]:

- **Test Management (TM)**: interfészt valósít meg a teszt végrehajtás és a tesztrendszer között, inicializáció után a TM-en belül kezdődik a végrehajtás, felelős a modulok megfelelő meghívásáért.
- **Test Logging (TL)**: rögzíti és a felhasználó számára megjeleníti a teszt végrehajtása során fellépő eseményeket (komponensek életciklusa, adat küldés és -fogadás stb.).
- **Component Handling (CH)**: a TE szétoszthatóvá különböző teszt eszközök között, ebben az esetben a CH teszi lehetővé a kommunikációt az elosztott entitások között.
- **TTCN-3 Executable (TE)**: a tesztrendszer azon része, amely a futtatható TTCN-3 teszt csomag végrehajtásáért felelős (ez tulajdonképpen a fordító kimenete).
- **Codec (CD)**: feladata a TTCN-3 típusú értékek megfelelő kódolása küldés (átalakítás TTCN-3 típusról a SUT Adapter-ben használt egységes típussá) előtt, valamint a fogadott értékek dekódolása (visszaalakítás TTCN-3 típussá) feldolgozás előtt.
- **SUT Adapter (SA)**: a TTCN-3 kommunikációs műveleteinek adaptálását valósítja meg, az absztakt teszt interfész realizációja (a valódi TSI).
- **Platform Adapter (PA)**: a futtatható TTCN-3 kódot (TE) egy adott platformhoz illeszti, létrehoz egy egységes időalapot a tesztrendszer számára és megvalósítja a külső és belső időzítőket.
- **TTCN-3 Control Interface (TCI)**: a TE és a teszt végrehajtás közötti interfész, amely négy funkcióhoz biztosít hozzáférést: adatkódolás és -dekódolás, tesztkomponensek kezelése, valamint log-olás.
- **TTCN-3 Runtime Interface (TRI)**: a TE és a SA, valamint PA közötti interakciókat definiáló interfész, amelynek elsődleges feladata, hogy a TE-ben lezajlott procedúra-hívások paramétereit univerzális formában átadja a SA-nak.

Egy TTCN-3 teszt csomag főmodulja az a modul, amelyben a control{...} blokk található. Egy ilyen modul feldolgozásának folyamata alapvetően négy fázisra tagolható:

1. **Inicializáció**: globális modulváltozók deklarálása és inicializálása, modul vezérlés entitásainak létrehozása és inicializálása.

2. **Frissítés**: időzítők értékének frissítése, változóik (pl. TIME-LEFT) beállítása, ha szükséges, a SUT-tól érkező üzenetek, procedúra hívások és válaszok elhelyezése a megfelelő port várakozási sorában.

3. **Kiválasztás**: egy nem-blokkolt állapotú module control entitás kiválasztása, és azonosítójának tárolása az Entity globális változóban.

4. **Végrehajtás**: az Entity változóban tárolt entitás soron következő execute(...) parancsának végrehajtása, majd a befejezési kritérium teljesülése esetén a végrehajtás leállítása, ellenkező esetben ugrás a 2. fázisra.

Az illesztő kód a 2. és 4. fázisban kap szerepet: előbbiben történik a SUT-tól érkező adatok, üzenetek és procedúra hívások fogadása, utóbbiban pedig a SUT felé irányuló műveletek lebonyolítása. Amint arról már a 4. fejezetben szó volt, az illesztő kódnak két feladatot kell ellátnia: adatkonverziót a TTCN-3 típusok és a C nyelv típusai között, valamint futásidejű illesztést, ami tipikusan a függvényhívások lebonyolítását jelenti.

5.1. Adattípusok közötti konverzió

Az adatkonverziót megvalósító kodekek és a TE (TTCN-3 Executable) között a TCI (TTCN-3 Control Interface) biztosítja az adatátvitelt, előre definiált függvények és adattípusok segítségével. A TCI-n belül megkülönböztethetjük az alábbi al-interfészeket és a hozzájuk tartozó szolgáltatásokat [12]:

- **TCI Test Management Interface (TCI-TM)**: magában foglalja az összes olyan műveletet, ami szükséges a tesztvégrehajtás menedzseléséhez (pl. tciRootModule, tciGetImportedModules, tciGetModuleParameters stb.).

- **TCI Component Handling Interface (TCI-CH)**: azokat a műveleteket tartalmazza, amelyek a tesztkomponensek életciklusát befolyásolják, illetve a közöttük történő kommunikációért felelősek (pl. tciStartTestComponent, tciConnect, tciTestComponentTerminated stb.).

- **TCI Coding/Decoding Interface (TCI-CD)**: megvalósítja az enkóderek és dekóderek eléréséhez szükséges műveleteket (pl. getInteger, getBoolean, decode, encode stb.).

- **TCI Test Logging Interface (TCI-TL)**: azokat a műveleteket tartalmazza, amelyek a tesztvégrehajtás során keletkező adatok továbbításához szükségesek (pl. tliCtrlTerminated, tliPrCall_c, tliPrGetCallDetected_c stb.).

A fenti felsorolásból látható, hogy az adattípusok közötti konverzió a TCI-CD interfészen keresztül lehetséges. Mind az OpenTTCN Tester, mind a TTworkbench tartalmaz néhány beépített kodeket a példaprojektekben használt adattípusok konvertálásához, amelyek kiindulási alapként szolgáltak az AUTOSAR modulok tesztelése során szükséges konverziók megvalósításához. Nem lehet azonban ebből a szempontból élesen különválasztani a kodeket és a SUT Adaptert, mert a BinaryString és a C típusok közötti konverziónak az SA-n belül kell történnie (lásd 5.2. fejezet).

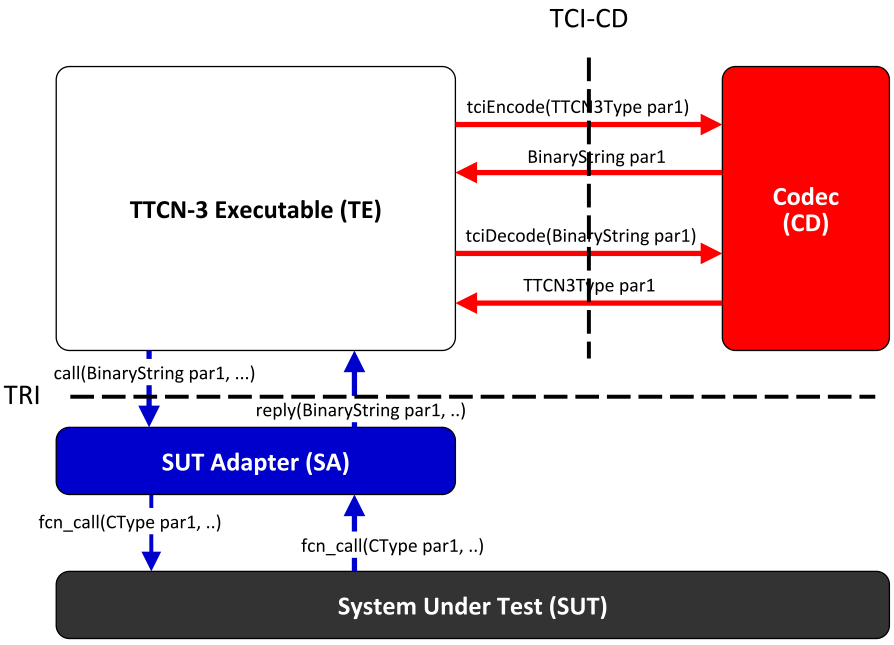
Mind a négy interfész kétirányú, az általuk megvalósított műveletek két csoportra oszthatók: a TE számára *szolgáltatott* és a TE-től *igényelt* műveletekre. Utóbbi két kifejezés tehát a felhasználó szemszögéből láttatja a műveleteket, vagyis hogy mit kell megvalósítanunk (szolgáltatnunk) az illesztő kódban és mi az, amit ehhez készen kapunk (igényelhetünk) a TE-től. Az igényelt funkciók implementációja adott, a szolgáltatott interfészt (a valódi kodeket) C nyelv esetén az alábbi függvények formájában kell biztosítanunk:

```
BinaryString tciEncode(Value v);
Value        tciDecode(BinaryString msg, Type decHypothesis);
```

A TCI és TRI interfészek függvényei által használt típusok definícióit az F.2. függelék tartalmazza. Az 5.2. ábra vázlatosan szemlélteti, hogyan zajlik a ki- és bemenő adatok konverziója egy procedúra hívás, illetve egy hívás fogadása során. A TE a SUT valamely függvényét az alábbi mechanizmust követve hívja meg:

- a TE-n belül a call parancs meghívódik az eredeti (TTCN-3 típusú) paraméterekkel
- a TCI-CD interfész tciEncode függvénye
  - meghívódik az eredeti paraméterek mindegyikére és konvertálja azokat
  - visszatér a BinaryString formátumba konvertált paraméterekkel
- a TRI-n keresztül BinaryString típusú paraméterek kerülnek átadásra a hívást megvalósító függvénynek (lásd 5.2. fejezet).

A SUT felől érkező függvényhívás paramétereit ezzel analóg módon konvertáljuk a tciDecode függvény segítségével. Annak érdekében, hogy a SUT-tól érkező válasz is BinaryString típusú paramétereket tartalmazzon, a SUT Adapter-ben is konvertálni kell a kimenő paramétereket (lásd 5.2. fejezet).



5.2. ábra. *A TCI-CD interfész vázlatos működése.*

5.1.1. Primitív típusok

A TTCN-3 adattípusok különböző attribútumainak (pl. integer esetén előjel, abszolútérték) meghatározására a fejlesztőkörnyezet beépített natív C könyvtárában jól használható függvények állnak rendelkezésre (pl. tciGetIntSign, tciGetIntAbs stb.), amelyek segítségével ezen egyszerű típusok konvertálása kényelmesen megoldható. A kódolást végző függvény törzse egy switch-case szerkezet. Először a tciGetType és tciGetTypeClass (igényelt) függvények segítségével megállapítjuk, hogy milyen típust szeretnénk konvertálni:

```
BinaryString tciEncode (Value v)
{
  BinaryString bs;
  char buff[255];
  int i;

  bs.bits = 0;
  bs.data = NULL;

  /* A kodolando tipus meghatarozasa */
  switch (tciGetTypeClass(tciGetType(v)))
  {
  /* case TYPE: ... */
  }
  bs.aux = NULL;
  return bs;
}
```

Maga a konverzió az egyes case ágakban történik. Az alábbi kódrészletben a tciEncode függvény enumerated és integer típusokat BinaryString-gé alakító része látható:

```
case TCI_ENUMERATED_TYPE:
{
  /* Ellenorzes: ervenytelen ertek? */
  if (tciNotPresent(v))
  {
    bs.bits = 8;
    bs.data = "0";
  }
  /* Ervenyes ertek */
  else
  {
    /* Bitek szamanak megadasa */
    bs.bits = 8*sizeof(int);
    /* Memoriafoglalas a mutatonak */
    bs.data = (char *)calloc(sizeof(uint8_t), (size_t)bs.bits/8);
    /* Enum ertek konvertalasa sztringge */
    bs.data = itoa(tciGetEnumIntValue(v), buff, 10);
  }
  break;
}
case TCI_INTEGER_TYPE:
{
  /* Ellenorzes: ervenytelen ertek? */
  if (tciNotPresent(v))
  {
    bs.bits = 8;
    bs.data = "0";
  }
  /* Ervenyes ertek */
  else
  {
    /* Bitek szamanak megadasa */
    bs.bits = tciGetIntNumberOfDigits(v)*8;
    /* Elojel-vizsgalat */
    switch (tciGetIntSign(v))
    {
    /* Pozitiv egesz szam */
    case 1:
      /* Memoriafoglalas a mutatonak */
      bs.data = (char *)calloc(sizeof(uint8_t), (size_t)bs.bits/8);
      /* Int ertek konvertalasa sztringge */
      bs.data = tciGetIntAbs(v);
      break;
    /* Negativ egesz szam */
    case 0:
      bs.data = (char *)calloc(sizeof(uint8_t), ((size_t)bs.bits/8)+1);
      /* Int ertek konvertalasa sztringge az atmeneti bufferben */
      buff = tciGetIntAbs(v);
      /* Negativ elojel beszurasa */
      bs.data = strcat("-", buff);
      break;
    }
  }
  break;
}
```

Dekódolás esetén arról, hogy a BinaryString milyen TTCN-3 típust reprezentál, a decodingHypothesis bemenő paraméter hordoz információt. Ennek megfelelően kell létrehoznunk egy új entitást a tciNewInstance függvény segítségével, majd a típus meghatározása után (tciGetTypeClass) a kódoláshoz hasonlóan egy switch-case szerkezetben végezhetjük el az adat dekódolását:

```
Value tciDecode (BinaryString msg, Type decodingHypothesis)
{
  Value v = NULL;
  char* str = NULL;

  /* Megfelelo tipusu entitas letrehozasa */
  v = tciNewInstance(decodingHypothesis);
  setDecodingRoot(v);

  /* A dekodolando tipus meghatarozasa */
  switch (tciGetTypeClass(decodingHypothesis))
  {
  /* case TYPE: ... */
  }
  return v;
}
```

A dekódoláshoz szintén rendelkezésre állnak beépített függvények, amelyekkel értéket adhatunk a létrehozott entitás egyes attribútumainak. Az alábbi kódrészlet végzi az enumerated és integer típusok dekódolását:

```
case TCI_ENUMERATED_TYPE:
{
  /* Az enum valtozo ertekenek beallitasa */
  tciSetEnumIntValue(v, atoi(msg.data));
  break;
}
case TCI_INTEGER_TYPE:
{
  /* A szamjegyek szamanak beallitasa */
  tciSetIntNumberOfDigits(v, msg.bits/8);
  /* Elojel-vizsgalat */
  switch (msg.data[0])
  {
  case '-':
    /* Negativ elojel beallitasa */
    tciSetIntSign(v, 0);
    break;
  default:
    /* Pozitiv elojel beallitasa */
    tciSetIntSign(v, 1);
    break;
  }
  /* Abszolutertek megadasa */
  tciSetIntAbs(v, msg.data);
  break;
}
```

5.1.2. Összetett típusok

Az összetett adattípusok közül az FrSM modul teszt csomagjában kizárólag record típusú adatok konvertálására van szükség. Ehhez is rendelkezésre állnak függvények a beépített könyvtárban. A record típusát a tciGetName függvény segítségével állapíthatjuk meg, mezőinek nevét pedig a tciGetRecFieldNames visszatérési értéke tartalmazza:

```
case TCI_RECORD_TYPE:
{
  /* Ellenorzes: ervenytelen ertek? */
  if (tciNotPresent(v))
  {
    bs.bits = 8;
    bs.data = "0";
  }
  /* Ervenyes ertek */
  else
  {
    /* Tipusnev tarolasa */
    char* name = tciGetName(tciGetType(v));
    /* A TTCN-es deklaracioban talalhato underscore karakter eltavolitasa */
    name[strlen(name)-1] = '\0';
    /* A record mezok nevenek tarolasa */
    char** fieldNames = tciGetRecFieldNames(v);

    /* A tipus kivalasztasa */
    if (!strcmp(name, "TYPE_NAME"))
    {
      /* Konverzio */
    }
    else if (!strcmp(name, "TYPE_NAME"))
    {
      /* Konverzio */
    }
    .
    .
    else
    {
      /* Hibajelzes */
    }
  }
}
```

A későbbi felhasználástól függően a record típusú adatok kódolására alapvetően kétféle lehetőség adódik (ezen a ponton is „összeér” a Codec és az SA). Azokban az esetekben, amikor a SUT meghívott függvénye közvetlenül írja és/vagy olvassa a paraméterként kapott adatot (pl. FrSM_GetVersionInfo), a kodekben létrehozhatunk egy megfelelő típusú struktúrára mutató pointert, amelynek tagonként értéket adunk, majd a BinaryString data mutatóját beállítjuk erre a struktúrára. Például a verzió információt tartalmazó record:

```
type record Std_VersionInfoType_
{
  integer vendorID,
  integer moduleID,
  integer sw_major_version,
  integer sw_minor_version,
  integer sw_patch_version
}
```

esetén létrehozunk egy ennek megfelelő C struktúrára mutató pointert, majd egy for ciklusban a record összes mezőjének értéket átadjuk a struktúra tagjainak, végül a BinaryString data mutatóját egyenlővé tesszük ezzel a pointerrel:

```
if (!strcmp(name, "Std_VersionInfoType"))
{
  Value v;
  /* Atmeneti mutato */
  Std_VersionInfoType* dat = malloc(sizeof(Std_VersionInfoType));

  for (i=0; fieldNames[i]!=NULL; i++)
  {
    /* A vendorID mezo ertekenek beallitasa */
    if (!strcmp(fieldNames[i], "vendorID"))
    {
      v = tciGetRecFieldValue(v, fieldNames[i]);
      dat->vendorID = atoi(tciGetIntAbs(v));
    }
    .
    .
    /* A sw_patch_version mezo ertekenek beallitasa */
    else if (!strcmp(fieldNames[i], "sw_patch_version"))
    {
      v = tciGetRecFieldValue(v, fieldNames[i]);
      dat->sw_patch_version = atoi(tciGetIntAbs(v));
    }
    else
    {
      /* Hibajelzes */
    }
  }
  /* A BinaryString data mutatojanak beallitasa  */
  bs.data = dat;
  /* Bitek szamanak megadasa  */
  bs.bits = sizeof(Std_VersionInfoType);
}
```

Az így kódolt record dekódolása hasonló módon történik, a record egyes mezőinek értékét a tciSetRecFieldValue függvény segítségével állíthatjuk be:

```
case TCI_RECORD_TYPE:
{
  /* Tipusnev tarolasa */
  char* name = tciGetName(tciGetType(v));
  /* A TTCN-es deklaracioban talalhato underscore karakter eltavolitasa */
  name[strlen(name)-1] = '\0';

  /* A tipus kivalasztasa */
  if (!strcmp(name, "Std_VersionInfoType"))
  {
```

```
    Value field;
    char buff[255];
    /* A record mezok nevenek tarolasa */
    char** fieldNames = tciGetRecFieldNames(v);
    /* Atmeneti mutato */
    Std_VersionInfoType* dat = msg.data;

    for (i=0; fieldNames[i]!=NULL; i++)
    {
      /* A vendorID mezo ertekenek beallitasa */
      if (!strcmp(fieldNames[i], "vendorID"))
      {
        /* Atalakitas sztringge az atmeneti bufferben */
        itoa(dat->vendorID, buff, 10);
        /* A field integer ertekenek beallitasa */
        tciSetIntAbs(field, buff);
        /* A megfelelo mezo ertekenek beallitasa */
        tciSetRecFieldValue(v, fieldNames[i], field);
      }
      .
      .
      /* A sw_patch_version mezo ertekenek beallitasa */
      else if (!strcmp(fieldNames[i], "sw_patch_version"))
      {
        /* Atalakitas sztringge az atmeneti bufferben */
        itoa(dat->sw_patch_version, buff, 10);
        /* A field integer ertekenek beallitasa */
        tciSetIntAbs(field, buff);
        /* A megfelelo mezo ertekenek beallitasa */
        tciSetRecFieldValue(v, fieldNames[i], field);
      }
      else
      {
        /* Hibajelzes */
      }
    }
  }
}
```

Egyéb esetekben (pl. amikor egy függvénycsonk szolgáltat adatot a SUT-nak) kényelmesebb megoldás, ha a record elemeit egy általunk definiált formájú sztringben tároljuk. Például a POC (Protocol Operation Control) információt tartalmazó record:

```
type record Fr_POCStatusType_
{
  boolean ColdstartNoise,
  boolean CHIHaltRequest,
  boolean Freeze,
  Fr_SlotModeType_ SlotMode,
  Fr_WakeupStatusType_ WakeupStatus,
  Fr_ErrorModeType_ ErrorMode,
  Fr_StartupStateType_ StartupState,
  boolean CHIReadyRequest,
  Fr_POCStateType_ State
}
```

esetén annak elemeit rendezetten egy adott formátumú sztringben tárolhatjuk, majd ezt adjuk át a BinaryString data mutatójának:

```
else if (!strcmp(name, "Fr_POCStatusType"))
{
  sprintf( buff, "{%d, %hu, %hu, %hu, %hu, %d, %d, %d, %d}",
    tciGetEnumIntValue(tciGetRecFieldValue(v, "State")),
    tciGetBooleanValue(tciGetRecFieldValue(v, "Freeze")),
    .
    .
    tciGetEnumIntValue(tciGetRecFieldValue(v, "StartupState"))
  );

  bs.data = strdup(buff);
  bs.bits = 8*(strlen(buff));
}
```

A fenti módszer egyik legnagyobb előnye, hogy így nincs szükség for ciklusra és feltételvizsgálatokra, másik előnye, hogy a kódolás/dekódolás kényelmesebb. Hátránya, hogy az ebben a formában reprezentált adatok feldolgozása az SA-ban valamivel bonyolultabb. A sztring formátumban tárolt record dekódolása az alábbi módon történhet:

```
else if (!strcmp(name, "Fr_POCStatusType"))
{
  Fr_POCStateType tmpState;
  boolean tmpFreeze;
  boolean tmpCHIHaltRequest;
  boolean tmpCHIReadyRequest;
  boolean tmpColdstartNoise;
  Fr_SlotModeType tmpSlotMode;
  Fr_ErrorModeType tmpErrorMode;
  Fr_WakeupStatusType tmpWakeupStatus;
  Fr_StartupStateType tmpStartupState;

  if (msg.bits/8 == sscanf( msg.data, "{%d, %hu, %hu, %hu, %hu, %d, %d, %d, %d}",
      tmpState,tmpFreeze,tmpCHIHaltRequest,tmpCHIReadyRequest,
      tmpColdstartNoise,tmpSlotMode,tmpErrorMode,tmpWakeupStatus,tmpStartupState))
  {
    tciSetRecFieldValue(v, "State", tmpState);
    tciSetRecFieldValue(v, "Freeze", tmpFreeze);
    .
    .
    tciSetRecFieldValue(v, "StartupState", tmpStartupState);
  }
  else
  {
    /* Hibajelzes */
  }
}
```

5.2. Futtatókörnyezet illesztése

A TE és a SUT Adapter, valamint Platform Adapter közötti interakciók a TRI szolgáltatásain keresztül lehetségesek (lásd 5.1. ábra). Ez az interfész teszi lehetővé a TE számára, hogy adatokat küldjön a SUT-nak, kezelje az időzítőket, illetve adatokat fogadjon a SUT-tól. A TRI két al-interfészre tagolható [11]:

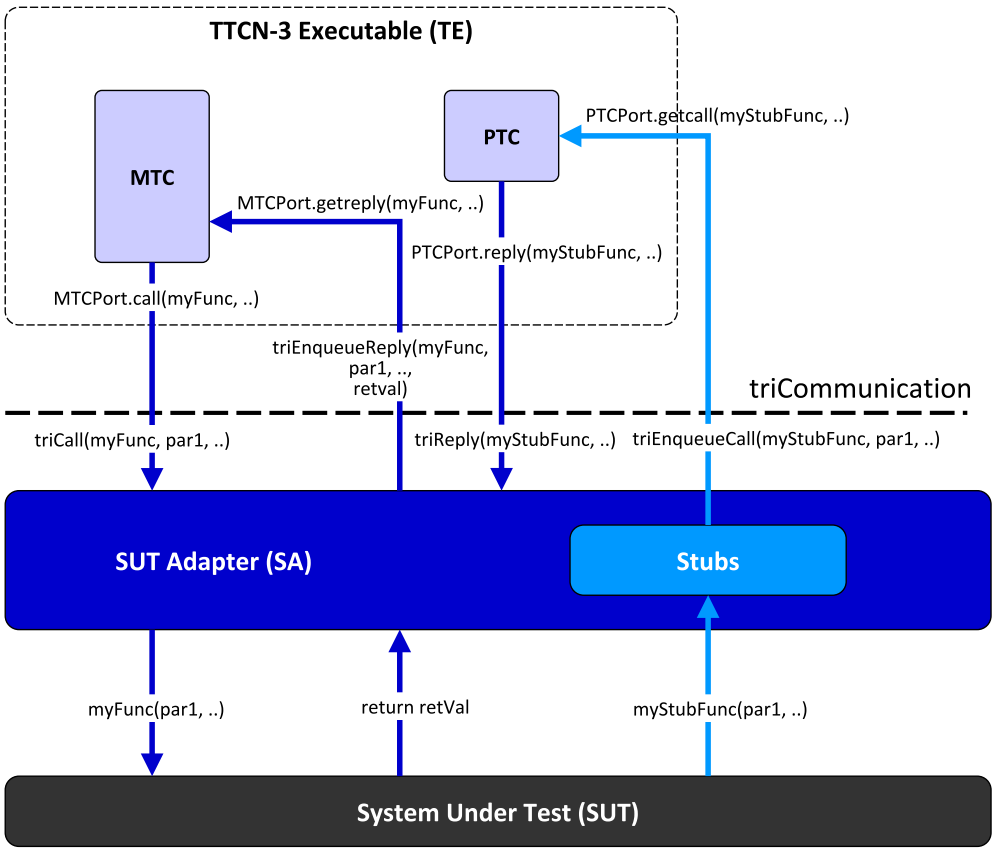
- **triCommunication**: biztosítja azokat a műveleteket, amelyek szükségesek a TTCN-3 teszt esetek és a SUT közötti interakciókhoz, beleértve a TSI inicializálását, a TE-SUT kapcsolat létrehozását a portokon keresztül, az üzenet- és procedúra-alapú kommunikáció lebonyolítását, valamint az SA újraindítását.
- **triPlatform**: lehetővé teszi a TE adaptálását egy adott platformhoz, lehetőséget nyújt a belső és külső időzítők kezelésére, azok állapotának lekérdezésére, external függvényhívásokra, valamint a PA újraindítására.

Az FrSM modul teszt csomagjában a Platform Adapter használatára nincs szükség, ezért csak a triCommunication interfész által biztosított műveleteket kell használni. Ezek közül az illesztés szempontjából releváns műveletek az alábbiak:

- **triMap (TE→SA)**: a map művelet hatására hívódik meg, feladata, hogy a compPortId paraméterként kapott komponens portjait összekösse a TSI megfelelő portjaival.
- **triCall (TE→SA)**: a call művelet hatására hívódik meg, a tsiPortId porton keresztül meghívja a componentId által kijelölt komponens signatureId által előírt függvényét a parameterList-nek megfelelő paraméterekkel.
- **triReply (TE→SA)**: a reply művelet hatására hívódik meg, a tsiPortId porton keresztül meghívja a componentId által kijelölt komponens signatureId által előírt függvényét a parameterList-nek megfelelő paraméterekkel és a returnValue-nak megfelelő visszatérési értékkel.
- **triEnqueueCall (SA→TE)**: az SA hívja meg, a tsiPortId porton keresztül meghívja a componentId által kijelölt komponens signatureId által előírt függvényét a parameterList-nek megfelelő paraméterekkel.
- **triEnqueueReply (SA→TE)**: az SA hívja meg, a tsiPortId porton keresztül meghívja a componentId által kijelölt komponens signatureId által előírt függvényét a parameterList-nek megfelelő paraméterekkel és a returnValue-nak megfelelő visszatérési értékkel.

Az 5.3. ábra szemlélteti a TE és a SUT közötti kommunikációban résztvevő triCommunication műveleteket és azok leképződését a SUT függvényhívásaira. Az MTC a SUT egy függvényét a call művelet segítségével hívhatja meg, aminek hatására a futtatókörnyezet meghívja a triCommunication interfész triCall függvényét a megfelelő paraméterekkel.

Ez a hívás képződik le az SA-n belül a SUT adott függvényének meghívására. A meghívott függvény válaszát (visszatérési érték, kimenő paraméterek) az SA a triEnqueueReply segítségével juttatja el az MTC-hez, amely a getreply művelettel fogadhatja a választ. A stub függvényeken keresztül zajló kommunikáció részletes leírása az 5.2.2. fejezetben található.

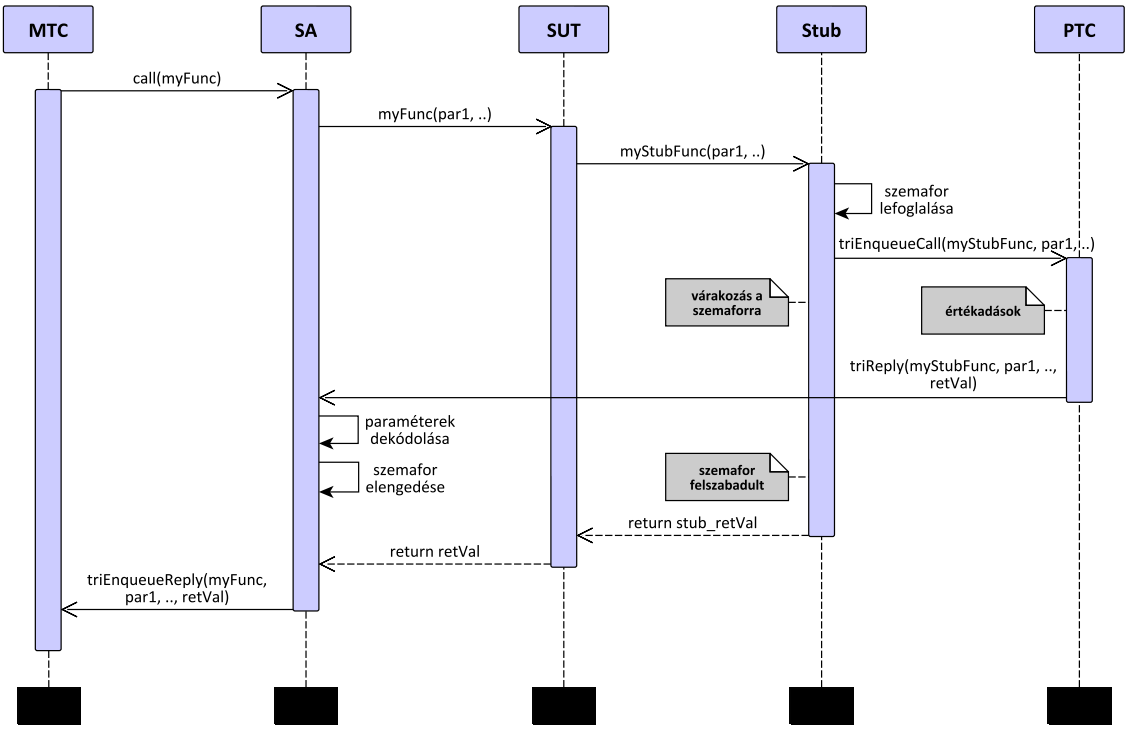


5.3. ábra. *Kommunikáció a TE és a SUT között az SA-n keresztül.*

5.2.1. Szálkezelés

A Java futtatókörnyezet sajátossága, hogy az egyes teszt esetek végrehajtása szálakon történik, amelyek futási időben, dinamikusan jönnek létre. Minden call és getcall parancs végrehajtásakor létrejön egy szál, amelyen az adott parancshoz tartozó blokk lefut.

Emiatt a SUT Adapter-nek is meg kell valósítania valamilyen egyszerű szálkezelési mechanizmust, hogy biztosítsa a TTCN-parancsok és a SUT függvényhívásainak helyes sorrendben történő, konkurens végrehajtását. Az 5.4. ábrán látható szekvenciadiagram által bemutatott példában szálkezelés és szemafor használata nélkül a stub függvény a triEnqueueCall meghívása után rögtön visszatérne, még azelőtt, hogy a visszatérési értéke vagy – ami még fontosabb – az esetleges kimeneti paraméterei érvényes értéket tartalmaznának.



5.4. ábra. *Szálkezelés működése a SUT Adapterben.*

Ez a probléma áthidalható, ha a SUT meghívott függvénye számára egy szálat, majd a függvénycsonkok meghívásakor lefoglaljuk a szemafort és várakozunk rá. A szemafor elengedése célszerűen a triReply függvényben (a PTC-ben esedékes értékadások befejeztével) történik, a visszatérési érték és egyéb kimeneti paraméterek dekódolása után. Ennek implementálásához pl. a MinGW pthread könyvtára biztosít eszközöket.

Az alábbi kódrészletek az FrSM_RequestComMode API függvény példáján keresztül mutatják be, hogy a triCall meghívásakor hogyan tároljuk a hívó paramétereket, valamint hogyan hozzuk létre a SUT függvénye számára a szálat:

```
TriStatus triCall(const TriComponentId   *componentId,
                  const TriPortId        *tsiPortId,
                  const TriAddress       *SUTaddress,
                  const TriSignatureId   *signatureId,
                  const TriParameterList *parameterList)
{
  /* A megfelelo fuggveny szignatura kivalasztasa */
  if (!strcmp(signatureId->objectName, "FrSM_RequestComMode"))
  {
    /* A hivo parameterek tarolasa a triEnqueueReply szamara */
    FrSM_RequestComModeParams.componentId   = dupTriComponentId(componentId);
    FrSM_RequestComModeParams.tsiPortId     = dupTriPortId(tsiPortId);
    FrSM_RequestComModeParams.SUTaddress    = dupTriAddress(SUTaddress);
    FrSM_RequestComModeParams.signatureId   = dupTriSignatureId(signatureId);
    FrSM_RequestComModeParams.parameterList = dupTriParameterList(parameterList);
    /* Szal deklaracio */
    pthread_t sutThread;
    /* Szal vegrehajtas elinditasa */
    pthread_create(&sutThread, 0, &run_FrSM_RequestComMode, 0);
    return TRI_OK;
  }
  .
  .
  /* Hibajelzes ha nincs talalat a fuggvenyre */
  return TRI_ERROR;
}
```

Az FrSM_RequestComMode számára létrehozott szálban történik a hívó paraméterek dekódolása, maga a függvényhívás, valamint a válasz elküldése a TE-nek a triEnqueueReply segítségével (itt használjuk fel a korábban elmentett hívó paramétereket):

```
void* run_FrSM_RequestComMode(void* p_)
{
  NetworkHandleType NetworkHandle;
  ComM_ModeType ComM_Mode;
  Std_ReturnType ret;
  /* Visszateresi ertek inicializalasa */
  TriParameter returnValue = {{NULL, 0, NULL}, TRI_IN};
  /* Hivo parameterek konvertalasa BinaryString formatumbol */
  constructor_NetworkHandleType(&NetworkHandle,
      FrSM_RequestComModeParams.parameterList->parList[0]->par.data);
  constructor_ComM_ModeType(&ComM_Mode,
      FrSM_RequestComModeParams.parameterList->parList[1]->par.data);
  /* Fuggvenyhivas es visszateresi ertek tarolasa */
  ret = FrSM_RequestComMode(NetworkHandle, ComM_Mode);
  /* A triEnqueueReply visszateresi ertekenek beallitasa, kodolas */
  returnValue.par.bits = 8*(code_Std_ReturnType((char**)&(returnValue.par.data),
                            ret));
  /* Valasz kuldese az MTC-nek */
  triEnqueueReply(FrSM_RequestComModeParams.tsiPortId,
                  FrSM_RequestComModeParams.SUTaddress,
                  FrSM_RequestComModeParams.componentId,
                  FrSM_RequestComModeParams.signatureId,
                  FrSM_RequestComModeParams.parameterList,
                  &returnValue);
  return NULL;
}
```

5.2.2. Stub függvények

Amennyiben a SUT meghívott függvényén belül további függvényhívás történik, azaz meghívja egy szomszédos modul (pl. FlexRay Interface) valamely API függvényét, az a megfelelő függvénycsonk által jut érvényre a TE-n belül (lásd 5.3. ábra). A meghívott függvénycsonk a triEnqueueCall segítségével jelzi a hozzá tartozó PTC-nek, hogy a hívás megtörtént, valamint elküldi a hívó paramétereket és a visszatérési értéket. A PTC a getcall művelet segítségével fogadja a hívást, majd a reply művelettel válaszol a SUT Adapter-nek, ezzel jelezve, hogy a TE-ben előírt műveletek végrehajtása sikeresen befejeződött.

Az alábbi kódrészletek a FlexRay Interface modul FrIf_GetTransceiverWUReason API függvényének példáján keresztül mutatják be, hogy egy stub függvény meghívásakor hogyan adjuk át a hívó paramétereket a PTC-nek, hogyan kezeljük a szemafort, valamint hogyan dekódoljuk az FrIfTrcvWUReasonPtr kimeneti paramétert:

```
Std_ReturnType FrIf_GetTransceiverWUReason(uint8 FrIf_CtrlIdx,
                      Fr_ChannelType FrIf_ChnlIdx,
                      FrTrcv_TrcvWUReasonType* FrIfTrcvWUReasonPtr)
{
  TriParameterList pl;
  TriSignatureId sigId;
  TriPortId tsiPort;
  TriParameter** parList = 0;

  /* triEnqueueCall parametereinek inicializalasa */
  memset(&sigId, 0, sizeof(sigId));
  memset(&pl, 0, sizeof(pl));
  memset(&tsiPort, 0, sizeof(TriPortId));
  /* Port informaciok megadasa (hard-coded) */
  tsiPort.portName = "tsiPort";
  tsiPort.portIndex = -1;
  /* Fuggveny szignatura beallitasa */
  sigId.objectName = "FrIf_GetTransceiverWUReason";

  /* Memoriafoglalas a parameterlistanak */
  pl.parList = (TriParameter **) malloc(sizeof(TriParameter*) * 3);
  pl.length = 3;
  parList = pl.parList;

  /* Az FrIf_CtrlIdx bemeno parameter kodolasa es iranyanak megadasa */
  parList[0] = (TriParameter*) malloc(sizeof(TriParameter));
  memset(&(parList[0]->par), 0, sizeof(BinaryString));
  parList[0]->mode = TRI_IN;
  parList[0]->par.bits = 8*(code_uint8((char**)&(parList[0]->par.data),
                FrIf_CtrlIdx));
  /* Az FrIf_ChnlIdx bemeno parameter kodolasa es iranyanak megadasa */
  parList[1] = (TriParameter*) malloc(sizeof(TriParameter));
  memset(&(parList[1]->par), 0, sizeof(BinaryString));
  parList[1]->mode = TRI_IN;
  parList[1]->par.bits = 8*(code_Fr_ChannelType((char**)&(parList[1]->par.data),
                FrIf_ChnlIdx));
```

```
  /* Az FrIfTrcvWUReasonPtr kimeno parameter inicializalasa es iranyanak megadasa */
  parList[2] = (TriParameter*) malloc(sizeof(TriParameter));
  memset(&(parList[2]->par), 0, sizeof(BinaryString));
  parList[2]->mode = TRI_OUT;

  /* Szemafor lefoglalasa */
  spin_semaphore = 0;
  /* A hivott komponens (PTCFrIf) parametereinek lekerdezese a triMap soran
     letrehozott listabol */
  struct IdListEntry entry = getEntry("pt_FrIf");
  /* A komponens sigId altal meghatarozott fuggvenyenek meghivasa */
  triEnqueueCall(entry.gPortId, NULL, entry.gComponentId, &sigId, &pl);
  /* Varakozas a szemaforra (a komponens valaszara) */
  spin_semaphore_wait();

  /* Az FrIfTrcvWUReasonPtr kimeno parameter dekodolasa */
  decode_p_FrTrcv_TrcvWUReasonType(
    FrIf_GetTransceiverWUReasonParams.parameterList->parList[2]->par.data,
    &(FrIf_TrcvWUReasonPtr));

  /* Memoria felszabaditasa */
  free(parList[0]);
  free(parList[1]);
  free(parList[2]);
  free(pl.parList);

  /* A globalis valtozo mostmar ervenyes erteket tartalmaz, stub visszater */
  return retval_FrIf_GetTransceiverWUReason;
}
```

A getcall parancshoz tartozó blokkban definiált műveletek végrehajtása után a PTC meghívja a triReply függvényt. Ebben tároljuk a kimenő paramétert, dekódoljuk a visszatérési értéket és tároljuk a megfelelő globális változóban, majd elengedjük a szemafort, hogy a stub függvény hátralévő része lefuthasson:

```
TriStatus triReply(const TriComponentId   *componentId,
                   const TriPortId        *tsiPortId,
                   const TriAddress       *SUTaddress,
                   const TriSignatureId   *signatureId,
                   const TriParameterList *parameterList,
                   const TriParameter     *returnValue)
{
  /* A megfelelo fuggveny szignatura kivalasztasa */
  if (!strcmp(signatureId->objectName,"FrIf_GetTransceiverWUReason"))
  {
    /* Parameterek tarolasa a stub fuggvenyben torteno felhasznalasra */
    FrIf_GetTransceiverWUReasonParams.parameterList =
                      dupTriParameterList(parameterList);
    /* Visszateresi ertek dekodolasa es tarolasa a globalis valtozoban */
    decode_Std_ReturnType(returnValue->par.data,
                &(retval_FrIf_GetTransceiverWUReason));
    /* Szemafor elengedese */
    spin_semaphore = 1;
    return TRI_OK;
  }
  .
  .
  .
```

```
  /* Hibajelzes ha nincs talalat a fuggvenyre */
  return TRI_ERROR;
}
```

5.3. Konfigurációs paraméterek illesztése

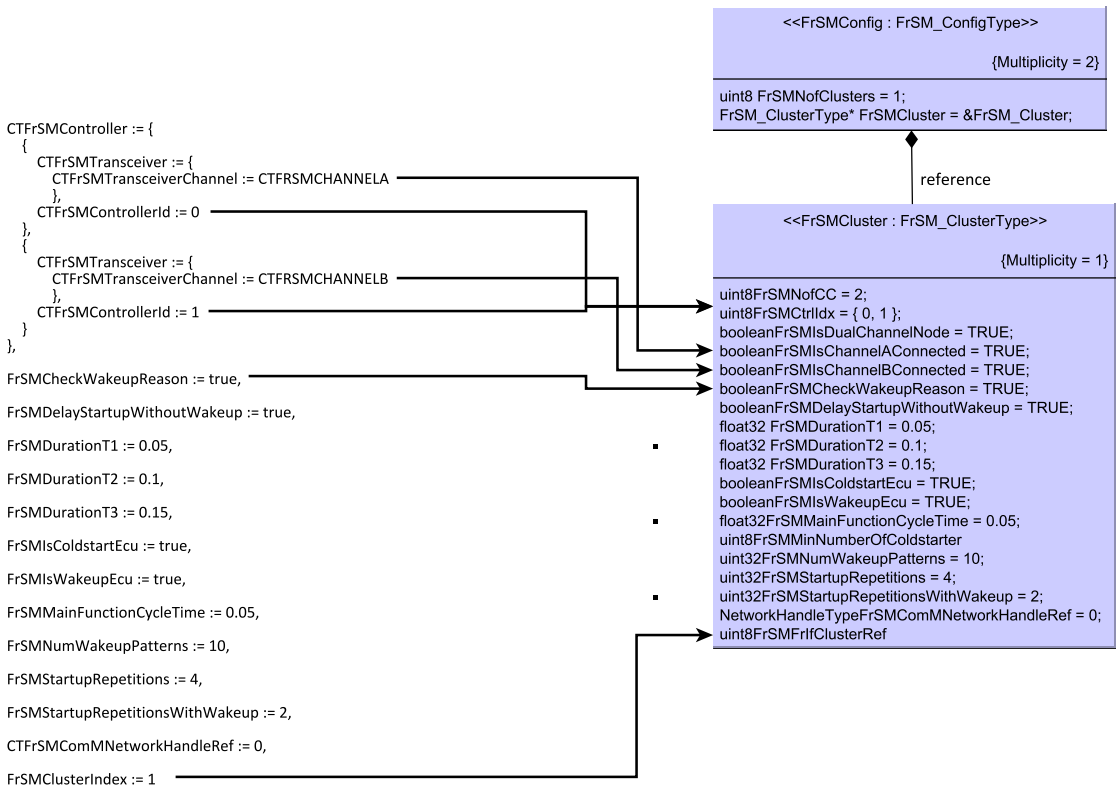
A FlexRay State Manager modul AUTOSAR teszt csomagja négyféle konfigurációs paraméter együttest definiál, amelyekre a teszt eseteket le kell futtatni a megfelelőség tanúsításához. Azt, hogy ezek közül futtatáskor melyiket használjuk, fordítás előtt kell kiválasztanunk a megfelelő konfigurációt tartalmazó modul importálásával:

```
import from FrSMConfigParameters_1 all;
```

Ezzel analóg módon a modul konfigurációban egy makró segítségével szintén fordítás előtt állíthatjuk be, hogy melyik struktúrát akarjuk használni a tesztelés során:

```
#if (defined (USE_FRSM_PBCFG_PARAMETERS_1))
.
.
#endif
```

A konfiguráció leírásának AUTOSAR CTS-ben használt formátuma néhány tekintetben eltér a C modulban alkalmazottól. Az 5.5. ábra szemlélteti, hogy hogyan képezhetők le a konfigurációs modulokban megadott értékek a C struktúrák egyes tagjainak értékére.



5.5. ábra. *Az AUTOSAR CTS konfiguráció leképződése modul konfigurációra.*

Például az FrSM_Cluster struktúra kontroller- és transceiver-specifikus paramétereinek meghatározása a CTFrSMController modulparaméter alapján történhet: a CTFrSMControllerId paraméterek meghatározzák a kontroller indexeket a cluster-en belül, multiplicitásuk pedig megadja a kontrollerek számát.

A transceiver-ekre vonatkozó CTFrSMTransceiverChannel paraméterek meghatározzák, hogy az A, a B vagy mindkét FlexRay csatornára csatlakozik-e az ECU, multiplicitásuk pedig meghatározza, hogy az FrSMIsDualChannelNode paraméter igaz vagy hamis.

6. fejezet

CUnit-alapú teszt csomag

Amint arról az 1.1. fejezetben szó volt, az AUTOSAR konzorcium a BSW modulokhoz csak a 4.0 verzió 2-es revíziójáig bezárólag adott ki megfelelőségi teszt csomagokat. Figyelembe véve, hogy a tesztelendő modul már a specifikáció *3-as revíziója* alapján készült, nyilvánvaló, hogy modultesztelés terén nem lehet hosszútávon számítani a konzorcium ilyen jellegű támogatására. A diplomaterv készítése közben kiderült ugyanis, hogy anyagi megfontolásból az AUTOSAR a 2-es revízió után kiadott szabványokhoz már *nem fog kiadni* megfelelőségi teszt csomagot. Ehelyett átvételi teszteket fog kibocsátani, azonban azokat is csak specifikáció szinten. A FlexRay State Manager modul esetében a két revízió között csak kevés, nem túl jelentős eltérés van, ezért az AUTOSAR CTS-sel történő tesztelésnek még van létjogosultsága.

Nyilvánvalóan felmerül az igény az AUTOSAR által kibocsátott teszt csomagok kiváltására valamilyen házon belül készített teszt implementáció segítségével. A TTCN-alapú tesztek helyettesítésére az optimális megoldás a CUnit használata, hiszen ehhez nem szükséges sem új fordító és/vagy szoftvercsomag beszerzése, sem egy másik nyelv elsajátítása – aki tud C-ben fejleszteni, az kis ráfordítással tudni fog CUnit-tal tesztelni is. Ugyancsak nyomós érv mellette, hogy *ingyenes*, nyílt forráskódú eszköz.

A CUnit egy C nyelvű tesztek írására, adminisztrálására és futtatására alkalmas eszköz, amely statikus könyvtárakból épül fel és a felhasználói kódhoz linkelődik. Logikai feltételek kiértélése CUnit-ban *assertion* segítségével lehetséges. A leggyakrabban használt assertion-ök [13]:

```
/* Pass, ha az ertek nem nulla */
CU_ASSERT_TRUE(val);
/* Pass, ha az ertek nulla */
CU_ASSERT_FALSE(val);
/* Pass, ha a ket (egesz) ertek egyenlo */
CU_ASSERT_EQUAL(actual, expected);
```

```
/* Pass, ha a ket (egesz) ertek nem egyenlo */
CU_ASSERT_NOT_EQUAL(actual, expected);
/* Pass, ha ugyanarra a cimre mutatnak */
CU_ASSERT_PTR_EQUAL(actual, expected);
```

A teszteket négyféle módban futtathatjuk:

- **Automated**: a Test Registry-hez hozzáadott összes teszt eset futtatása a háttérben, a tesztek eredménye egy XML fájba kerül.
- **Basic**: a felhasználói interfész nem interaktív, a tesztek eredménye a standard kimeneten jelenik meg, lehetőséget biztosít csak meghatározott teszt esetek futtatására, valamint a megjelenített információ részletességének beállítására.
- **Interactive console**: interaktív interfész, amely minden végrehajtás előtt felkínálja az egyes opciókat: tesztek futtatása, egy adott teszt csomag kiválasztása, teszt csomagok listázása, hibák megjelenítése, kilépés.
- **Interactive curses**: az Interactive console módhoz hasonló, csak Unix rendszer alatt használható.

A legnagyobb rugalmasságot a Basic mód biztosítja, ezért a tesztelés során ezt célszerű használni verbose módban, ekkor minden elérhető információt megjelenít a konzolon.

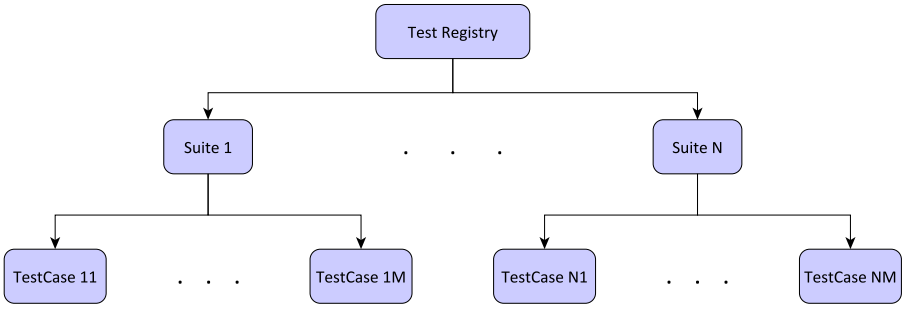
6.1. Tesztkörnyezet kialakítása

A CUnit nyilvánvaló előnye a TTCN-3 tesztkörnyezetekkel szemben, hogy használatához nincs szükség speciális illesztőkódra. Hátránya az AUTOSAR CTS-ekhez képest, hogy ebben az esetben a teszt eseteket nem kapjuk készen egy harmadik féltől, azok megtervezése, specifikálása és implementálása a fejlesztő/tesztelő feladata.

AUTOSAR BSW modulok megfelelőségi tesztelése esetén a vizsgálódás középpontjában a modulnak a környezete felé megvalósított funckionalitása áll, vagyis (amint arról az 1.1. fejezetben szó volt) az, hogy az adott szoftverkomponens a szabvány által előírt interfészeken keresztül az elvárt interakciókat bonyolítja-e le a szomszédos modulokkal. Ennek ellenőrzésére létre kell hozni egy olyan tesztkörnyezetet, amely egyrészt biztosítja a megfelelő függvénycsonkokat a SUT számára, másrészt amelyben lehetőség nyílik azok meghívását regisztrálni és a későbbiekben vizsgálni.

A CUnit tesztrendszer hierarchiáját szemlélteti a 6.1. ábra. A végrehajtandó teszt csomagokat a Test Registry nevű leíró tartalmazza, ehhez hozzá kell adni az összes

olyan suite-ot, amelyet futtatni szeretnénk. Egy suite egy vagy több teszt esetből állhat, végrehajtásuk szekvenciálisan történik.



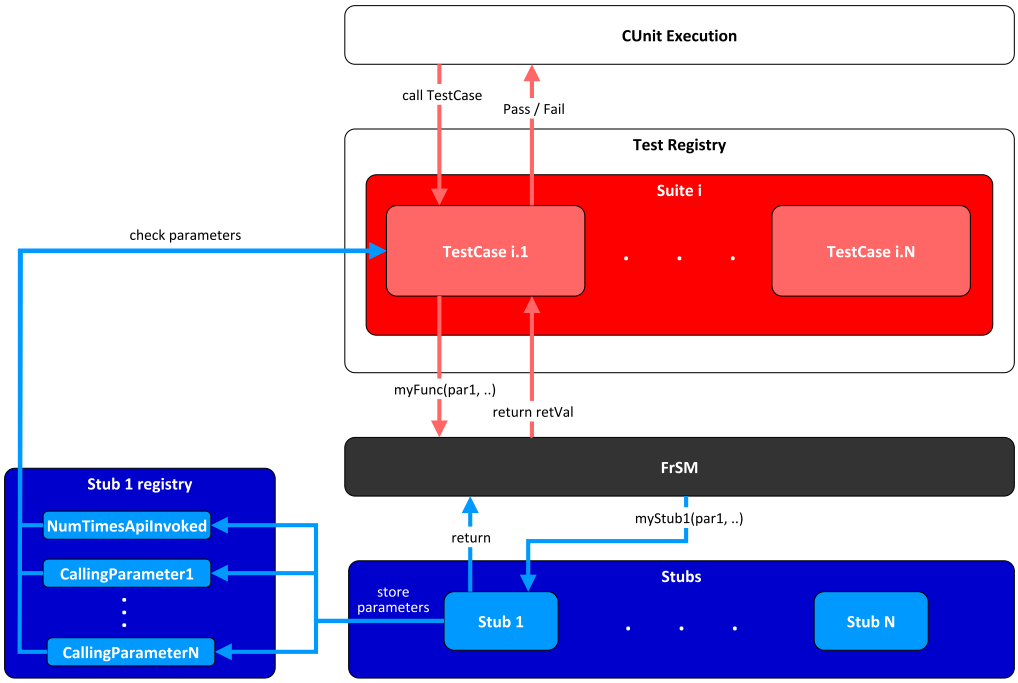
6.1. ábra. *CUnit tesztrendszer általános felépítése.*

A CUnit futtató működése során kiválasztja a következő teszt csomagot a registry-ből, majd annak inicializálása után végrehajtja az első teszt esetet az adott csomagból. Ha a teszt esetben található összes assertion sikeres volt, beállítja a státuszát pass-ra és folytatja a végrehajtást a következő teszt esettel. Sikertelen assertion esetén a futás nem áll meg (kivéve ha az assertion típusa FATAL, de ennek alkalmazása nem javasolt), a soron következő assertion-ök is kiértékelődnek, a teszt eset státusza azonban fail lesz. Egy teszt csomag végére érve a futtató meghívja annak cleanup függvényét, majd kiválasztja a soron következő teszt csomagot és így tovább.

Annak érdekében, hogy a SUT egy meghívott függvénye által kezdeményezett függvényhívások nyomonkövethetők legyenek, a SUT összes interfésze számára létre kell hoznunk egy függvénycsonkot, amely valamilyen globális leíróban tárolja a hívások számát és a hívó paramétereket (lásd 6.2. ábra).

Ez a globális leíró célszerűen egy struktúra lehet, amely egyedi az adott függvénycsonkra nézve. Például a Det_ReportError függvényhez az alábbi leíró tartozik:

```
/* A fuggveny szignaturaja */
Std_ReturnType Det_ReportError(uint16 ModuleId, uint8 InstanceId, uint8 ApiId,
                               uint8 ErrorId);
/* A hozza tartozo leiro */
typedef struct {
  /* Fuggvenyhivasok szama */
    unsigned int NumTimesApiInvoked;
  /* Hivo parameterek */
    uint16       ModuleId;
    uint8        InstanceId;
    uint8        ApiId;
    uint8        ErrorId;
} Det_ReportError_RegType;
```



6.2. ábra. *Az FrSM modul tesztkörnyezete.*

A teszt esetekben az assertion-ök elvégezhetők a globális leíró elemeire. Azon függvénycsonkok esetén, amelyeket a SUT egy API függvénye legfeljebb egyszer hív meg, elegendő csak egy paramétert tárolni a leíróban (lásd alább), míg egyéb esetekben a paraméterek a számukra létrehozott tömbben tárolhatók (lásd 6.3.2. fejezet).

```
/* A fuggvenyhivasok szamanak ellenorzese */
CU_ASSERT_EQUAL(Det_ReportError_Registry.NumTimesApiInvoked, 1);
/* A hivo parameterek ellenorzese */
CU_ASSERT_EQUAL(Det_ReportError_Registry.ApiId,      FrSM_MainFunction_API_ID);
CU_ASSERT_EQUAL(Det_ReportError_Registry.ErrorId,    FRSM_E_CLUSTER_STARTUP);
CU_ASSERT_EQUAL(Det_ReportError_Registry.InstanceId, FRSM_INSTANCE_ID);
CU_ASSERT_EQUAL(Det_ReportError_Registry.ModuleId,   FRSM_MODULE_ID);
```

Sok esetben az állapotgép valamely átmenetéhez tartozó őrfeltétel teljesülése függ egy vagy több függvénycsonk valamely kimeneti paraméterének értékétől. Például az FRSM_STARTUP→FRSM_WAKEUP átmenet függ a WUReason feltételtől, amely futási időben értékelődik ki attól függően, hogy az adott cluster transceivereit a buszról ébresztették-e fel és ha igen, melyik csatornán.

Ezeket az információkat a FlexRay Interface modul alábbi függvényeinek kimeneti paraméterei (FrIf_TrcvWUReasonPtr és FrIf_WakeupRxStatusPtr) tartalmazzák (lásd [6] és [14]):

```
Std_ReturnType FrIf_GetTransceiverWUReason(uint8 FrIf_CtrlIdx,
                                    Fr_ChannelType FrIf_ChnlIdx,
                                    FrTrcv_TrcvWUReasonType* FrIf_TrcvWUReasonPtr);

Std_ReturnType FrIf_GetWakeupRxStatus(uint8 FrIf_CtrlIdx,
                                      uint8* FrIf_WakeupRxStatusPtr);
```

A bejárt trajektória is ettől a feltételtől függ, ugyanis NO_WU_BY_BUS esetén a T01(a) vagy T01(b) átmenet következik be, míg PARTIAL_WU_BY_BUS esetén a T01(c). A FlexRay Interface modul fent említett függvényeinek csonkjai tehát olyan kimeneti paramétereket kell, hogy szolgáltassanak, amelyek miatt a WUReason NO_WU_BY_BUS-ra (ha T01(a) vagy T01(b) átmenetet akarunk előidézni) vagy PARTIAL_WU_BY_BUS-ra (ha T01(c) átmenetet akarunk előidézni) fog kiértékelődni.

Ahhoz, hogy az FrSM_MainFunction meghívásakor az előidézni kívánt állapotátmenetnek megfelelő őrfeltételeket biztosítani tudjuk a függvénycsonkokon keresztül, szükség van néhány globális „dummy" változóra, amelyek értékét az adott stub paraméterlistán visszaadja:

```
/* A WUReason kiertekeleseben van szerepe */
uint8 DummyWakeupRxStatus;
/* A lowNumberOfColdstarters kiertekeleseben van szerepe */
uint8 DummyNumOfStartupFrames;
/* Az AllChannelsAwake kiertekeleseben van szerepe */
uint32 DummyWakeupChannel;
/* POC allapot, ami alapjan a vPOC strukturat feltoltjuk */
Fr_POCStatusType DummyPOCStatus;
/* A WUReason kiertekeleseben van szerepe */
FrTrcv_TrcvWUReasonType DummyWUReason;
```

Ennek megfelelően az FrIf_GetPOCStatus függvénycsonk az itt látható módon tárolja a hívó paramétert, beállítja a kimeneti paramétert és növeli a hívásokat nyilvántartó számlálót:

```
Std_ReturnType FrIf_GetPOCStatus(uint8 FrIf_CtrlIdx,
                                 Fr_POCStatusType* FrIf_POCStatusPtr)
{
  /* Kontroller index tarolasa a leiroban */
  FrIf_GetPOCStatus_Registry.CtrlIdx[FrIf_GetPOCStatus_Registry.NumTimesApiInvoked]
    = FrIf_CtrlIdx;
  /* Mutato beallitasa a dummy strukturara */
  FrIf_POCStatusPtr = &DummyPOCStatus;
  /* Fuggvenyhivas szamlalo novelese */
  FrIf_GetPOCStatus_Registry.NumTimesApiInvoked += 1;
  return E_OK;
}
```

Amint arról ebben a fejezetben már szó volt, egy teszt csomag végrehajtásának első lépéseként a futtató meghívja annak inicializáló függvényét, utolsó lépéseként pedig a cleanup függvényét. Ezért minden suite-hoz kötelező megadni egy-egy init és cleanup függvényt, amikor felvesszük a Test Registry-be. Mindkét függvény visszatérési értéke integer típusú kell, hogy legyen (sikeres végrehajtás esetén 0), paraméterük nem lehet, feladatuk az adott teszt csomag követelményeitől függ. Például, ha a teszt esetek során fájlba írunk vagy onnan olvasunk, akkor az init függvényben célszerű megnyitni vagy létrehozni és megnyitni a fájlt, a cleanup függvényben pedig bezárni. Jelen esetben a függvénycsonkokhoz tartozó leírók inicializálása történik

ezekben a függvényekben:

```
int init_Suite_Default(void)
{
  init_Registries();
  return 0;
}

int clean_Suite_Default(void)
{
  init_Registries();
  return 0;
}
```

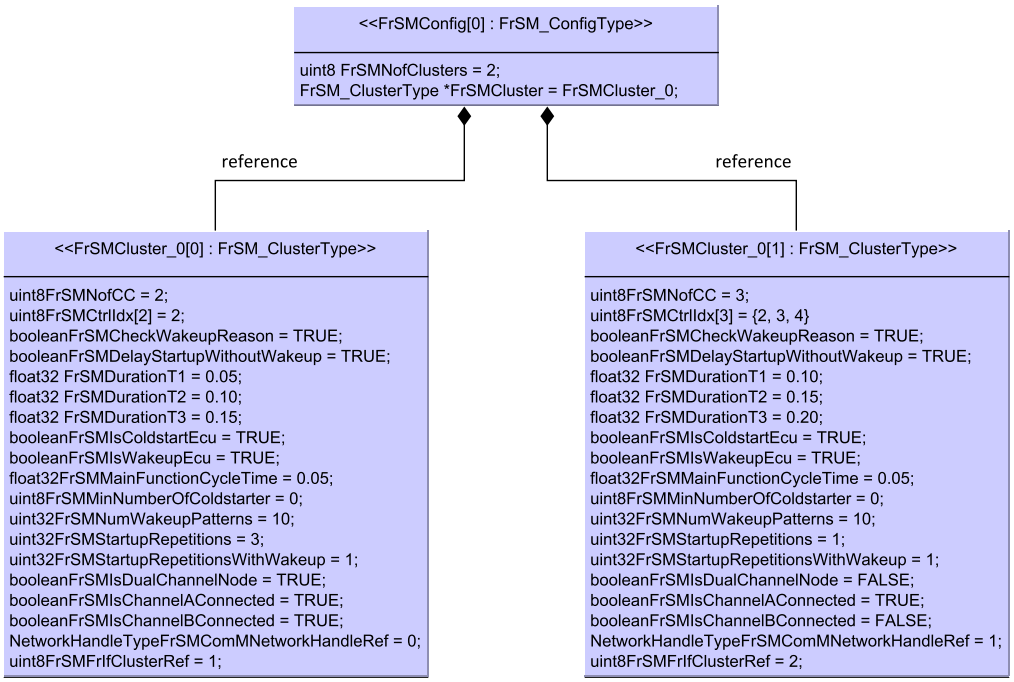
Az init_Registries függvény az összes függvénycsonk leírójának minden paraméterét beállítja az alapértelmezett értékre:

```
void init_Registries(void)
{
  unsigned int initIdx;

  BswM_FrSM_CurrentState_Registry.NumTimesApiInvoked = 0U;
  .
  .
  FrNm_StartupError_Registry.NumTimesApiInvoked      = 0U;

  for (initIdx=0; initIdx<CUNIT_NOF_PARAMETERS; initIdx++)
  {
    BswM_FrSM_CurrentState_Registry.Network[initIdx]      = 0U;
    BswM_FrSM_CurrentState_Registry.CurrentState[initIdx] = FRSM_BSWM_READY;
    .
    .
    FrNm_StartupError_Registry.NetworkHandle[initIdx] = 0U;
  }
}
```

6.2. Modul konfiguráció

A tesztelés során használt konfiguráció struktúráján változtattam az AUTOSAR CTS-hez képest annak érdekében, hogy a teszt eseteket egy futtatás alkalmával több konfigurációs szettre is végre lehessen hajtani. Ennek megfelelően azt, hogy melyik konfiguráció készletet használjuk az 5.3. fejezetben alkalmazott módszerrel ellentétben nem egy makró választja ki, tehát nem dől el fordítási idő előtt.

Az FrSM_Config container multiplicitása (vagyis a konfigurációs szettek száma) négy, mindegyikhez eltérő számú és paraméterezésű cluster tartozik. A másik container (FrSM_General) mindössze két paramétert tartalmaz (lásd 2.7. ábra). Ezek pre-compile paraméterek, vagyis makrók formájában adottak. Ezért – bár ez a kevésbé elegáns megoldás – egyszerűbb ha a tesztek futtatása után kézzel átállítjuk ezeket a paramétereket, majd újra lefuttatjuk a teszt csomagot. Így elegendő csupán az FrSM_Config-ból létrehozni több példányt és a teszteket mindegyikre futtatni.



6.3. ábra. *A modul első konfigurációs paraméterkészlete.*

Példaképp a 6.3. ábrán látható az első (0. indexű) konfigurációs szett UML osztálydiagramja. A konfigurációs paraméterkészleteket úgy állítottam össze az AUTOSAR CTS alapján, hogy ne legyen olyan teszt eset, amely egyszer sem fut le az ellenőrzési pontokon való fennakadás miatt.

A fentiekben bemutatott konfigurációs struktúra felhasználásával a teszt esetek általános felépítése a következő:

```
void Test_FrSM_<TestCaseId>(void)
{
  unsigned int ConfigSetIdx, ClstrIdx;
  unsigned int NofClusters;

  /* Az osszes konfiguracios szettre */
  for (ConfigSetIdx=0; ConfigSetIdx<FRSM_NOF_CONFIG_SETS; ConfigSetIdx++)
  {
    /* SUT inicializacioja */
    FrSM_Init(&(FrSM_Config[ConfigSetIdx]));
    NofClusters = FrSM_Config[ConfigSetIdx].FrSMNofClusters;
    /* Az adott konfiguracioban talalhato osszes cluster-re */
    for (ClstrIdx=0; ClstrIdx<NofClusters; ClstrIdx++)
    {
      /* Globalis leirok inicializacioja */
      init_Registries();

      /* Teszt funkcionalitas: ellenorzesi es verifikacios pontok */
      .
      .
    } /* for (ClstrIdx=0; ... ) */
  } /* for (ConfigSetIdx=0; ... ) */
}
```

Vagyis egy adott teszt eset mindig lefut az összes konfigurációs paraméterkészlet összes cluster-ére. A továbbiakban terjedelmi okok miatt az egyes teszt esetek bemutatásánál a példákban csak a teszt funkcionalitást megvalósító részek (belső for ciklus magja) fognak szerepelni.

6.3. A teszt csomag felépítése

A FlexRay State Manager modul megfelelőségi teszt csomagjának specifikációja kiindulási alapként szolgált a teszt implementáció során, azonban bizonyos szempontból mégis célszerű kicsit változtatni a teszt esetek szervezésén és implementációján. Mindenképp érdemes tanulmányozni az AUTOSAR által kibocsátott teszt specifikációt, ugyanis ez elvileg – az ott felsorolt kivételektől eltekintve – 100%-os követelmény lefedettséget biztosít [15]. Azonban néhány igen fontos szempont indokolttá teszi, hogy bizonyos mértékben módosítsuk a teszt esetek szervezését:

- A 3-as revízió specifikációja tartalmaz néhány olyan új követelményt a 2-eshez képest, amelyet az AUTOSAR CTS nem fed le.
- Az AUTOSAR CTS nem fedi le az API függvények fejlesztési hibák (Development Error) detektálására vonatkozó követelményeit.
- Az AUTOSAR CTS hatékonysága csak a követelmény-fedettség révén mérhető, más tesztelési módszerekkel az utasítás lefedettség és a döntési feltétel lefedettség is javítható.

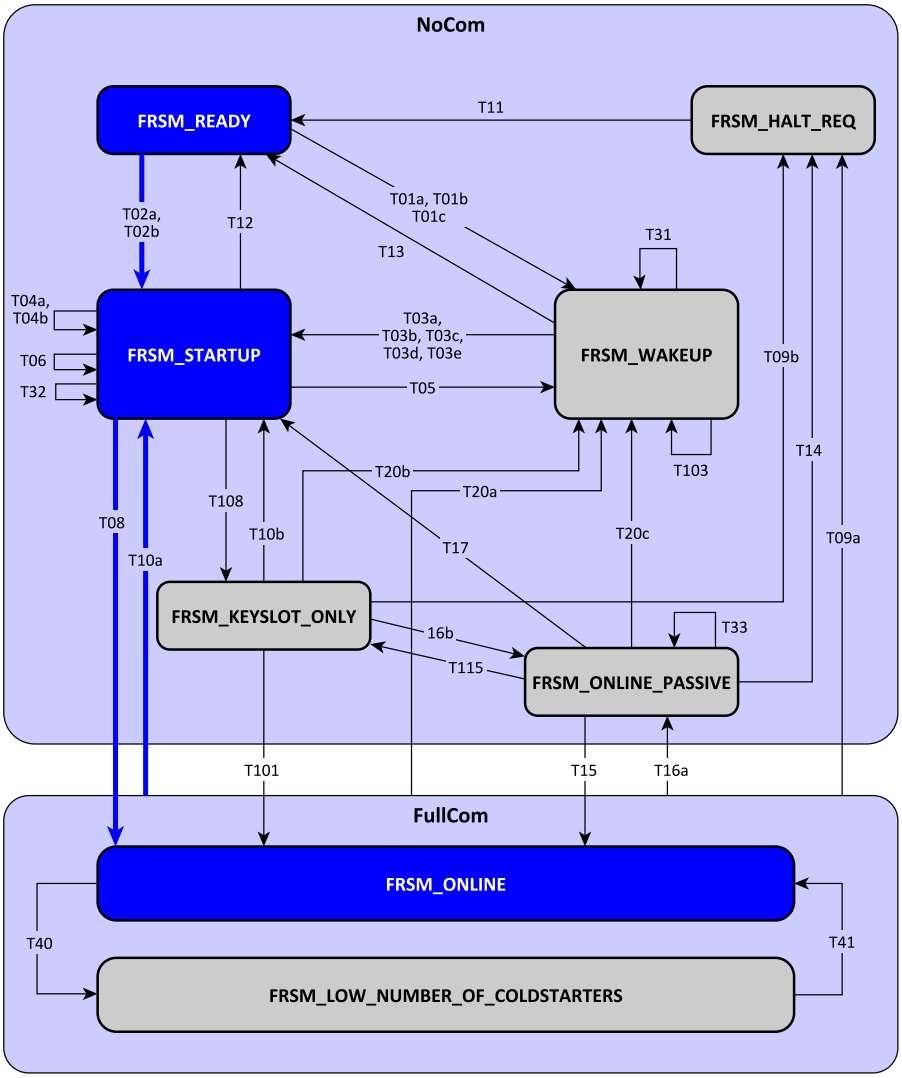
Az AUTOSAR által specifikált megfelelőségi teszt csomag szervezési elve az, hogy minden teszt esetben egy *trajektóriát* járunk be az állapottérképen és ennek során pontról pontra ellenőrizzük az oda vonatkozó követelmények teljesülését. Például a TC_FRSM_0069a teszt eset, amely a T10(a) állapotátmenet során végrehajtandó aktivitások ellenőrzésére szolgál, az alábbi trajektórián vezet végig az állapottérképen (lásd 6.4. ábra): FRSM_READY → FRSM_STARTUP → FRSM_ONLINE → FRSM_STARTUP.

Közös tulajdonsága az AUTOSAR CTS-nek és a CUnit teszt csomagnak, hogy minden teszt eset tartalmaz *ellenőrzési pontokat* (check point) és *verifikációs pontokat* (verification point). Előbbi általában egy olyan konfigurációs elem értékének vizsgálatát jelenti, amely alapvetően befolyásolja azt, hogy az adott állapotátmenet bekövetkezhet-e (pl. FrSMIsDualChannelNode), utóbbi pedig egy elemi követelmény ellenőrzését, amelynek a specifikáció szerint teljesülnie kell (általában egy aktivitás végrehajtása).

Egy teszt esetet akkor tekintünk sikeresnek (pass), ha minden teljesülő ellenőrzési ponthoz tartozó verifikációs pont sikeres volt. Ha van olyan ellenőrzési pont, amely teljesül, de egy hozzá tartozó verifikációs pont sikertelen, akkor a teszt eset sikertelen (fail). A TTCN-3 ezen felül lehetőséget biztosít a nem teljesülő konfigurációs ellenőrzési pontok megkülönböztetésére a none státusz segítségével. A 6.4. ábrán látható teszt esetben az alábbi lépéseken keresztül vezetjük végig az állapotgépet az ábrán látható trajektória mentén:

1. FrSM_Init meghívása az adott konfigurációs szettre.
2. Az alábbi lépések végrehajtása az összes cluster-en:
   (a) FrSM_RequestComMode meghívása COMM_FULL_COMMUNICATION paraméterrel.
   (b) Ellenőrzési pont (T02(a) és T02(b) átmenet feltételei): FrSMCheckWakeupReason == TRUE és FrSMIsWakeupEcu == FALSE.
   (c) WakeupRxStatus beállítása mindkét csatornára (ALL_WU_BY_BUS).
   (d) FrSM_SetEcuPassive meghívása FALSE paraméterrel.
   (e) FrSM_MainFunction meghívása az adott cluster-re (T02(a) és T02(b) átmenetek előidézése).
   (f) FrSM_RequestComMode meghívása COMM_FULL_COMMUNICATION paraméterrel.
   (g) POC Status beállítása a T08 átmenetnek megfelelően.
   (h) FrSM_MainFunction meghívása az adott cluster-re (T08 átmenet előidézése).
   (i) FrSM_RequestComMode meghívása COMM_FULL_COMMUNICATION paraméterrel.
   (j) POC Status beállítása a T10 átmenetnek megfelelően.
   (k) Ellenőrzési pont (T10(a) átmenet feltételei): FrSMCheckWakeupReason == TRUE.
   (l) FrSM_MainFunction meghívása az adott cluster-re (T10(a) átmenet előidézése).

A teszt esetek fent bemutatott koncepcióját, vagyis az állapottérkép bejárását alapvetően nem változtattam meg a CUnit teszt csomagokban. Külön suite-okban teszteltem azonban az API függvények viselkedését megfelelő és hibás hívó paraméterek esetén (lásd 6.3.1. fejezet), valamint az állapotátmeneteket (lásd 6.3.2. fejezet). Célul tűztem ki a 100%-os utasítás lefedettség és a legalább 90%-os döntési feltétel lefedettség elérését, ezért az összes állapotátmenetet külön teszteltem.



6.4. ábra. *A TC_FRSM_0069a teszt eset során bejárt állapotok.*

6.3.1. API függvények teszt esetei

A modul egyes API függvényeinek teszt eseteit külön suite-okba csoportosítottam. Minden suite tartalmaz egy teszt esetet, amely a függvény sikeres futását teszteli, valamint egy-egy teszt esetet a szabvány által specifikált lehetséges érvénytelen hívások tesztelésére [6]:

FrSM_Init

- Ha FrSMDevErrorDetect == ON és a függvény bemenő paramétere null pointer, a függvény hibát jelez a Det felé.

FrSM_RequestComMode

- Ha a függvényt COMM_SILENT_COMMUNICATION paraméterrel hívják meg, visszatér E_NOT_OK-kal anélkül, hogy bármilyen műveletet végrehajtana, ezen felül hibát jelez a Det felé, ha FrSMDevErrorDetect == ON.

- Ha FrSMDevErrorDetect == ON és a függvény érvénytelen paraméterrel vagy paraméterekkel lett meghívva, hibát jelez a Det felé és E_NOT_OK-kal tér vissza anélkül, hogy bármilyen műveletet végrehajtana.

- Ha FrSMDevErrorDetect == ON és a modul inicializációja nem történt meg, a függvény hibát jelez a Det felé és E_NOT_OK-kal tér vissza anélkül, hogy bármilyen műveletet végrehajtana.

FrSM_GetCurrentComMode

- Ha FrSMDevErrorDetect == ON és a függvény érvénytelen NetworkHandle paraméterrel lett meghívva, hibát jelez a Det felé és E_NOT_OK-kal tér vissza anélkül, hogy bármilyen műveletet végrehajtana.

- Ha FrSMDevErrorDetect == ON és a ComM_ModePtr paraméter null pointer, a függvény hibát jelez a Det felé és E_NOT_OK-kal tér vissza anélkül, hogy bármilyen műveletet végrehajtana.

- Ha FrSMDevErrorDetect == ON és a modul inicializációja nem történt meg, a függvény hibát jelez a Det felé és E_NOT_OK-kal tér vissza anélkül, hogy bármilyen műveletet végrehajtana.

FrSM_AllSlots

- Ha FrSMDevErrorDetect == ON és a függvény érvénytelen NetworkHandle paraméterrel lett meghívva, hibát jelez a Det felé és E_NOT_OK-kal tér vissza anélkül, hogy bármilyen műveletet végrehajtana.

- Ha FrSMDevErrorDetect == ON és a modul inicializációja nem történt meg, a függvény hibát jelez a Det felé és E_NOT_OK-kal tér vissza anélkül, hogy bármilyen műveletet végrehajtana.

FrSM_SetEcuPassive

- Ha FrSMDevErrorDetect == ON és a modul inicializációja nem történt meg, a függvény hibát jelez a Det felé és E_NOT_OK-kal tér vissza anélkül, hogy bármilyen műveletet végrehajtana.

FrSM_MainFunction

- Ha FrSMDevErrorDetect == ON és a modul inicializációja nem történt meg, a függvény hibát jelez a Det felé és E_NOT_OK-kal tér vissza anélkül, hogy bármilyen műveletet végrehajtana.

Külön suite-ba tartoznak az inicializálatlan viselkedést vizsgáló teszt esetek. Ezeket kell futtatni először, hiszen az összes többi teszt eset elején meg kell hívni az FrSM_Init függvényt:

```
CU_TestInfo Tests_FrSM_UnInit[] = {
    {"Test_FrSM_RequestComModeUnInit",    Test_FrSM_RequestComModeUnInit},
    {"Test_FrSM_GetCurrentComModeUnInit", Test_FrSM_GetCurrentComModeUnInit},
    {"Test_FrSM_AllSlotsUnInit",          Test_FrSM_AllSlotsUnInit},
    {"Test_FrSM_SetEcuPassiveUnInit",     Test_FrSM_SetEcuPassiveUnInit},
    {"Test_FrSM_MainFunctionUnInit",      Test_FrSM_MainFunctionUnInit},
    CU_TEST_INFO_NULL
  };
```

Az API függvények teszt csomagjai közül az FrSM_RequestComMode függvényhez tartozó fedi le a legtöbb teszt esetet. Ezért az API tesztelés bemutatása ezek alapján fog történni. A suite az alábbi teszt esetekből épül fel:

```
CU_TestInfo Tests_FrSM_RequestComMode[] = {
    {"Test_FrSM_RequestComModeSuccess",   Test_FrSM_RequestComModeSuccess},
    {"Test_FrSM_RequestComModeSilentCom", Test_FrSM_RequestComModeSilentCom},
    {"Test_FrSM_RequestComModeInvHandle", Test_FrSM_RequestComModeInvHandle},
    {"Test_FrSM_RequestComModeInvMode",   Test_FrSM_RequestComModeInvMode},
    CU_TEST_INFO_NULL
  };
```

A sikeres függvényhívás esetének vizsgálata során az összes konfigurációs paraméter szett összes cluster-ére meghívjuk a függvényt mindkét lehetséges érvényes paraméterével:

```
void Test_FrSM_RequestComModeSuccess(void)
{
  .
  .
  /* Verifikacios pont: visszateresi ertek E_OK */
  CU_ASSERT_EQUAL(
    FrSM_RequestComMode(
      FrSM_Config[ConfigSetIdx].FrSMCluster[ClstrIdx].FrSMComMNetworkHandleRef,
      COMM_FULL_COMMUNICATION),
    E_OK);
  /* Verifikacios pont: visszateresi ertek E_OK */
  CU_ASSERT_EQUAL(
  FrSM_RequestComMode(
  FrSM_Config[ConfigSetIdx].FrSMCluster[ClstrIdx].FrSMComMNetworkHandleRef,
  COMM_NO_COMMUNICATION),
  E_OK);

  /* Verifikacios pont: nem tortent hibajelzes */
#if (STD_ON == FRSM_DEV_ERROR_DETECT)
  CU_ASSERT_EQUAL(Det_ReportError_Registry.NumTimesApiInvoked, 0);
#endif
}
```

Ha a kért kommunikációs mód COMM_SILENT_COMMUNICATION, akkor a visszatérési értéket (E_NOT_OK) minden esetben, a hibajelzést bekapcsolt hibadetektálás esetén kell megvizsgálnunk:

```
void Test_FrSM_RequestComModeSilentCom(void)
{
  .
  .
  /* Verifikacios pont: visszateresi ertek E_NOT_OK */
  CU_ASSERT_EQUAL(
    FrSM_RequestComMode(
      FrSM_Config[ConfigSetIdx].FrSMCluster[ClstrIdx].FrSMComMNetworkHandleRef,
      COMM_SILENT_COMMUNICATION),
    E_NOT_OK);

#if (STD_ON == FRSM_DEV_ERROR_DETECT)
  /* Verifikacios pont: Det_ReportError meghivasainak szama */
  CU_ASSERT_EQUAL(Det_ReportError_Registry.NumTimesApiInvoked,1);
  .
  .
  /* Verifikacios pont: Det_ReportError ErrorId parametere */
  CU_ASSERT_EQUAL(Det_ReportError_Registry.ErrorId,FRSM_E_INV_MODE);
#endif
}
```

Az érvénytelen NetworkHandle teszt esetben olyan paraméterrel hívjuk meg a függvényt, amely nem egyenlő a cluster FrSMComMNetworkHandleRef paraméterével, a visszatérési értéket minden esetben, a hibajelzést csak bekapcsolt FrSMDevErrorDetect esetén kell megvizsgálnunk:

```
void Test_FrSM_RequestComModeInvHandle(void)
{
  .
  .
  /* Verifikacios pont: visszateresi ertek E_NOT_OK */
  CU_ASSERT_EQUAL(FrSM_RequestComMode(30U, COMM_FULL_COMMUNICATION), E_NOT_OK);

#if (STD_ON == FRSM_DEV_ERROR_DETECT)
  /* Verifikacios pont: Det_ReportError meghivasainak szama */
  CU_ASSERT_EQUAL(Det_ReportError_Registry.NumTimesApiInvoked,1);
  .
  .
  /* Verifikacios pont: Det_ReportError ErrorId parametere */
  CU_ASSERT_EQUAL(Det_ReportError_Registry.ErrorId,FRSM_E_INV_HANDLE);
#endif /* STD_ON == FRSM_DEV_ERROR_DETECT */
}
```

Érvénytelen kommunikációs módnak számít minden a FullCom és NoCom kérés kivételével, ilyenkor a visszatérési értéket minden esetben, a hibajelzést csak bekapcsolt FrSMDevErrorDetect esetén kell megvizsgálnunk:

```
void Test_FrSM_RequestComModeInvMode(void)
{
  .
  .
  /* Verifikacios pont: visszateresi ertek E_NOT_OK */
  CU_ASSERT_EQUAL(
    FrSM_RequestComMode(
      FrSM_Config[ConfigSetIdx].FrSMCluster[ClstrIdx].FrSMComMNetworkHandleRef,
      COMM_MODE_UNDEFINED),
    E_NOT_OK);
```

```
#if (STD_ON == FRSM_DEV_ERROR_DETECT)
  /* Verifikacios pont: Det_ReportError meghivasainak szama */
  CU_ASSERT_EQUAL(Det_ReportError_Registry.NumTimesApiInvoked,1);
  .
  .
  /* Verifikacios pont: Det_ReportError ErrorId parametere */
  CU_ASSERT_EQUAL(Det_ReportError_Registry.ErrorId,FRSM_E_INV_MODE);
#endif /* STD_ON == FRSM_DEV_ERROR_DETECT */
}
```

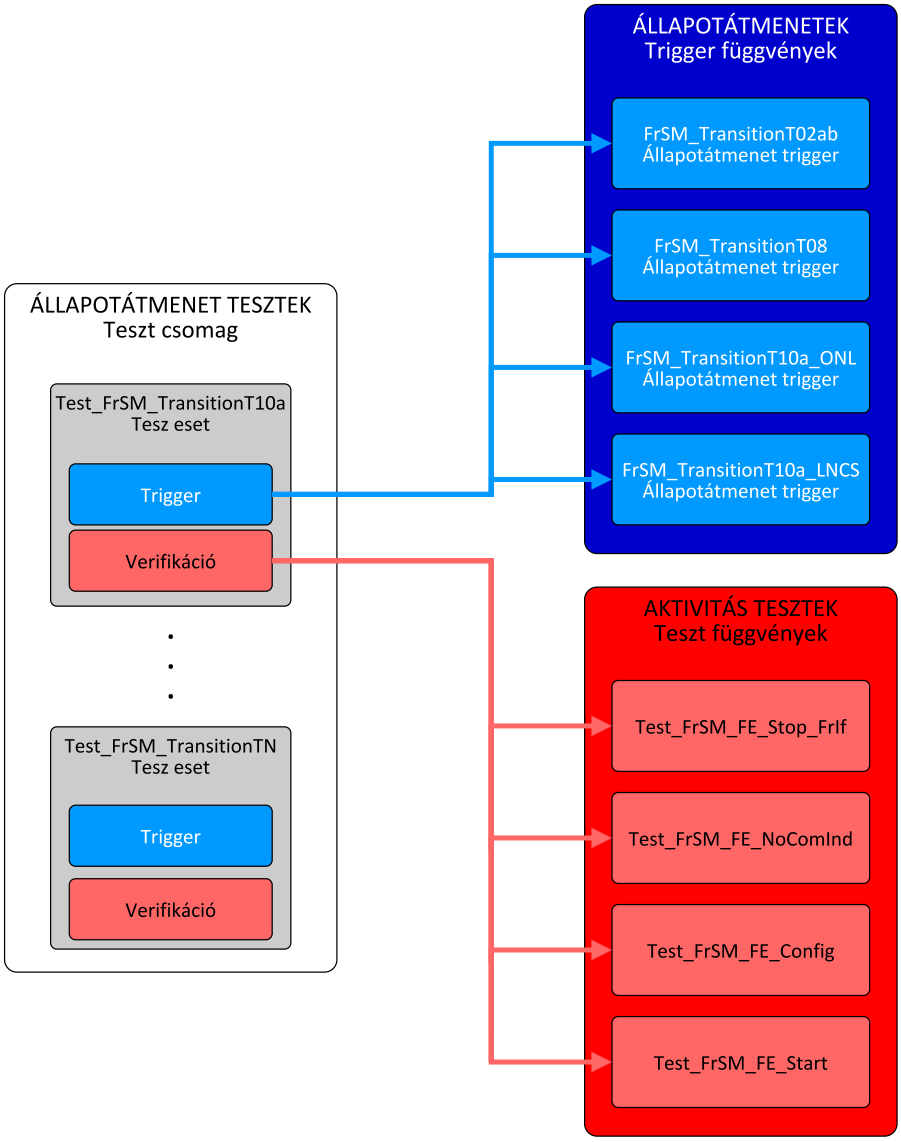
Ha az inicializátlan modul FrSM_RequestComMode függvényét egyébként helyes paraméterekkel hívjuk meg, E_NOT_OK-kal kell visszatérnie és bekapcsolt hibadetektálás esetén jeleznie kell a Det-nek:

```
void Test_FrSM_RequestComModeUnInit(void)
{
  .
  .
  /* Verifikacios pont: visszateresi ertek E_NOT_OK */
  CU_ASSERT_EQUAL(
    FrSM_RequestComMode(
      FrSM_Config[0].FrSMCluster[0].FrSMComMNetworkHandleRef,
      COMM_FULL_COMMUNICATION),
    E_NOT_OK);

#if (STD_ON == FRSM_DEV_ERROR_DETECT)
  /* Verifikacios pont: Det_ReportError meghivasainak szama */
  CU_ASSERT_EQUAL(Det_ReportError_Registry.NumTimesApiInvoked,1);
  .
  .
  /* Verifikacios pont: Det_ReportError ErrorId parametere */
  CU_ASSERT_EQUAL(Det_ReportError_Registry.ErrorId,FRSM_E_UNINIT);
#endif /* STD_ON == FRSM_DEV_ERROR_DETECT */
}
```

6.3.2. Állapotátmenetek teszt esetei

A lehető legnagyobb lefedettség elérése érdekében az állapotátmeneteket egyenként, az *összes* konfigurációs paraméterkészletre teszteltem. Ennek során az AUTOSAR CTS-ben megismert elvet követtem, azaz egy átmenet teszteléséhez először eljuttattam az állapotgépet a kiindulási állapotba, majd a megfelelő trigger feltételek biztosításával kiváltottam a tesztelni kívánt átmenetet (teszt eset trigger része, lásd 6.5. ábra). A teszt eset második, verifikációs részében teszteltem a specifikáció által előírt aktivitások végrehajtását.

Az aktivitások tesztelését a szabvány mintájára a nekik megfelelő függvényekbe partícionáltam, így egy egységbe zárva tesztelhetők, ezáltal a teszt esetek kódja egyszerűbb és áttekinthetőbb.



6.5. ábra. *Az állapotátmenet teszt esetek felépítése.*

Az állapotátmenetek tesztelésének koncepciójának bemutatásához a 6.3. fejezetben megismert T10(a) átmenetet választottam, mert ennek folyamata nem túlságosan bonyolult, azonban alkalmas egy összetett állapotból (FullCom) történő átmenet vizsgálatának bemutatására. A teszt eset elején a T02 → T08 trajektórián eljutunk FRSM_ONLINE állapotba, onnan pedig T10(a) során FRSM_STARTUP-ba. Ezután következnek az ellenőrzési és verifikációs pontok.

A T10(a) teszteléséhez a FullCom mindkét állapotából el kell jutnunk a célállapotba, ezért a teszt eset második felében a T02 → T08 → T40 trajektórián eljutunk FRSM_LOW_NUMBER_OF_COLDSTARTERS állapotba, majd ott előidézzük az átmenetet és elvégezzük a vizsgálatokat:

```
void Test_FrSM_TransitionT10a(void)
{
  .
  .
  /* Allapotatmenet: FRSM_READY -> FRSM_STARTUP */
  FrSM_TransitionT02ab(ConfigSetIdx, ClstrIdx);
  /* Allapotatmenet: FRSM_STARTUP -> FRSM_ONLINE */
  FrSM_TransitionT08(ConfigSetIdx, ClstrIdx);
  /* Allapotatmenet: FRSM_ONLINE -> FRSM_STARTUP */
  FrSM_TransitionT10a_ONL(ConfigSetIdx, ClstrIdx);

  /* Ellenorzesi pont: FrSMCheckWakeupReason == TRUE */
  if (TRUE==FrSM_Config[ConfigSetIdx].FrSMCluster[ClstrIdx].FrSMCheckWakeupReason)
  {
    /* Verifikacios pont: meghivasok szama */
    CU_ASSERT_EQUAL(Dem_ReportErrorStatus_Registry.NumTimesApiInvoked,1);
    /* Verifikacios pont: bemeno parameter EventId */
    CU_ASSERT_EQUAL(Dem_ReportErrorStatus_Registry.EventId,
      FrSMClusterDemEventParameterRefs[ClstrIdx].FrSM_E_ClusterSyncLoss);
    /* Verifikacios pont: bemeno parameter EventStatus */
    CU_ASSERT_EQUAL(Dem_ReportErrorStatus_Registry.EventStatus,
      DEM_EVENT_STATUS_FAILED);
    /* Verifikacios pont: meghivasok szama */
    CU_ASSERT_EQUAL(Det_ReportError_Registry.NumTimesApiInvoked, 1);
    /* Verifikacios pont: bemeno parameter ApiId */
    CU_ASSERT_EQUAL(Det_ReportError_Registry.ApiId,      FrSM_MainFunction_API_ID);
    /* Verifikacios pont: bemeno parameter ErrorId */
    CU_ASSERT_EQUAL(Det_ReportError_Registry.ErrorId,    FRSM_E_CLUSTER_SYNC_LOSS);
    /* Verifikacios pont: bemeno parameter InstanceId */
    CU_ASSERT_EQUAL(Det_ReportError_Registry.InstanceId, FRSM_INSTANCE_ID);
    /* Verifikacios pont: bemeno parameter ModuleId */
    CU_ASSERT_EQUAL(Det_ReportError_Registry.ModuleId,   FRSM_MODULE_ID);
    /* Aktivitasok tesztelse */
    Test_FrSM_FE_Stop_FrIf(ConfigSetIdx, ClstrIdx);
    Test_FrSM_FE_NoComInd(ConfigSetIdx, ClstrIdx);
    Test_FrSM_FE_Config(ConfigSetIdx, ClstrIdx);
    Test_FrSM_FE_Start(ConfigSetIdx, ClstrIdx);
  }
  .
  .
  /* Allapotatmenet: FRSM_READY -> FRSM_STARTUP */
  FrSM_TransitionT02ab(ConfigSetIdx, ClstrIdx);
  /* Allapotatmenet: FRSM_STARTUP -> FRSM_ONLINE */
  FrSM_TransitionT08(ConfigSetIdx, ClstrIdx);
  /* Allapotatmenet: FRSM_ONLINE -> FRSM_LOW_NUMBER_OF_COLDSTARTERS */
  FrSM_TransitionT40(ConfigSetIdx, ClstrIdx);
  /* Allapotatmenet: FRSM_LOW_NUMBER_OF_COLDSTARTERS -> FRSM_STARTUP */
  FrSM_TransitionT10a_LNCS(ConfigSetIdx, ClstrIdx);

  /* Ellenorzesi pont: FrSMCheckWakeupReason == TRUE es
   * NumOfStartupFrames < FrSMMinNumberOfColdstarter */
  if ((TRUE==FrSM_Config[ConfigSetIdx].FrSMCluster[ClstrIdx].FrSMCheckWakeupReason)
    &&(DummyNumOfStartupFrames <
        FrSM_Config[ConfigSetIdx].FrSMCluster[ClstrIdx].FrSMMinNumberOfColdstarter))
  {
    /* Verifikacios pontok */
  }
}
```

FRSM_ONLINE állapotból az alábbi trigger függvény segítségével idézhetjük elő az állapotátmenetet (lásd [6]). A trigger függvények is tartalmaznak ellenőrzési es verifikációs pontokat, annak érdekében, hogy nyomonkövethető legyen az állapotok változása az állapottérkép bejárása során:

```
void FrSM_TransitionT10a_ONL(unsigned int ConfigSetIdx, unsigned int ClstrIdx)
{
  /* Globalis leirok inicializalasa */
  init_Registries();

  /* Az allapotatmenetnek megfelelo dummy parameterek beallitasa */
  DummyNumOfStartupFrames = 4U;
  DummyPOCStatus.State    = FR_POCSTATE_HALT;
  DummyPOCStatus.Freeze   = FALSE;

  /* Verifikacios pont: visszateresi ertek E_OK */
  CU_ASSERT_EQUAL(
    FrSM_RequestComMode(
      FrSM_Config[ConfigSetIdx].FrSMCluster[ClstrIdx].FrSMComMNetworkHandleRef,
    COMM_FULL_COMMUNICATION),
    E_OK);

  /* Allapotatmenet eloidezese */
  FrSM_MainFunction(ClstrIdx);

  /* Ellenorzesi pont: FrSMCheckWakeupReason == TRUE */
  if (TRUE==FrSM_Config[ConfigSetIdx].FrSMCluster[ClstrIdx].FrSMCheckWakeupReason)
  {
    /* Verifikacios pont: meghivasok szama */
    CU_ASSERT_EQUAL(BswM_FrSM_CurrentState_Registry.NumTimesApiInvoked,1);
    /* Verifikacios pont: bemeno parameter Network */
    CU_ASSERT_EQUAL(BswM_FrSM_CurrentState_Registry.Network,
      FrSM_Config[ConfigSetIdx].FrSMCluster[ClstrIdx].FrSMComMNetworkHandleRef);
    /* Verifikacios pont: bemeno parameter CurrentState */
    CU_ASSERT_EQUAL(BswM_FrSM_CurrentState_Registry.CurrentState,
      FRSM_BSWM_STARTUP);
  }
}
```

Az aktivitások tesztelését megvalósító függvényre egy példa az FE_STOP_FRIF végrehajtását ellenőrző függvény, amelyben a verifikációs pontok az FrIf_SetState meghívásának számát és paramétereit (cluster index, állapotátmenet) vizsgálják:

```
void Test_FrSM_FE_Stop_FrIf(unsigned int ConfigSetIdx, unsigned int ClstrIdx)
{
  /* Verifikacios pont: meghivasok szama */
  CU_ASSERT_EQUAL(FrIf_SetState_Registry.NumTimesApiInvoked,1);
  /* Verifikacios pont: bemeno parameter ClstIdx */
  CU_ASSERT_EQUAL(FrIf_SetState_Registry.ClstIdx[0],
    FrSM_Config[ConfigSetIdx].FrSMCluster[ClstrIdx].FrSMFrIfClusterRef);
  /* Verifikacios pont: bemeno parameter StateTransition */
  CU_ASSERT_EQUAL(FrIf_SetState_Registry.StateTransition[0],
    FRIF_GOTO_OFFLINE);
}
```

7. fejezet

Megfelelőségi tesztelés

A szabvány alapján összeállított megfelelőségi tesztek eredményei alapvető fontosságú információkat hordoznak a modul „feketedoboz” viselkedéséről és arról hogy várhatóan illeszkedni fog-e a szoftverkörnyezetbe, amiben majd működnie kell.

A különböző kódfedettségi metrikák mellett fontos szempont a tesztelés jellemzésére a követelmény lefedettség. Egy AUTOSAR BSW modul esetén a szabvány által specifikált követelmények alapvetően két csoportra oszthatók: statikus és dinamikus követelményekre (lásd 1.1. fejezet). A futtatható megfelelőségi teszt esetek értelemszerűen csak dinamikus követelmények ellenőrzésére alkalmasak, a statikus követelmények teljesülése a code review alkalmával vizsgálható.

Ennek megfelelően a követelmény lefedettség számításánál csak a viselkedési követelményeket vettem figyelembe, azok közül is kihagytam azokat, amelyeket a szabvány a „Non-observable Module Behavior” kategóriába sorol.

7.1. AUTOSAR teszt csomag futtatása

A konzorcium által kibocsátott TTCN-3 teszt csomagot OpenTTCN Tester és TTworkbench Professional segítségével is lefuttattam a FlexRay State Manager modulra, az 5. fejezetben bemutatott illesztőkód mindkét szoftver esetén használható volt.

A teszt csomag egyes teszt eseteinek számát a konfiguráció függvényében a 7.1. táblázat tartalmazza. Ebből a statisztikából is látszik az AUTOSAR CTS egyik komoly hiányossága: az API függvények egyedi tesztelésére nem helyez elég nagy hangsúlyt.

7.1. táblázat. *Az AUTOSAR CTS összetétele.*

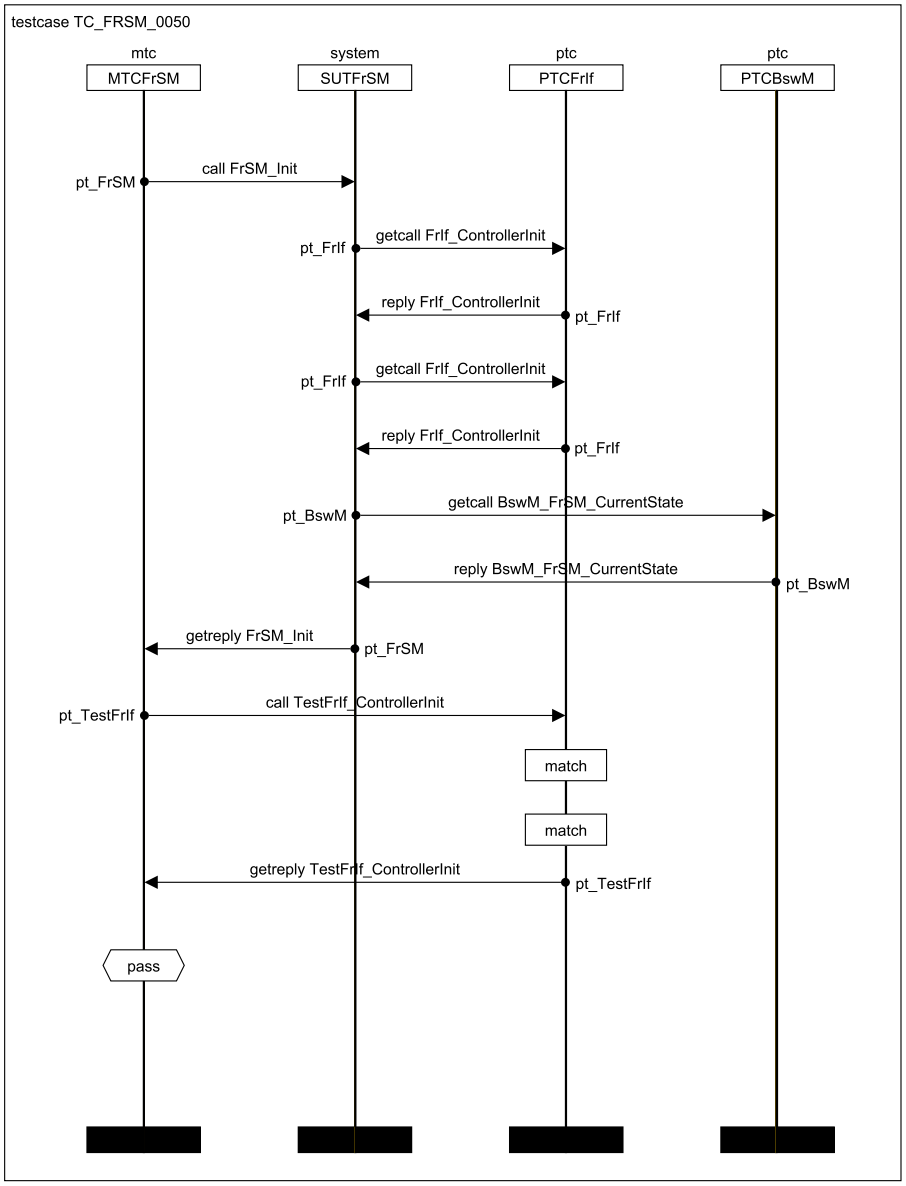
	VersionInfo OFF	**VersionInfo ON**
FrSM_Init	1	
Állapotátmenet	44	
FrSM_GetVersionInfo	0	1
Összesen	45	46

Amint arról a 6.3. fejezetben szó volt, van néhány nem elhanyagolható különbség a 2-es és 3-as revízió követelményei között, amelyeket a tesztelés során figyelembe kell venni. Például az AUTOSAR teszt csomagban az FrSM_Init teszt esete nem tartalmaz verifikációs pontot annak ellenőrzésére, hogy a modul jelezte-e az FRSM_READY állapotba történő belépést a BswM_FrSM_CurrentState függvény meghívásával. A szabvány 3-as revíziója ezzel szemben az [FrSm145] követelményben előírja, hogy a modul *minden* állapotátmenetet köteles jelezni a BSW Mode Manager-nek.

Az ehhez hasonló különbségek azonban a teszetelés során fennakadást nem okoznak, hiszen egy teszt eset nem lesz sikertelen azért, mert a modul a specifikált viselkedésen felül végrehajt valamilyen függvényhívást. Jelentős eltérés azonban, hogy a 2-es revízióban az állapottérképnek eggyel kevesebb állapota van: ebben ugyanis még nem szerepel az FRSM_LOW_NUMBER_OF_COLDSTARTERS állapot, és ennek megfelelően az FrSMMinNumberOfColdstarter konfigurációs paraméter sem. Ez tehát egy olyan különbség, ami a követelmény- és utasítás lefedettséget is negatívan befolyásolja.

Az ehhez hasonló kisebb-nagyobb inkonzisztenciák miatt nem hagyatkozhatunk a tesztelés során kizárólag erre a CTS-re – a későbbi revíziók esetén pedig valószínűleg még ennyire sem. A 7.1. ábrán látható az FrSM_Init teszt eset (TC_FRSM_0050) futtatásakor létrejövő szekvenciadiagram.

Amint az a szekvenciadiagramon is látható, a BswM_FrSM_CurrentState meghívását nem vizsgálja a teszt eset, csak az FrIf_ControllerInit hívásainak számát és paramétereit. A futtatókörnyezet minden teszt esethez generál egy ehhez hasonló szekvenciadiagramot, ezáltal jól követhetővé téve a teszt végrehajtásának folyamatát.



7.1. ábra. *A TC_FRSM_0050 teszt eset futásának szekvenciadiagramja.*

Sajnos a különböző kódfedettségi metrikák számításához nem nyújt támogatást a TTCN-3 teszt csomag. A specifikációja alapján a 2-es revízióra vonatkozó követelmény lefedettsége (a statikus és dinamikus analízis elvégzése esetén) 100%. A 3-as verzió követelményei közül a fent említett eltérések miatt lényegesen kevesebbet, a viselkedési követelmények 90,5%-át fedi le.

7.2. CUnit tesztek futtatása

A 6.3. fejezetben bemutatott teszt csomag egy makefile segítségével kényelmesen futtatható parancssorból egy alkalmas C fordító (pl. GCC) használatával. A teszt csomag egyes teszt eseteinek számát a konfiguráció függvényében a 7.2. táblázat tartalmazza.

7.2. táblázat. *A CUnit teszt csomag összetétele.*

	VersionInfo OFF	**VersionInfo ON**
Uninit teszt esetek	5	
FrSM_Init	2	
FrSM_SetEcuPassive	1	
Állapotátmenet	41	
FrSM_RequestComMode	4	
FrSM_GetCurrentComMode	3	
FrSM_AllSlots	2	
FrSM_GetVersionInfo	0	1
Összesen	58	59

Látható, hogy a teszt esetek összetétele ebben az esetben is csak az FrSMVersionInfoApi paramétertől függ. A teszt csomag futtatásának eredménye a Basic interfész verbose módban történő használata esetén a 7.2. ábrán látható. Amint arról az 59 lefutott teszt eset is árulkodik, ez az eredmény mindkét konfigurációs paraméter bekapcsolt állapota mellett született.

A kódfedettségi metrikák meghatározásához valamilyen profiler eszköz használata javasolt. Ilyen pl. a Gcov, amely része a GCC eszközláncnak és a `gcov` paranccsal futtatható. A Gcov bemenetéül szolgáló .gcno és .gcda fájlokat a futó alkalmazás generálja, az ehhez szükséges kiegészítéseket a GCC és a linker az `-fprofile-arcs` és `-ftest-coverage` opciók hatására helyezi el a kódban.

```
Suite: Suite_FrSM_UnInitTests
  Test: Test_FrSM_RequestComModeUnInit ...passed
  Test: Test_FrSM_GetCurrentComModeUnInit ...passed
  Test: Test_FrSM_AllSlotsUnInit ...passed
  Test: Test_FrSM_SetEcuPassiveUnInit ...passed
  Test: Test_FrSM_MainFunctionUnInit ...passed
Suite: Suite_FrSM_Init
  Test: Test_FrSM_InitSuccess ...passed
  Test: Test_FrSM_InitNullPointer ...passed
Suite: Suite_FrSM_RequestComMode
  Test: Test_FrSM_RequestComModeSuccess ...passed
  Test: Test_FrSM_RequestComModeSilentCom ...passed
  Test: Test_FrSM_RequestComModeInvHandle ...passed
  Test: Test_FrSM_RequestComModeInvMode ...passed
Suite: Suite_FrSM_GetCurrentComMode
  Test: Test_FrSM_GetCurrentComModeSuccess ...passed
  Test: Test_FrSM_GetCurrentComModeInvHandle ...passed
  Test: Test_FrSM_GetCurrentComModeNullPtr ...passed
Suite: Suite_FrSM_GetVersionInfo
  Test: Test_FrSM_GetVersionInfoSuccess ...passed
Suite: Suite_FrSM_AllSlots
  Test: Test_FrSM_AllSlotsSuccess ...passed
  Test: Test_FrSM_AllSlotsInvHandle ...passed
Suite: Suite_FrSM_SetEcuPassive
  Test: Test_FrSM_SetEcuPassiveSuccess ...passed
Suite: Suite_FrSM_Transitions
  Test: Test_FrSM_TransitionT00 ...passed
  Test: Test_FrSM_TransitionT01ab ...passed
  Test: Test_FrSM_TransitionT01c ...passed
  Test: Test_FrSM_TransitionT02ab ...passed
  Test: Test_FrSM_TransitionT03ab ...passed
  Test: Test_FrSM_TransitionT03cd ...passed
  Test: Test_FrSM_TransitionT03e ...passed
  Test: Test_FrSM_TransitionT13 ...passed
  Test: Test_FrSM_TransitionT30_FRSM_WAKEUP ...passed
  Test: Test_FrSM_TransitionT31 ...passed
  Test: Test_FrSM_TransitionT103 ...passed
  Test: Test_FrSM_TransitionT04a ...passed
  Test: Test_FrSM_TransitionT04b ...passed
  Test: Test_FrSM_TransitionT05 ...passed
  Test: Test_FrSM_TransitionT06 ...passed
  Test: Test_FrSM_TransitionT08 ...passed
  Test: Test_FrSM_TransitionT12 ...passed
  Test: Test_FrSM_TransitionT30_FRSM_STARTUP ...passed
  Test: Test_FrSM_TransitionT32 ...passed
  Test: Test_FrSM_TransitionT108 ...passed
  Test: Test_FrSM_TransitionT40 ...passed
  Test: Test_FrSM_TransitionT41 ...passed
  Test: Test_FrSM_TransitionT09a ...passed
  Test: Test_FrSM_TransitionT10a ...passed
  Test: Test_FrSM_TransitionT16a ...passed
  Test: Test_FrSM_TransitionT20a ...passed
  Test: Test_FrSM_TransitionT09b ...passed
  Test: Test_FrSM_TransitionT10b ...passed
  Test: Test_FrSM_TransitionT16b ...passed
  Test: Test_FrSM_TransitionT20b ...passed
  Test: Test_FrSM_TransitionT30_FRSM_KEYSLOT_ONLY ...passed
  Test: Test_FrSM_TransitionT101 ...passed
  Test: Test_FrSM_TransitionT14 ...passed
  Test: Test_FrSM_TransitionT15 ...passed
  Test: Test_FrSM_TransitionT17 ...passed
  Test: Test_FrSM_TransitionT20c ...passed
  Test: Test_FrSM_TransitionT30_FRSM_ONLINE_PASSIVE ...passed
  Test: Test_FrSM_TransitionT33 ...passed
  Test: Test_FrSM_TransitionT115 ...passed
  Test: Test_FrSM_TransitionT11 ...passed
  Test: Test_FrSM_TransitionT30_FRSM_HALT_REQ ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      8      8    n/a      0        0
               tests     59     59     59      0        0
             asserts   6482   6482   6482      0      n/a

Elapsed time =    0.078 seconds
```

7.2. ábra. *A Cunit teszt csomag futásának eredménye.*

A Cunit teszt csomaggal elért lefedettségi adatok az alábbiak:

- **Követelmény lefedettség**: 97,44%.
- **Utasítás lefedettség**: 100%.
- **Döntési feltétel lefedettség**: 99,48%.

A Gcov által számított utasítás lefedettség ugyan nem éri el a 100%-ot, ez azonban annak tudható be, hogy a kódban a MISRA-C előírásai miatt vannak olyan default és else ágak, amelyek a működés során semmi esetben sem futhatnak le. Az ezek figyelmenkívül hagyásával kalkulált valódi kódfedettség 100%.

7.3. A TTCN-3 és CUnit tesztek összehasonlítása

A FlexRay State Manager modul tesztelése során szerzett tapasztalatok az alábbi összehasonlítás formájában foglalhatók össze:

	AUTOSAR CTS	**Tesztelés CUnit segítségével**
Előnyök	A szabványalkotó testület által specifikált teszt esetek, használatukkal elkerülhető, hogy értelmezési tévedések miatt rosszul teszteljünk.	Ingyenes eszköz, amelynek használatához nem szükséges új szoftver beszerzése vagy egy tesztleíró nyelv elsajátítása: olcsó és egyszerű.
	Segítségével nemcsak az egyes függvényhívások száma és paraméterei tesztelhetők, hanem azok sorrendje is, amelyet a szabvány sok esetben követelményként rögzít.	A Gcov révén lehetővé teszi kódfedettségi metrikák számítását, amelyek bizonyos esetekben a követelmény lefedettségnél alkalmasabbak kvantitatív jellemzésre.
	A tesztek során generált szekvenciadiagramoknak köszönhetően lehetőség van a teszt futásának pontos követésére, ez megkönnyíti a hibakeresést.	Nincs szükség sem adatkonverzióra sem a futtatókörnyezet illesztésére, nem kell tehát speciális glue code-ot írni a C nyelven implementált SUT-hoz.
Hátrányok	Használatához szükség van egy új nyelv legalább alap szintű elsajátítására, valamint egy adott esetben igen drága szoftver beszerzésére.	CUnit-tal történő tesztelés esetén a teszt eseteknek sem a specifikációja, sem az implementációja nem adott, azokat a fejlesztőknek kell megvalósítani.
	Ahhoz, hogy a C nyelven implementált SUT-ra lehessen futtatni a teszt eseteket, komplexebb modul esetén igen terjedelmes illesztő kód szükséges.	Közvetlenül nem támogatja a függvényhívások sorrendjének tesztelését, ami követelmény lefedettség és viselkedési tesztelés szempontjából fontos lenne.
	Nem nyújt támogatást kódfedettségi metrikák számításához, az egyetlen kvantitatív adat, amire hagyatkozhatunk, a teszt specifikációban publikált követelmény lefedettség.	Az eredmények vizuális megjelenítésére és kiértékelésére semmilyen lehetőséget nem biztosít, csupán arról szolgáltat információt, hogy egy teszt eset megfelelt-e vagy sem.

Mindkét módszerről elmondható, hogy külön-külön is alkalmasak egy AUTOSAR BSW modul megfelelőségi tesztelésének elvégzésére, azonban igazán meggyőző eredményt a kettő *kombinációjával* lehetne elérni. Hiszen az AUTOSAR által specifikált viselkedési teszt esetek sok esetben iránymutatók lehetnek tervezési kérdésekben, a CUnit segítségével pedig számíthatók és javíthatók a kódfedettségi mutatók, amelyek önmagukban is alkalmasak egy teszt csomag minőségének jellemzésére.

A CUnit legnagyobb előnye egyszerűsége, valamint az, hogy ingyenes eszköz, ami telepítés után gyakorlatilag azonnal használható és jól dokumentált. Ez kárpótol az általa nyújtott kevesebb szolgáltatásért. A TTCN-3, mint tesztelési módszer amiatt tűnt vonzónak, hogy az AUTOSAR konzorcium ezen a nyelven bocsátotta ki a hivatalos megfelelőségi teszt csomagját. Ennek hiányában – a ráfordítandó pénz és erőforrás miatt – kevésbé csábító alternatíva, azonban ettől függetlenül egy feketedoboz tesztelésre kiválóan alkalmas leíró nyelv.

8. fejezet

Összefoglaló

A diplomaterv az AUTOSAR FlexRay State Manager modul megfelelőségi tesztelését a tesztelési alapok, az FrSM modul, a TTCN-3 nyelv és a CUnit keretrendszer bemutatásán keresztül ismerteti. Összegzésképp elmondható, hogy a feladat megoldása során értékes tapasztalatokat szereztem a TTCN-3 nyelvvel, a TTCN-alapú eszközökkel, a tesztelési módszertanokkal, valamint a teszt esetek tervezésével kapcsolatban. A modul megvalósításának minősége nagymértékben javult a teszteknek köszönhetően, azok során számos értelmezésbeli és implementációs hibára derült fény.

8.1. Értékelés

A dolgozat alapjául szolgáló tervezési és fejlesztési munka annak tükrében értékelhető, hogy a diplomaterv kiírásában megfogalmazott feladatok elkészültek-e, illetve a velük kapcsolatban kitűzött célok megvalósultak-e:

- A dolgozatban hivatkozott források és a FlexRay State Manager modul AUTOSAR teszt specifikációja segítségével elsajátítottam a megfelelőségi tesztelés alapjait és a TTCN-3 nyelv használatát.
- A rendelkezésemre álló információk alapján kiválasztottam két TTCN-alapú eszközt, amelyek használhatóság, dokumentáltság és elérhetőség szempontjából kiemelkedtek a többi közül.
- Megvizsgáltam hogyan illeszthető a C nyelven implementált modul a TTCN-3 futtatókörnyezethez, megterveztem az illesztés mechanizmusát, majd implementáltam a glue code-ot.
- Az AUTOSAR által specifikált teszt esetek alapján megterveztem a CUnit-alapú modultesztek struktúráját, a lehető legjobb utasítás- és döntési feltétel lefedettségre törekedve.

- Kialakítottam a modul számára egy tesztkörnyezetet, amely lehetővé teszi a feketedoboz viselkedés ellenőrzését.
- Mindkét eszközkészlettel elvégeztem a modul megfelelőségi tesztelését, megvizsgáltam, hogy a 2-es revízióhoz kiadott CTS mekkora részét fedi le a 3-as revízió követelményeinek és hogy mekkora javulást jelent ehhez képest a CUnit-alapú teszt csomag.
- Az általam specifikált teszt esetek segítségével sikerült elérni a kitűzött kódfedettségi mérőszámokat, valamint javítani a követelmény lefedettséget az AUTOSAR CTS-hez képest.

A megfelelőségi tesztek eredményeiből levonható a következtetés, miszerint – bár általában kisebb kódfedettséget biztosít – nagyon hasznos egy harmadik fél által specifikált teszt csomag, mert sok értelmezésbeli kérdésben támpontot nyújthat. Ennek ellenére nem szabad alábecsülni a saját magunk által specifikált és implementált tesztek jelentőségét.

A TTCN-3 használata során kiderült, hogy viselkedési tesztelés terén számos előnye van egy hagyományos programozási nyelvhez képest, segítségével nagyban leegyszerűsíthető a teszt esetek leírása. Hatékony alkalmazásának feltétele, hogy az illesztőkódot valamilyen sablon alapján generálni lehessen a különböző modulokhoz. Ennek hiányában nem biztos, hogy megéri a plusz ráfordítást.

8.2. Fejlesztési lehetőségek

Mindkét teszt eszköz esetén felmerül a továbbfejlesztés lehetősége. A TTCN-3 esetében az egyik ilyen lehetőség az illesztőkód generálása, ami nagymértékben megnövelné a tesztelés egyszerűségét, a másik a teszt esetek kiegészítése oly módon, hogy azok lefedjék a 3-as revízió követelményeit is. Mindkettő megvalósítható, hiszen az illesztőkód nagy része generálható egy sablon alapján, ha ismerjük a SUT interfészeit és saját típusait, valamint a használni kívánt adattípusokat.

CUnit esetén felmerül az igény arra, hogy a SUT által kezdeményezett függvényhívások sorrendjét is ellenőrizni tudjuk, tehát a tesztkörnyezet ebben a tekintetben fejleszthető. A TTCN-hez hasonlóan itt is felmerülhet a függvénycsonkok generálásának igénye, ami egy összetett modul esetén sok munkától kímélné meg a tesztelőt, ez ebben az esetben is viszonylag egyszerűen megvalósítható.

Irodalomjegyzék

[1] AUTomotive Open System ARchitecture. AUTOSAR basics. http://autosar.org, October 2012.

[2] HTB. *Szoftvertesztelés egységesített kifejezéseinek gyűjteménye*. Magyar Szoftvertesztelői Tanács Egyesület, February 2012. http://www.hstqb.com/images/b/b3/HTB-Glossary-3_12.pdf.

[3] Alain Gilberg. AUTOSAR conformance testing using TTCN-3. In *TTCN-3 User Conference*, pages 23–26, Sophia Antipolis, France, 2009.

[4] European Telecommunications Standards Institute. ETSI Conformance Testing. http://www.etsi.org/WebSite/technologies/ConformanceTesting.aspx, October 2012.

[5] AUTOSAR Consortium. *Layered Software Architecture*. Automotive Open System Architecture, October 2011. http://autosar.org/download/R4.0/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf.

[6] AUTOSAR Consortium. *Specification of FlexRay State Manager v2.2.0*. Automotive Open System Architecture, December 2011. http://autosar.org/download/R4.0/AUTOSAR_SWS_FlexRayStateManager.pdf.

[7] ETSI Standard. *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*. European Telecommunications Standards Institute, April 2012. ETSI ES 201 873-1 V4.4.1.

[8] European Telecommunications Standards Institute. ETSI's official TTCN-3 home page. http://www.ttcn-3.org, November 2012.

[9] Testing Technologies. *TTworkbench User's Guide*. Testing Technologies IST GmbH, April 2012.

[10] OpenTTCN. *OpenTTCN Tester 2012 User Guide*. OpenTTCN Ltd., July 2012.

[11] ETSI Standard. *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)*. European Telecommunications Standards Institute, April 2012. ETSI ES 201 873-5 V4.4.1.

[12] ETSI Standard. *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI)*. European Telecommunications Standards Institute, April 2012. ETSI ES 201 873-6 V4.4.1.

[13] A Unit Testing Framework for C. CUnit home page. `http://cunit.sourceforge.net/`, November 2012.

[14] AUTOSAR Consortium. *Specification of FlexRay Interface v3.3.0*. Automotive Open System Architecture, December 2011. `http://autosar.org/download/R4.0/AUTOSAR_SWS_FlexRayInterface.pdf`.

[15] AUTOSAR Consortium. *Conformance Test Specification of FlexRay State Manager v1.0.0*. Automotive Open System Architecture, March 2011. `http://autosar.org/download/R4.0/AUTOSAR_CTSP_FlexRayStateManager.pdf`.

[16] AUTOSAR Consortium. *Conformance Test Process Definition Path D*. Automotive Open System Architecture, April 2011. `http://autosar.org/download/R4.0/AUTOSAR_PD_CTProcessDefinitionPathD.pdf`.

[17] AUTOSAR Consortium. *Conformance Test Process Definition Path A-C*. Automotive Open System Architecture, April 2011. `http://autosar.org/download/R4.0/AUTOSAR_PD_CTProcessDefinitionPathAToC.pdf`.

Függelék

F.1. Lehetőségek a megfelelőség tanúsítására

Az AUTOSAR konzorcium négy különböző módszert biztosít partnereinek a megfelelőség igazolására (F.1.1. ábra). A sikeres megfelelőségi tanúsítvány minden esetben igazolja, hogy a tesztelt termék mind működés, mind újrafelhasználhatóság és hordozhatóság, mind skálázhatóság és konfigurálhatóság szempontjából megfelel a releváns AUTOSAR szabványnak.

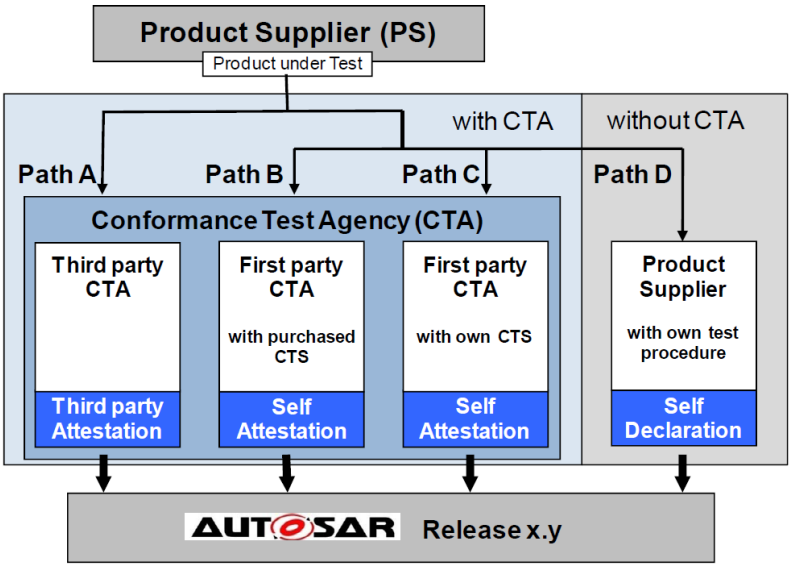
A megfelelőségi tesztcsomagok az AUTOSAR szoftver alábbi komponenseit fedik le:

- Basic Software (BSW)
- Runtime Environment (RTE)
- Alkalmazás réteg függvényei (SW-C)
- Eszközök (konfiguráció stb.)

Fontos megjegyezni, hogy a megfelelőségi tesztcsomagok nem szolgálnak a szabványok közötti konzisztencia ellenőrzésére, azt előfeltételként kezelik. A hivatalos CTS specifikációkat minden esetben az AUTOSAR konzorcium adja ki, azok a szabvány részét képezik.

Az F.1.1. ábrán látható A, B és C módszer közös jellemzője, hogy míg a D módszer esetén a tesztek implementálását, a tesztelést és a megfelelőségi nyilatkozat kibocsátását is a terméket fejlesztő cég végzi, addig előbbi esetekben a tesztcsomag implementálása – esetleg a tesztek végrehajtása – és a megfelelőség tanúsítása egy CTA[1] feladata.

[1]Conformance Test Agency - AUTOSAR megfelelőségi tesztek elvégzésére szakosodott cég, amely rendelkezik erre vonatkozó minősítéssel

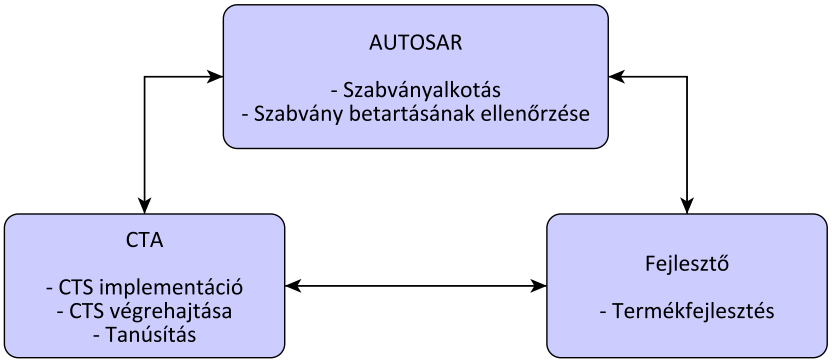


F.1.1. ábra. *Beszállítók lehetőségei a megfelelőség tanúsítására [16].*

A módszerek közötti szignifikáns különbségek az alábbiak [17][16]:

A módszer

Ebben az esetben az F.1.2. ábrán látható feladatmegosztás érvényes, vagyis a CTA egy harmadik fél, aki kizárólag a megfelelőségi tesztelésért felel, azért viszont teljes mértékben.

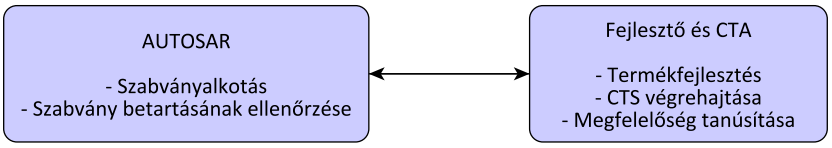


F.1.2. ábra. *A megfelelőség tanúsítása az A módszer szerint.*

Ekkor a fejlesztő biztosítja a tesztelendő terméket, a CTA végzi a tesztcsomag implementálását, végrehajtását, jelentést készít a teszt eredményekről, valamint sikeres lefutás esetén kiállítja a megfelelőséget igazoló dokumentumot.

B és C módszer

Ebben az esetben az F.1.3. ábrán látható feladatmegosztás érvényes, ekkor a CTA nem egy harmadik fél, ezt a szerepet betöltheti a fejlesztő cég is.

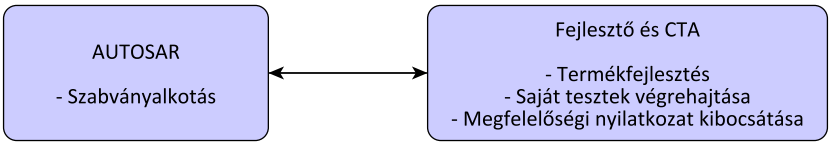


F.1.3. ábra. *A megfelelőség tanúsítása a B és C módszer szerint.*

A két módszer közötti különbség az, hogy amíg a B esetben a tesztcsomag (CTS) származhat harmadik féltől, addig a C módszer esetén a CTS-t is a fejlesztő kell, hogy implementálja. A C esetben is ragaszkodni kell azonban a megfelelőségi tesztcsomagra vonatkozó AUTOSAR szabványhoz.

D módszer

Ebben az esetben a tesztelési folyamat minden lépését a fejlesztő végzi, valamint ő felelős a tesztcsomag specifikálásért és implementálásáért is (F.1.4. ábra).



F.1.4. ábra. *A megfelelőség tanúsítása a D módszer szerint.*

A fejlesztő a saját maga által definiált teszt eseteket hajtja végre a terméken. Sikeres lefutásuk esetén megfelelőségi nyilatkozatot tesz, amellyel igazolja, hogy a termék megfelel a releváns AUTOSAR szabványnak.

F.2. TTCN-3 Runtime Interface adattípusai

TRI Absztrakt Típus	ANSI C Reprezentáció

BinaryString	`typedef struct BinaryString {`
	`  unsigned char* data;`
	`  long int bits;`
	`  void* aux;`
	`} BinaryString;`

QualifiedName	`typedef struct QualifiedName {`
	`  char* moduleName;`
	`  char* objectName;`
	`  void* aux;`
	`} QualifiedName;`

TriAddress	`typedef BinaryString TriAddress;`

TriAddressList	`typedef struct TriAddressList {`
	`  /* Cimlista */`
	`  TriAddress** addrList;`
	`  /* Elemszam */`
	`  long int length;`
	`} TriAddressList;`

TriComponentId	`typedef struct TriComponentId {`
	`  BinaryString compInst;`
	`  String compName;`
	`  QualifiedName compType;`
	`} TriComponentId;`

TriComponentIdList	`typedef struct TriComponentIdList {`
	`  TriComponentId** compIdList;`
	`  long int length;`
	`} TriComponentIdList;`

TriParameter	`typedef struct TriParameter {`
	`  BinaryString par;`
	`  TriParameterPassingMode mode;`
	`} TriParameter;`

TriParameterList	`typedef struct TriParameterList {`
	`  TriParameter** parList;`
	`  long int length;`
	`} TriParameterList;`

TriParameterPassingMode	`typedef enum {`
	`  TRI_IN = 0,`
	`  TRI_INOUT = 1,`
	`  TRI_OUT = 2`
	`} TriParameterPassingMode;`

TriPortId	`typedef struct TriPortId {`
	`  TriComponentId compInst;`
	`  char* portName;`
	`  long int portIndex;`
	`  QualifiedName portType;`
	`  void* aux;`
	`} TriPortId;`

TriPortIdList	`typedef struct TriPortIdList {`
	`  TriPortId** portIdList;`
	`  long int length;`
	`} TriPortIdList;`