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

Vajay Levente

AUTOSAR CAN TRANSPORT LAYER TESZTELÉSE

EGYETEMI KONZULENS:
Dr. Sujbert László

KÜLSŐ KONZULENS:
Szikszay László
thyssenkrupp Components Technology Hungary Kft.

BUDAPEST, 2023

TARTALOMJEGYZÉK

HALLGATÓI NYILATKOZAT

Alulírott **Vajay Levente**, 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 hitelesített felhasználók számára) közzé tegye. 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.

Kelt: Budapest, 2023. 06. 10.

...…….

Vajay Levente

Abstract

During my diploma work, I familiarized myself with the CAN-FD and FlexRay communication protocols. I learned about the structure of messages, transmission methods, and their application areas. FlexRay is faster and more redundant, but its implementation is also more expensive. CAN-FD evolved from the CAN protocol, focusing on compatibility and cost, making it more widespread in not only the automotive industry but also in other sectors. The CanTp module, which is the topic of my thesis, belongs to the software architecture of the AUTOSAR standard family, specifically within the communication stack. This document presents the architecture's structure and provides a general description and operation of the modules within the communication stack. Before discussing the testing of CanTp, let's overview a software life cycle model in general, where testing plays a significant role. The document covers black-box and white-box testing techniques, as well as their roles and importance. Following that, I present the software testing related to the CanTp module, specifically providing an overview of the requirements, establishing the test environment, and implementing individual test suites and cases. All created test cases pass successfully, and the branch coverage value of the module reaches 100%, meaning that my tests fully cover the module. As part of my diploma work, I performed the generator testing for the associated module, ensuring the provision of informative messages in case of erroneous configuration inputs, and verifying that the generator does not provide faulty configuration files in such cases. Finally, I conducted the integrated testing of CanTp, where I familiarized myself with the entire CAN stack, its modules, and created the appropriate configuration for its operation. Using the interfaces of the Dcm module, I created I-PDUs and initiated message transmission on the CAN network using these I-PDUs. By connecting the two CAN channels of the hardware on the same ECU, I implemented message sending, receiving, and error injection, thereby confirming the correct functioning of CanTp. Based on software and integration tests, the CanTp module operates according to the requirements.

Összefoglaló

A diplomamunkám során megismerkedtem a CAN-FD és FlexRay kommunikációs protokollokkal. Megismertem az üzenetek felépítését, a küldési metódusokat és az alkalmazási területeiket. A FlexRay gyorsabb, redundánsabb, viszont alkalmazása költségesebb is. A CAN-FD a CAN protokollból fejlődött ki, a kompatibilitást és a költségeket szem előtt tartva, így ennek a protokollnak az alkalmazása elterjedtebb, nem csak az autóiparban, de más iparágakban is. A CanTp modul, amely a dolgozat témáját is adja, az AUTOSAR szabványcsaládhoz tartozó szoftverarchitektúrába tartozik, ezen belül is a kommunikációs stack-hez. Ez a dokumentum ismerteti az architektúra felépítését és a kommunikációs stack-ben lévő modulok általános leírását és működését. A CanTp tesztelése előtt tárgyaljuk általánosságban egy szoftver életciklus modelljét, amelyből a tesztelés külön kiemelt szerepet kap. A dokumentum tárgyalja a Black és White-box tesztelési technikákat, valamint ezek szerepét és fontosságát. Ezek után a CanTp modulhoz kapcsolódó szoftver tesztelést mutatom be, azon belül is a követelmények áttekintését, a tesztkörnyezet felépítését és az egyes teszt suite-ok és esetek megvalósítását. Minden létrehozott teszteset sikeresen átmegy, valamint a modul branch-coverage értéke eléri a 100%-ot, tehát a modult a tesztjeim teljes mértékben lefedik. A diplomamunkám során elvégeztem a modulhoz tartozó generátor tesztelését, amelynél megbizonyosodtam a hibás konfigurációs bemenetek esetén történő informatív üzenetek szolgáltatásáról, valamint arról, hogy a generátor ezekben az esetekben nem szolgáltat hibás konfigurációs fájlokat. Végezetül a CanTp integrált tesztelését végeztem, amely során megismertem az egész CAN stack-et, annak moduljait és létrehoztam a működéshez megfelelő konfigurációt. A feladat során a Dcm modul interfészeit felhasználva I-PDU-kat hoztam létre, majd ezeket az I-PDU-kat felhasználva üzenetátvitelt kezdeményeztem a CAN hálózaton. A hardver két CAN csatornáját összekapcsolva azonos ECU-n valósítottam meg az üzenetek küldését, fogadását, valamint a hiba injektálást, amely alapján a CanTp-t helyes működéséről megbizonyosodtam. A CanTp modul a szoftver- és az integrációstesztek alapján a követelményeknek megfelelően működik.

Rövidítésjegyzék

CanIf	CAN Interface
CanTp	CAN Transport Layer
PduR	PDU Router
Det	Default Error Tracer
BfS	buffer size
STF	StartFrame
PDU	Protocoll Data Unit
N_Ar	A CAN keret átviteli ideje (bármely N-PDU) a vevő oldalon
N_As	A CAN keret átviteli ideje (bármely N-PDU) az adó oldalon
N_Br	Idő a következő FlowControl N-PDU kiküldéséig
N_Bs	Idő a következő FlowControl N-PDU fogadásáig
N_Cr	Idő két egymást követő ConsecutiveFrame N-PDU fogadása között
N_Cs	Idő két egymást követő ConsecutiveFrame N-PDU kiküldése között
PCI	Protocol Control Information
SF	SingleFrame
FF	FirstFrame
CF	ConsecutiveFrame
FC	FlowControl
STmin	SeparationTime minimum
Sa	Source Address
Ta	Target Address
Ae	Address Extension

Bevezetés

A modern gépjárművek biztonságtechnikai és kényelmi funkcióinak megvalósításában, környezetvédelmi jellemzőinek javításában stb. egyre jelentősebb szerepet kapnak a számítástechnikai megoldások. Ma egy prémium személyautó gyártójának közel 150 elektronikus vezérlőegységből (ECU) és számos fedélzeti kommunikációs sínből kell kialakítani egy megbízhatóan működő elosztott rendszert, amely komoly algoritmus- és kommunikációtervezési, illetve munkaszervezési kihívást jelent. Az így adódó komplexitás uralására alakultak ki különféle szabványok, pl.: a megbízható kommunikáció biztosítására a CAN és FlexRay sínek, a valós idejű feladatok futtatására az OSEK operációs rendszer vagy a futási idejű monitorozást támogató XCP protokollcsalád.

A vezető autógyártók által 2002-ben életre hívott AUTOSAR konzorcium célja az, hogy ezen szakterületi szabványokra építve specifikáljon egy (i) alapvető szolgáltatásstruktúrát, amely eltakarja a hardver sajátosságait és támogatja az alkalmazási szoftver hordozhatóságát (base software stack, BSW), (ii) egy modellezési nyelvet az ECU-kon futó alkalmazási szoftver szabványos leírására (software component template), és (iii) az alkalmazások és BSW-k ECU-n belüli és ECU-k közti transzparens kommunikációját lehetővé tevő elosztott runtime szolgáltatást (RTE):

A base software stack magában foglalja az alacsony szintű eszközmeghajtókat (pl.: EEPROM és Flash driverek), az ezeket eltakaró absztrakciós rétegeket (pl.: memória absztrakciós felület) és az ezekre ültetett magas szintű funkciókat (pl.: perzisztens adattárolás).

A modellezési nyelv lehetővé teszi, hogy precízen specifikáljuk az adattípusokat, illetve az alkalmazást alkotó komponensek interface-eit és belső felépítését.

Az RTE egy generált glue kód réteg, amely eltakarja az alkalmazáskomponensek elől azt, hogy az általuk fogadott vagy küldött információ pontosan hogyan jut el a forrástól a célig, potenciálisan ECU-k közötti kommunikációs buszok igénybevételével.

A konzorcium jelentős hangsúlyt fektet az API-k szabványosítására, de kifejezetten támogatja a versengést az egyes szolgáltatások megvalósításában („Cooperate on standards, compete on implementation”). Az AUTOSAR egy élő, aktívan fejlesztett szabvány, amelynek évente jelenik meg újabb verziója.

Ez a dokumentum összefoglalja a szükséges előzetes ismereteket az AUTOSAR CanTp modul működésének megértéséhez. Ezenfelül tárgyalja a modul teszteléséhez elengedhetetlen információkat, metódusokat, amelyek segítségével a CanTp tesztelését elvégeztem szoftveres és hardveres környezetben egyaránt.

1 KOMMUNIKÁCIÓS PROTOKOLLOK

A mai világban a modern járművekben a kormányműveket és a fékcsöveket vezetékekre és integrált áramkörökre cserélik. Ezeknél az újszerű megoldásoknál az adatáramlásban a biztonság, a robusztusság és a hibatűrő viselkedés megkövetelt és elengedhetetlen. A CAN egy, már létező multi Master broadcast serial bus kommunikációs protokoll, amely kapcsolatot biztosít az ECU-k között az autóipari alkalmazásokban. A FlexRay a CAN-hez képest újabb fejlesztésű protokoll, amely a növekvő számú biztonsági és komfort követelményeket elégíti ki.

Manapság az autóipari rendszereket komplex, elosztott, beágyazott hálózatok alkotják. A nem kritikus rendszerek esetén számos különböző kommunikációs protokollt alkalmaznak (pl.: LIN, Bluetooth, ZigBee). Habár a CAN egy olyan serial protokoll, amely hatékonyan támogatja az elosztott valósidejű alkalmazásokat, az újabb kihívásokkal és sebességigénnyel már nehezen veszi fel a versenyt. Általánosságban a CAN egy eseményvezérelt kommunikációt valósít meg, ahol minden node az aktuális buszon hozzáfér minden üzenethez, amelyek sebessége maximum 1Mbit/sec lehet.

Biztonságkritikus és komplexebb alkalmazások esetére kifejlesztettek számos protokollt, mint például a Byteflight, TTP/C, TT-CAN, FlexRay. Az autóiparban vegyesen használják ezeket, viszont a nagysebességű alkalmazások esetén a FlexRay vagy az Ethernet a legtöbbször felhasznált.[10]

Az autókban található számos vezérlőeszköz, érzékelő és aktuátor közötti adatcsere jellemzően CAN hálózaton keresztül történik. Az új X-by-wire rendszerek bevezetése megnövekedett követelményeket eredményez, különösen a hibatűrés és az üzenetátvitel időbeli kiszámíthatóságának tekintetében.

A CAN története az 1980-as években kezdődik. A Robert Bosch GmbH kifejlesztette az új hálózati kontrollert, amely a vezetékes rendszerek egyszerűsítését szolgálta. A CAN serial buszként nagysebességű és megbízhatóságú, viszont alacsony költségű rendszernek lett fejlesztve az elosztott hálózatok számára. Az EMC toleranciája és az öndiagnosztikája miatt egyre népszerűbb és elterjedtebb lett ipari környezetben.

Alacsony költsége, nagy teljesítménye, bővíthetősége és rugalmassága miatt a kutatók kezdeményezték ennek a protokollnak a katonai, repülési, elektronikai, gyári és sok más iparágban történő bevezetését.

1.1 CAN-FD (Flexible Data Rate)

A technológia fejlődésével a lehetőség adott volt, hogy a nagyobb sebesség igényű alkalmazásoknál a mérnökök alternatív kommunikációs protokollokat használjanak, pl.: FlexRay vagy Ethernet. Viszont ezek használata a CAN-hez képest költségesebb.

Tehát az autóipar sávszélességi követelményei miatt a CAN adatkapcsolati réteg protokollját tovább kellett fejleszteni. A Bosch 2011-ben az autógyártókkal és más CAN-szakértőkkel szoros együttműködésben megkezdte a CAN-FD (rugalmas adatátviteli sebesség) fejlesztését. A továbbfejlesztett protokoll két CAN-korlátozást hidal át: 1 Mbit/s-nál gyorsabban lehet adatokat továbbítani, és a hasznos teher (payload), másnéven adatmező, mostantól akár 64 bájt hosszú is lehet, azaz nem korlátozódik többé 8 bájtra. Általánosságban az ötlet egyszerű, ha csak egy csomópont sugároz a bitráta növelhető, mivel nem kell szinkronizálni a csomópontokat. Természetesen az ACK slot bit átvitele előtt a csomópontokat újra kell szinkronizálni. A CAN-FD kerethossza konkrét értékeket vehet fel, azaz: 8, 12, 16, 20, 24, 32, 48, 64 bájt méreteket. Amennyiben úgy alakulna, hogy az adatmező hossza nem egyezne meg az előbb felsorolt diszkrét értékekkel, ebben az esetben kötelezően valamilyen előre meghatározott értékkel ki kell tölteni a keret maradék bájtjait, ameddig el nem éri a keretméret a hozzá legközelebb eső értéket a felsoroltak közül.

1 Mbit/s- ról az akár 8 Mbit/s-os sebességre való változás szembetűnő különbség. A megnövelt adatsebesség nagyon előnyös az olyan alkalmazások újraprogramozásánál, ahol nagy adatcsomagok átvitelére van szükség. Összehasonlítva a 8 bájtos CAN üzenetekkel ez egy hatalmas ugrás a hasznos adatok átvitelében, mivel megnőtt a gyorsaság a kommunikációban és ezáltal az ECU-k hatékonysága is a járművekben.

Az üzenetek keretformátuma átment néhány jelentős változáson a CAN-FD busz szabványban. A legjelentősebb változás az a lehetőség, hogy a vezérlőadatokat (arbitráció és nyugtázás) eltérő bitsebességgel (általában 500 Kbit/s) küldjük, viszont a tényleges adatokat nagyobb bitsebességgel. A CAN-FD továbbfejlesztett ciklikus redundancia-ellenőrzést (CRC) és "védett stuff-bit számlálót" használ, amelyek csökkentik a detektálatlan hibák kockázatát. Ez létfontosságú a biztonság szempontjából kritikus alkalmazásokban, mint például a járművek és az ipari automatizálás. A CAN-FD és a CAN protokollok bizonyos feltételek mellett alkalmazhatóak együtt, ezzel lehetővé téve a CAN-FD fokozatos bevezetését.[13]

A klasszikus CAN kétféle adatkeretet támogat, az alapvető különbség csak a CAN azonosítóban található. A "Classical Base Frame Format (CBFF)" 11 bites azonosítót, míg a kiterjesztett "Classical Extended Frame Format (CEFF)" a 29 bit hosszúságút támogatja.

1. ábra: Klasszikus CAN adatkeret.

A CAN adatkeret (1. ábra) a "Start Of Frame" (SOF) bittel kezdődik. Ennek a bitnek a segítségével a hálózat összes csatlakoztatott csomópontja szinkronizálódik egy CAN-adatkeret átvitelének időtartamára. Ezt követi az "Arbitration field", amely a CAN azonosítót és a "Remote Transmission Request (RTR)" bitet tartalmazza. Az RTR bit a CAN adatkeret és a CAN távoli keret közötti különbségtételre szolgál. Az ezt követő "Vezérlőmező" tartalmazza az "Identifier Extension (IDE)" bitet, valamint a "Data Length Code (DLC)" bitet. Az IDE bit lehetővé teszi a CEFF és a CBFF megkülönböztetését, ami az arbitrációs folyamat része. Ez azt jelenti, hogy egy CBFF-ben lévő adatkeret a CEFF-ben lévő adatkerettel szemben az első 11 CAN-ID bit azonos értékeit használva megnyeri az arbitrációt.

A DLC az azt követő adatmező méretét egy bájt többszöröseként fejezi ki. Az adatmező az alkalmazási adatokat tartalmazza, amelyek nulla és legfeljebb nyolc adatbájt között változhatnak. Az adatkeretek integritását a ciklikus redundáns ellenőrzés (CRC) összege garantálja. Amennyiben egy adatkeretet eddig a pontig helyesen továbbítottak, az összes fogadó csomópont az acknowledge (ACK) mezőben, az ACK slot bit domináns bitjének továbbításával megerősíti a helyes vételt. A keretek végét a "Keret vége (EOF)" mező jelzi, amelyet a 3 bites szünet mező követ, ami az egymást követő kereteket elválasztó bitek minimális száma. Hacsak egy másik csatlakoztatott csomópont nem kezd adást, a hálózat ezután üresjáratban marad. A szünet mező része a keretközi térnek (IFS), amely tartalmazhatja a busz üresjárati idejét és a 8 bites felfüggesztett adást hibás passzív csomópont esetén.

A kiterjesztett keretformátum és az alapkeretformátum közötti különbség a használt CAN-azonosító hossza. A 29 bites azonosító a 11 bites alap azonosítóból és egy 18 bites kiterjesztésből áll. Az alap keretformátum és a kiterjesztett keretformátum közötti

különbségtétel az IDE bit használatával történik. A kiterjesztett keretformátum (29 bites CAN azonosító) esetén az IDE bitet recesszíven, míg az alap keretformátum (11 bites CAN azonosító) esetén dominánsan kell továbbítani. A kiterjesztett keretformátumnak van néhány kompromisszuma, mint a hosszabb, azaz legalább 20 bites hálózati késleltetési idő és a kiterjesztett formátumú adatkeretek magasabb sávszélesség igénye. Ezen kívül a kiterjesztett keretformátum hiba detektálási teljesítménye is alacsonyabb, mivel a 15 bites CRC-hez választott polinomot a 112 bites kerethosszig optimalizálták.

A klasszikus adatkeretek és a CAN-FD adatkeretek megkülönböztetésére a korábban fenntartott bitek egyike, az FDF vagyis FD keret bit használatos. Ha az FDF bit recesszív értékű, akkor a következő bitsorozatot CAN-FD adatkeretként kell értelmezni, viszont amennyiben domináns értékű, akkor klasszikus adat- vagy távoli keretről van szó. Az újonnan bevezetett BRS (bitráta kapcsoló) bitben a második bitráta kerül alkalmazásra, ha recesszív értékű. Ha domináns értékű, akkor az adatfázisban is az arbitrációs fázis bitidő-beállítása kerül alkalmazásra.

A CAN-FD protokollvezérlőnek támogatnia kell a klasszikus CAN kereteket is. Mindkét CAN protokoll (a klasszikus és a CAN-FD) nemzetközileg szabványosított az ISO szabványban. A 11 bites azonosítókkal rendelkező CAN-FD adatkeretek az FBFF (FD alapkeret formátum), a 29 bites azonosítókkal rendelkező adatkeretek pedig a FEFF (FD kiterjesztett keret formátum) formátumot használják.

1.2 FlexRay

A FlexRay protokoll kommunikációs infrastruktúraként szolgál a járművek nagy sebességigényű alkalmazásai számára. A protokoll képes a determinisztikus, ciklus alapú üzenet küldésre, valamint a szinkronizációs eljárásokhoz közös időalapot biztosít az egyes node-ok számára. A FlexRay továbbá hibakezelést és hibajelzést kezelő menedzsment szolgáltatást kínál, valamint egy energiagazdálkodáshoz hasznos "wakeup service"-t is.

A FlexRay 2.1 előnyei közé tartozik a 10 Mbit/s sebességű üzenet továbbítás, illetve a két csatorna, ami kétszeres sávszélességet vagy redundanciát biztosít a hibatűréshez. Többféle konfiguráció létezik az üzenetek bufferelésére, mint a változtatható buffer méret, fogadó oldali buffer és a single vagy double buffered üzenet küldési lehetőségek. Továbbá lehetővé teszi az üzenetek szűrését a keret azonosító, a ciklus számláló és csatorna alapján az üzenetek küldéséhez és fogadásához egyaránt.

A FlexRay kommunikációs ciklus az alkalmazás igényeitől függően statikus és dinamikus szegmensekre oszlik. A statikus adatátvitel lehetővé teszi az idővezérelt kommunikációt, hogy megfeleljen az időkritikus rendszerek követelményeinek is, míg a dinamikus átvitel lehetővé teszi, hogy minden csomópont a teljes sávszélességet használja az eseményvezérelt kommunikációhoz. A FlexRay statikus adatátvitele determinisztikus, minimális üzenetkésleltetéssel és ingadozással garantált. A FlexRay támogatja a redundanciát és a hibatűrő elosztott óraszinkronizálást egy globális időalaphoz, így az összes hálózati csomópont ütemezését egy szűk, előre meghatározott precíziós ablakon belül tartja.[9]

A FlexRay szabvány támogatja mind az optikai, mind az elektromos fizikai rétegeket, hogy lehetővé tegye a felhasználó számára az igényeinek leginkább megfelelő alkalmazási sémát. A FlexRay rendszer magasfokon skálázható, az egycsatornás busztól a kétcsatornás többcsillagos topológiákig, teljes redundanciával. Még az egy- és kétcsatornás csomópontok egy rendszerben való keverését is lehetővé teszi a költséghatékonyabb megoldások biztosítása érdekében.

2 AZ AUTOSAR BASIC SOFTWARE ELHELYEZKEDÉSE, FELÉPÍTÉSE, MŰKÖDÉSE

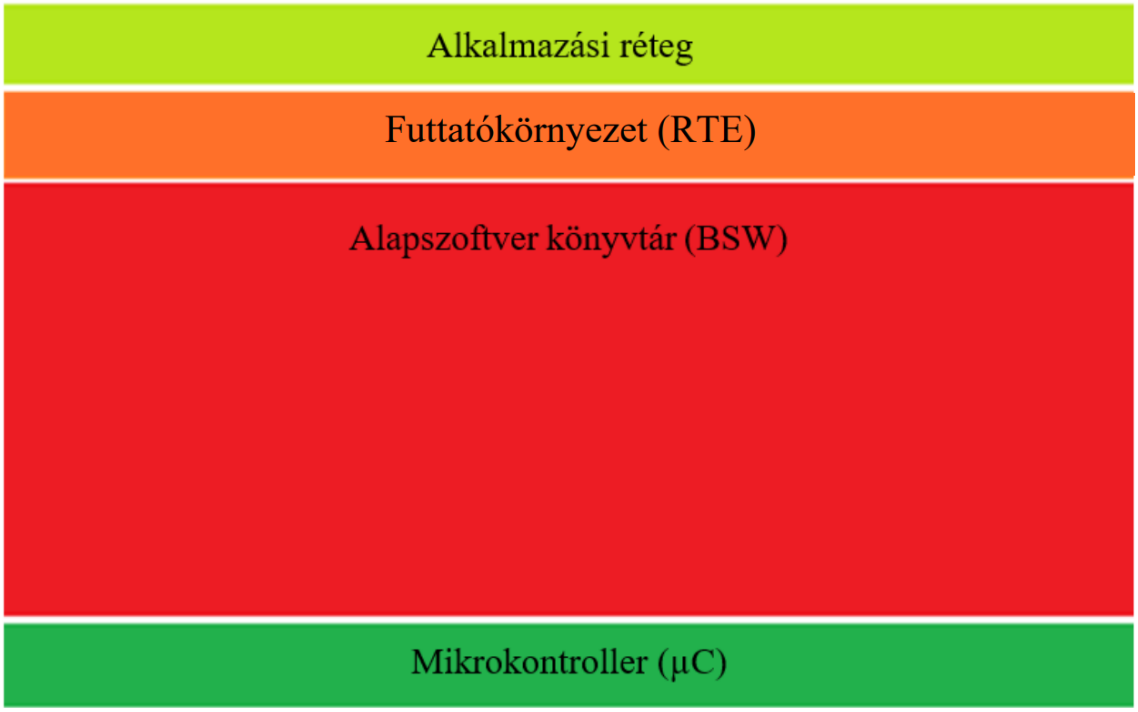
Az AUTOSAR a mai modern járművek elektronikai vezérlőegységei számára kifejlesztett szabványcsalád. A szabvány tartalmaz alapvető szolgáltatásokat, amelyek a hordozhatóságot szolgálva eltakarják a hardverspecifikus tulajdonságokat és egységes kezelőfelületet mutatnak az alkalmazási szoftverek számára. Ez az alapszoftverréteg a BSW (Basic Software stack), amely tartalmazza az alapszolgáltatásokat. Az AUTOSAR az autóipari cégek és beszállítóik által mai napig is írt és szerkesztett szabvány, amely komponensorientált szoftverarchitektúrát fejleszt. Ez azt jelenti, hogy a BSW-ben működő egységek egymástól jól elkülönülő funkcionalitással rendelkeznek. A szabványcsalád három fő részre bontható.

A komponensorientált modellezési nyelvre, amely egy szabványos felületet teremt, a gazdag alapszoftverkönyvtár, amely segítségével létrehoztak egy szabványos modellező nyelvet és lehetővé tették a dinamikus kódgenerálást az egyes modulok együttműködése érdekében.

A modulok tesztelhetőségi és megfigyelhetőségi vizsgálatára létrehoztak egy tesztkészletet és folyamatot, amellyel a modulokat vizsgálhatják a követelmények szerint.

Mivel teljesítményhatékonyság és ezáltal költség szempontjából a C programozási nyelv a legalkalmasabb beágyazott rendszerek fejlesztésére, így a magasszintű nyelvek hasznos fogalmait nélkülözni kell (pl.: osztály vagy névtér fogalma). Ezeket konvenciókkal és kódgenerálással helyettesítik, ami magasabb szintű nyelvek használatával (pl.: Java) történik.

A szabvány három különböző absztrakciós szintet különít el egymástól, amelyek az Alkalmazási réteg, a Futtatókörnyezet, azaz RTE (Runtime Environment) és az Alapszoftver könyvtár (BSW). [1](2. ábra)



2. ábra: Rétegzett szoftverarchitektúra[1]

Az alkalmazási rétegben kapnak helyet azok a komponensek, amelyek teljesen hardverfüggetlenek, így alkalmazásuk minden AUTOSAR kompatibilis környezetben lehetséges. A futtatott szoftverkomponensek szabványosított portokon kapcsolódnak a Virtual Function Bus-hoz (VFB), ami az alkalmazásoktól független módon látja el a kommunikációt az alkalmazások és a környezet között, így függetlenítve az applikációkat a vezérlőegységtől. Az RTE-t a konfigurációtól függően generálják, ezt a réteget látják az alkalmazások virtuális busznak. A BSW legfelső szinten lévő moduljai ezen keresztül képesek kiszolgálni a „hasznos" alkalmazások igényeit és hozzáférést biztosítani a mikrovezérlő szolgáltatásaihoz. A BSW egy szabványos interfészekkel rendelkező moduláris könyvtár, amelynek egyes elemei a vezérlőegységhez mérten konfigurálhatóak, valamint a követelmények függvényében testreszabhatóak.

A BSW tovább bontható további négy részre. Szolgáltatási, ECU Absztrakciós, Mikrokontroller absztrakciós és Komplex Driver rétegekre. (3. ábra)

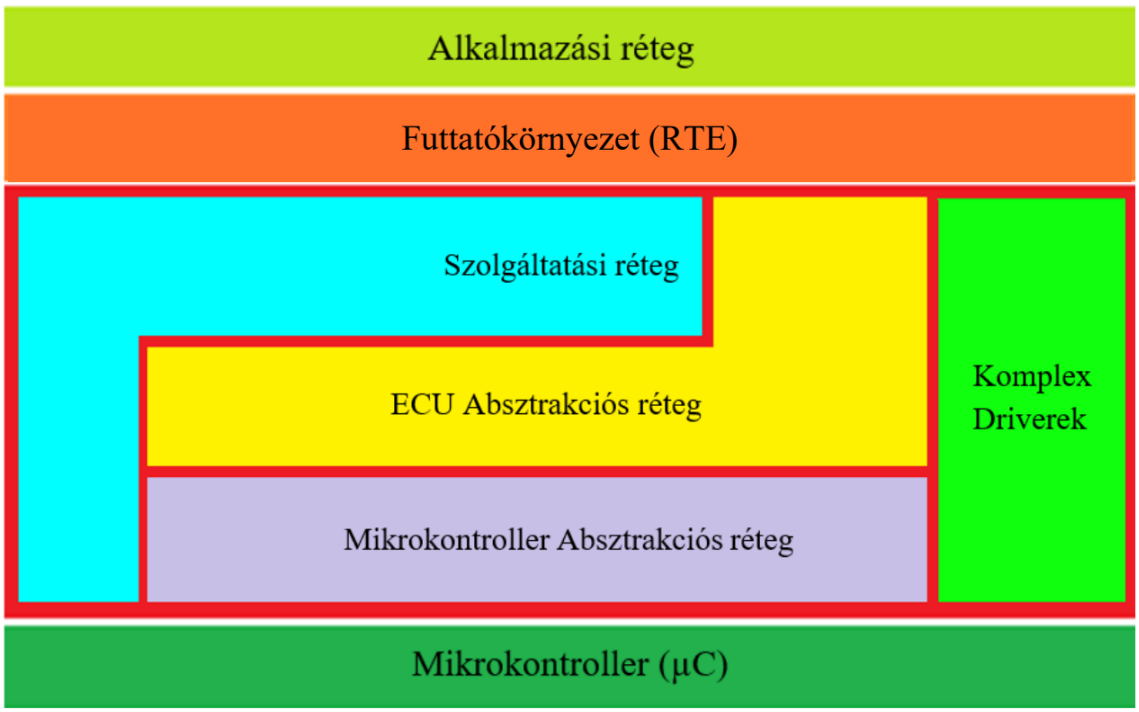
A Komplex Driver réteg az AUTOSAR által nem szabványosított szolgáltatásokat végző modulokat foglalja magába. Közvetlenül az RTE és a kontroller között helyezkedik el, így direkt módon csatlakozik a hardver szenzor és beavatkozó szerveihez.

Tartalmazhat például olyan funkciókat, melyeknek szigorú időzítési feltételeknek kell megfelelniük vagy különleges biztonsággal kell a feladatukat ellátni.

A Mikrokontroller absztrakciós réteg a BSW legalsó szintjén helyezkedik el. Drivermodulokból áll, melyek feladata biztosítani a hardverfüggetlen hozzáférést a mikrokontroller perifériáihoz és funkcióihoz.

Az ECU absztrakciós réteg a driverek szolgáltatásaira támaszkodva nyújt hozzáférést a vezérlő-egység funkcióihoz függetlenül attól, hogy az adott funkciót megvalósító periféria a mikrokontroller belső vagy külső perifériája.

A Szolgáltatási réteg a BSW legfelső szintje, amely hardverfüggetlenül alapvető szolgáltatásokat nyújt a BSW modulok, valamint az RTE-n keresztül az alkalmazási réteg számára. Itt történik a hálózatkezelési és -menedzselési feladatok lebonyolítása, többek között a memóriamenedzsment feladatainak ellátása, valamint az ECU állapot és futási módját is kézben tartja. Ezek mellett a jármű diagnosztikájáért felelős modulok is itt találhatóak.



3. ábra : A BSW felosztása

Az alapszoftvereket tovább bonthatjuk szolgáltatásaik szerint. (4. ábra) Az illesztőprogramok a belső vagy külső eszközök vezérlésének és elérésének funkcióit tartalmazzák. A be- és kimenetekért felelős (I/O) szolgáltatások szabványosított hozzáférést biztosítanak az érzékelőkhöz, beavatkozókhoz és a mikrokontroller fedélzeti

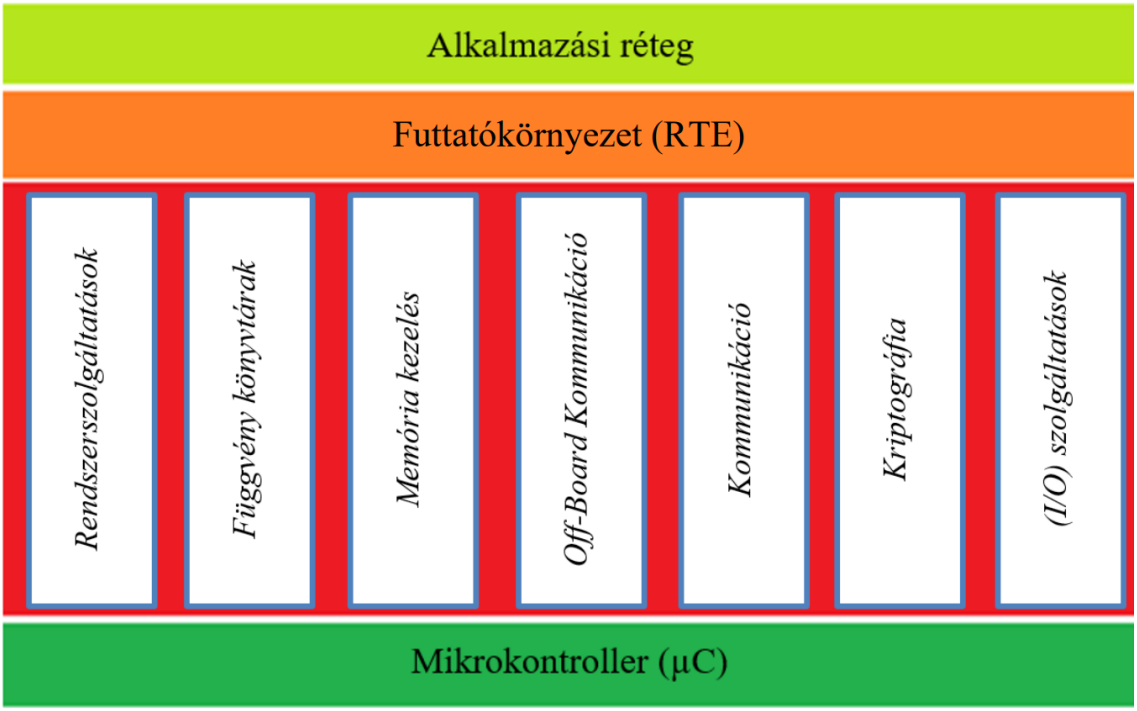
perifériáihoz. Az I/O hardver absztrakcióhoz tartozó modulok csoportja függetleníti a perifériákat a fizikai elhelyezkedésüktől, mivel előfordulhat, hogy azonos porton akár több szenzor helyezkedik el.

A modulok egy csoportja a nemfelejtő memóriához való szabványosított elérést teszi lehetővé, függetlenül a hardverelrendezéstől. Külső vagy belső EEPROM esetén is az elérési metódus megegyezik.

A kriptográfiáért felelős modulok kontrollerfüggetlenül kínálnak egységes hozzáférési mechanizmust a belső és szoftveres titkosítási eszközökhöz.

A hibakezelést, a feladatok végrehajtásának ütemezését és a valósidejűségért felelős operációsrendszeri feladatokat (pl.: időzítési szolgáltatások) a rendszerszolgáltatások csoportja látja el.

A belső kommunikációért felelős modulok standardizált metódust nyújtanak a vezérlőegységek és a szoftverkomponensek közötti információcseréhez. Minden kommunikációs rendszerhez egy specifikus kommunikációs hardver-absztrakció szükséges (pl.: LIN, CAN, FlexRay, Ethernet).



4. ábra: BSW Funkcionális felosztásban

3 KOMMUNIKÁCIÓS STACK

A járműben található vezérlőegységek elosztott hálózatként működnek, amelyben minden vezérlő saját egyéni feladatát kell, hogy elvégezze. Folyamatos kapcsolatban állnak egymással, így csak a feladatukhoz legszorosabban kötődő szenzorok által mért jelek alapján szükséges számításokat kell végezniük. Az egyéb működéshez szükséges adatokat a többi egységtől gyűjtik be. Ebből is következik, hogy az alkalmazások hatékonysága erősen függ a kommunikáció minőségétől.

A különböző feladatkörök ellátásához eltérő sávszélesség és adatátviteli sebesség szükséges, ebből adódóan az autóiparban leggyakrabban alkalmazott kommunikációs protokollok, mint a CAN, LIN, FlexRay és az Ethernet is különféle esetekhez használható.

A LIN kisebb és kevésbé kritikus egységek elérésére szolgál (pl.: ablakemelő vezérlése). Jellemzően alhálózataként, buszrendszerben működik a CAN-hez kapcsolódva egy köztes eszköz segítségével.

A CAN aszinkron és eseményvezérelt kommunikáció, amely adatátviteli sebessége legfeljebb 1Mbit/s és egy keret maximum 8 bájt méretű lehet. A járművekben központi hálózatként, buszrendszerben működik. A nagyobb bitráta és az átvihető keretek növelése érdekében a klasszikus CAN helyett egyre inkább az újabb fejlesztésű CAN-FD (Controller Area Network Flexible Data-Rate) protokollt használják.

A FlexRay a legfiatalabb protokoll. Nagysebességű (10 Mbit/s), hibatűrő és redundáns, ezáltal idő- és biztonságkritikus feladatok esetén is alkalmazható.

Az Ethernet az autóiparban újnak mondható, egyébként jól ismert hálózati technológia. Eredetileg csak diagnosztikai alkalmazásokhoz és elektromos járművek intelligens töltéséhez használták, ma már az Ethernet hálózatok megvalósítására is lehetőség nyílik a járművekben.

3.1 Rétegek közötti kommunikáció

A kommunikációs szolgáltatásokért felelős modulok a szolgáltatási rétegben helyezkednek el, ahonnan a kommunikációs hardver-absztrakción keresztül kapcsolódnak a kommunikációs meghajtókhoz. A járműhálózat kommunikációs moduljainak egy csoportja tartozik ide (CAN, LIN, FlexRay és Ethernet). A szolgáltatások egységes interfészt nyújtanak az alkalmazási réteg számára az RTE-n

keresztül. Elrejtik a protokoll- és üzenettulajdonságokat, továbbá egységes szolgáltatásokat nyújtanak a hálózatkezeléshez.

Az adott protokoll által értelmezhető adategység a PDU. (Protocol Data Unit) Adatok küldése esetén az adott réteg PDU-ja, az alsóbb kapcsolódó rétegben SDU-ként (Service Data Unit) jelenik meg, ahol kiegészül a vezérlőinformációkkal és így, egy új PDU-ba csomagolva halad tovább a rétegen. Fogadásnál a vezérlőinformációkat leválasztva, és az alapján kicsomagolva továbbítja az SDU-t az alsó réteg a felsőbb felé.

A kommunikációs szolgáltatásokon belül az egységeket két csoportba, a protokollfüggő és -független modulok csoportjába sorolhatjuk. Független modulokból ECU-nként egy példány létezik. Ilyen modul az AUTOSAR COM, a Diagnostic Communication Manager, a Communication Manager, a Generic Network Management Interface és a PDU Router.

Az AUTOSAR COM modul feladata továbbítani az alkalmazásoktól érkező szignálokat, megszűrni, majd I-PDU-kba (Interaction Layer Protocol Data Unit) csomagolni azokat. Hasonlóan feladata a hálózatról érkező PDU-k kicsomagolása és ezek alapján szignálok továbbítása az adott alkalmazásnak az RTE-n keresztül.

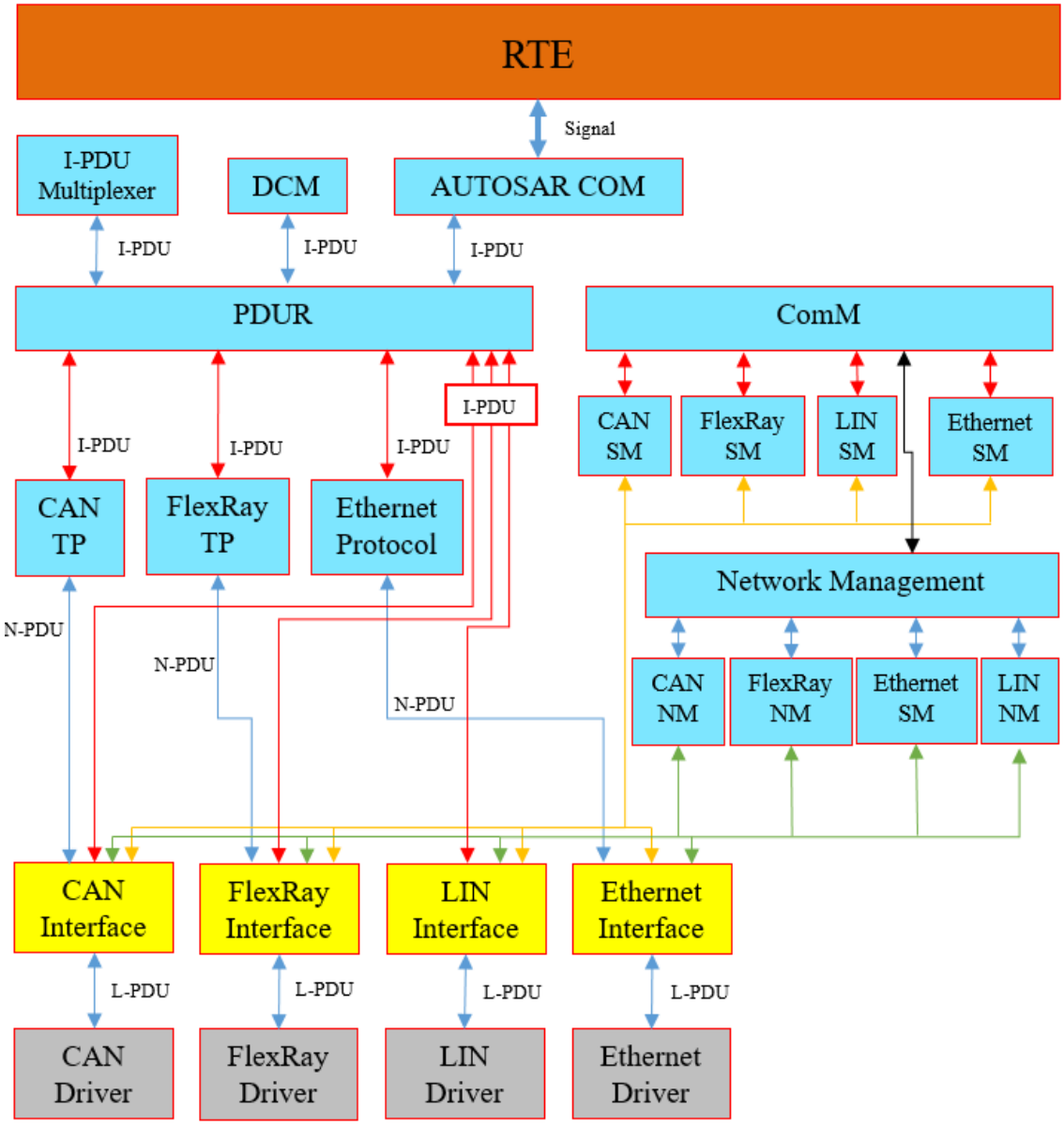
A Diagnostic Communication Manager (DCM) a diagnosztikai jellegű üzenetek forgalmáért felelős. Biztosítja a diagnosztikai adatáramlást és kezeli a diagnosztikai állapotokat. Ezenkívül ellenőrzi a diagnosztikai szolgáltatáskérés érvényességét, hogy a szolgáltatás az aktuális munkamenetben végrehajtható-e a diagnosztikai állapot szerint [3].

A Communication Manager (ComM) kezeli a hálózati kommunikációs erőforrásokat, a hálózatok buszait és hardveregységeit összehangoltan és megbízhatóan működteti, mellyel a kommunikációs buszok használatát egyszerűvé teszi a felhasználók számára. A modul egyszerre több kommunikációs csatorna kezelésére is képes, amelyekhez egyenként tartozik egy állapotgép, ami a hardveregységeket állítja a megfelelő State Manager (SM) modul segítségével. A busz állapotát a Network Management (NM) modulon keresztül képes kezelni.

A Network Management Interface egy adaptációs réteg az AUTOSAR Communication Manager és az AUTOSAR buszspecifikus hálózatkezelő modulok (pl.: CAN Network Management és FlexRay Network Management) között.

A PDU Router a PDU-k közlekedését irányítja egy kapcsolási tábla alapján a kommunikációs stack-ben a modulok között. A felsőbb réteg felől érkező PDU-kat a tábla alapján továbbítja az alsóbb réteg moduljainak, kommunikációs protokollt társítva hozzájuk. Üzenetfogadáskor az alsóbb rétegekből továbbít a felsőbb modulok felé. Gateway funkcióként kommunikációs csatornák összeköttetésére is szolgál, amelyek rendelkezhetnek azonos vagy eltérő protokollal is.

A kommunikációs szolgáltatási egységek második csoportja a protokollfüggő modulokból áll, mint a Transport Protocol (TP) (pl.: CanTp), State Manager (pl.: CanSM) és a Network Management. Ezekből a protokollspecifikus kommunikációs modulokból minden egyes kommunikációs hálózattípusra jut külön-külön.[1]



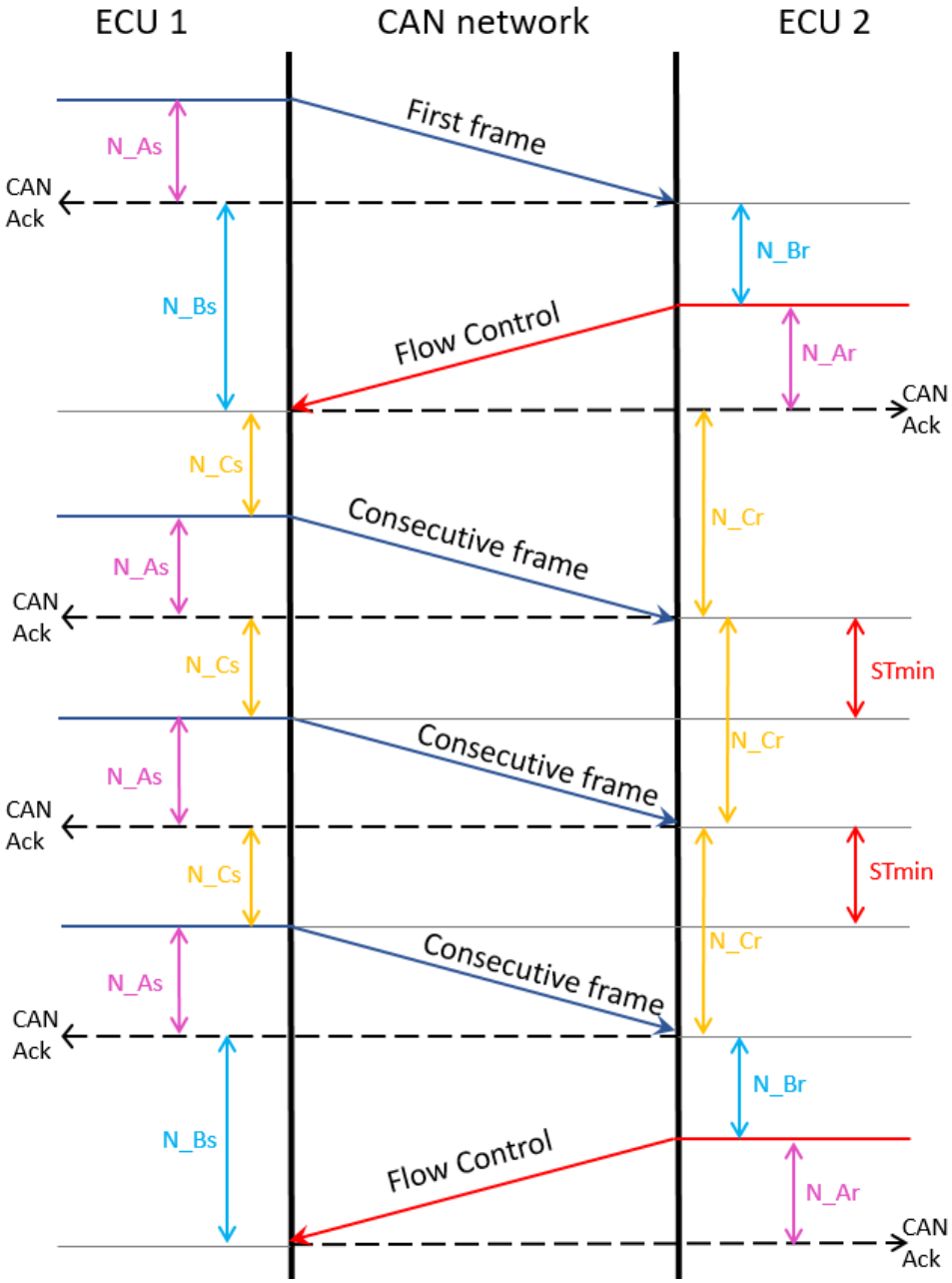
5. ábra: Kommunikációs stack (részlet) [2]

3.2 A CAN Transport Layer

A CAN Transport Layer (CanTp) modul a szolgáltatási rétegben helyezkedik el, a PDU Router (PduR) és a CAN Interface (CanIf) modulok között. A szállításiréteg-protokollok feladata, hogy lehetővé tegyék a maximális fizikai keretméretnél nagyobb üzenetek továbbítását. Ezt a küldő oldalon szegmentálással, a fogadó oldalon pedig egyesítéssel teszik lehetővé. Erre autóiparban tipikusan diagnosztikai körülmények között kerül sor. Például amikor az adott jármű bent áll a szervízben és egy szoftverfrissítést szeretnének telepíteni, amely esetben akár a száz kilobájtot is meghaladhatja az üzenetcsomag mérete. Régebbi eszközök esetén, ahol a maximális fizikai keret hosszúság nem haladja meg a 8 bájtot, előfordulhat, hogy signal alapú kommunikációnál is meghaladjuk a kerethosszt és ekkor is szegmentációra van szükség.

Amint már említettem, a CanTp 4.3.0 feladata a szegmentáció és az újraegyesítés, amit az ISO 15765-2:2016 verzió alapján kell megvalósítania a modulnak. Ennek során lehetőség van, konfigurációtól és a kapott üzenet hosszától függően SingleFrame-ek (SF) átvitelére, ami azt jelenti, hogy a teljes üzenet mérete nem haladja meg a bekonfigurált PDU hosszát, ekkor nem történik szegmentálás. A ISO15765-2:2016 alapján lehetséges a maximális. keretméretet 64 bájtnál kisebbnek is választani, amik a szabványban meghatározott FD értékek lehetnek. Amennyiben meghaladja a kiküldendő keretméret a konfigurált PDU hosszát, az átvitel egy FirstFrame (FF) elküldésével kezdődik, amiben a küldő definiálja és üzenetbe foglalja a teljes üzenet hosszát (DL). Ezután a fogadó oldal küld egy a folyamszabályozásra használt FlowControl (FC) üzenetet, amiben jelzi, hogy elfogadja-e az átvitelt és ha igen, akkor milyen paraméterekkel. A FC-ban lévő paraméterek egyike a BS, amely a két FC üzenet között megengedett maximális fogadható keretek számát (blokk méretet) jelöli. A FC másik paramétere pedig a SeparationTimeMinimum (STmin) ami az egymást követő ConsecutiveFrame-ek (CF) küldése között definiált legkisebb időközt jelenti. A FC üzenet után a küldő oldal a megkapott BS és STmin paraméterek alapján elküldi a CF –eket. Ezek a keretek sorszámozva vannak 0-tól 15-ig, ahol a 0. elem a FirstFrame. Ez a sorszám az összes CF-ben szerepel a PCI mezőben, mint SequenceNumber (SN) érték. Periódikusan ismétlődik, azaz amennyiben több mint 15 keretből álll a teljes üzenet, akkor minden 15. keret után az SN számláló nullázódik. Egy blokk átvitele után a küldő oldal ismét vár a fogadó oldal FC üzenetére, majd annak függvényében folytatódik az átvitel, majd az utolsó CF

elküldésével az átvitel befejeződik. Egy szegmentált átvitelre és az egyes konfigurálható időzítési paraméterek jelentőségére mutat példát a 6. ábra.



6. ábra: CAN üzenetváltás

A CanTp széleskörűen konfigurálható, vannak általános, csatorna specifikus és a csatornákon belül megadható fogadó és küldő oldali konfigurációs beállításai. Általános beállítás az egyes API-k be- és kikapcsolása, amely lehetséges például a CanTp_GetVersionInfo, CanTp_ReadParameter vagy a CanTp_ChangeParameter API-knál, ezek be- illetve kikapcsolhatóak. Konfigurálható a fejlesztésidejű hibák detektálásának engedélyezése, valamint beállítható a PaddingByte értéke, ami alapján a

kiküldött üzenetekben a nemhasznált bájtok értéket kapnak. Bekapcsolható a dinamikus ID kezelés és engedélyezhető a generikus kapcsolatok támogatása.

Mint ahogyan a legtöbb BSW modulnál, itt is konfigurálható a main függvény periódus ideje. Ez nálunk a későbbiekben az időzítők kezelésénél lesz fontos, hiszen az idő múlását a tesztekben a main függvény meghívásával tudjuk számítani és szimulálni.

Továbbá beállíthatunk korlátlan mennyiségű csatornát, amelyek egyaránt lehetnek full- vagy half-duplex módban. A csatornákon belül is megkülönböztetünk küldő (Tx) és fogadó (Rx) oldalakat, amelyeknél beállíthatunk különböző, az üzenetváltásokhoz szükséges időzítőket, címzési formátumot, célcím típust (TaType), valamint padding használatot és különböző címzési adatokat.

Mind a fogadó, mind a küldő oldalon beállítható az N-SDU id, amely a felsőbb rétegek által is ismert azonosítót tartalmazza. Ezenfelül a szabványnak megfelelően beállítható a küldő és a fogadó oldalon egyaránt a Target address (Ta), a Source address (Sa) és az Address Extension (Ae). A fogadó és a küldő oldalon is megtalálható, egy az üzenetek küldésére és egy az üzenetek fogadására alkalmas PDU id, viszont ezek a két oldalon mondhatni pont ellentétes szerepet töltenek be. Egy szegmentált üzenet átvitele során a fogadó oldalon a Tx PDU id a FC üzenetek küldésére szolgál, tehát a küldő oldalon az Rx PDU id szolgál ezen folyamszabályzó üzenetek fogadására. A küldő oldalon a Tx PDU id segítségével kerül kiküldésre, a fogadó oldalon az Rx PDU id-val kerül azonosításra a szegmentált üzenet. Az Rx és Tx oldalakhoz rendelt PDU-k referálhatnak úgynevezett meta adatokat, ilyen esetben generikus kapcsolatról beszélhetünk.

Az AUTOSAR CanTp 4.3.0 szabvány definiál egyfajta MetaData (meta adat) kezelést, amely segítségével a felsőbb rétegben lévő modul képes megadni a címzési adatokat. Amennyiben a CanTp konfigurációban engedélyezték a CanTpDynIdSupport kapcsolót és generikus kapcsolatról beszélünk, a modulnak a MetaData alapján, amely a PduInfoPtr-en keresztül érkezik a felsőbb rétegtől, valamint a konfigurált címzési mód alapján kell összeállítania a megfelelő címzési információkat.

A CanTp-ben az egyes kapcsolatoknál konfigurálhatóak az alábbi címzési módok a következők:

- CanTpStandard - normál címformátum használata.
- CanTpExtended - kiterjesztett címformátum használata.

- CanTpMixed - vegyes 11 bites címformátum használata.
- CanTpNormalFixed - normál fix címformátum használata.
- CanTpMixed29Bit - vegyes 29 bites címformátum használata.

A címformátumtól függően a modul összeállítja az adott PDU-t és a MetaData-t, amelyet továbbít a CanIf modul számára. A különböző címzések esetén a szabvány meghatározza, hogy milyen címzési adatok kerüljenek felhasználásra, azaz, hogy az Sa, Ta, Ae címparaméterek közül melyek kerüljenek be a kiküldendő üzenetbe. Ezen felül a címzési mód alapján kerül be az adott címzési paraméter az adat mezőbe vagy a meta adatok közé, amely alapján majd a CanIf összeállítja a 11 vagy 29 bites CAN azonosítót. Amennyiben nincs engedélyezve MetaData kezelés, a CanTp teljes mértékben a konfigurációban meghatározott címzési paraméterek alapján működik, nem számít milyen adatok érkeztek, mint meta adatok.

A fogadó oldalon továbbá konfigurálhatóak a FlowControl üzenetek paraméterei, azaz a BlockSize és az STmin értékek. Ezenfelül beállíthatunk egy WftMax értéket, ami megadja, hogy hányszor küldhető FC(WAIT), mielőtt megszakítaná a fogadó az üzenetátvitelt.

A küldő oldal különlegessége, hogy az egyes SDU-khoz tartozó üzenetküldések megszakítása engedélyezhető vagy tiltható. Mind a küldő, mind a fogadó oldalakhoz tartoznak időzítési paraméterek, amelyeknek egy küldés, avagy fogadás során meg kell felelni. Ezek a fogadó oldalon az N_Ar, N_Br, N_Cr, míg a küldő oldalon az N_As, N_Bs, N_Cs paraméterek. (6. ábra)

3.3 FlexRay és CAN Transport Layer modulok összehasonlítása

Maga a FlexRay Transport Layer modul a BSW stack-ben, a PDU Router és a FlexRay Interface (FrIf) modulok között helyezkedik el. Alapvetően a FlexRay Transport Layer (FrTp) modul feladata, ahogyan a CAN Transport Layer feladata is, az üzenetek szegmentálása és újraegyesítése. Mind a CanTp és az FrTp esetén is a felsőbb rétegtől a PduR modulon keresztül I-PDU-k érkeznek a modulokba. A 4.3.0 szabványban megtalálható a FlexRay AUTOSAR Transport Layer (FrArTp) és a FlexRay ISO Transport Layer (FrISOTp). Az FrArTp esetén az alapvető ötlet egy olyan Transport Layer modul specifikálása volt, amely a 4.0-ás verzió előtt is megfeleltethető az ISO

15765-2 szabványnak. A 4.3.0 verzióban az AUTOSAR szabvány megfelel az ISO10681–es nemzetközi kommunikációs szabványnak.[18]

Az interfész modulok (CanIf, FrIf) biztosítják a buszhoz való hozzáférést, azok fizikai elhelyezkedésétől függetlenül.

Mind a két protokoll képes:

- Üzenetek szegmentálására és újraegyesítésére
- Üzenetek nyugtázására (Acknowledgement)
- Adatfolyam szabályozására
- Detektálni a hibákat
- Megszakítani egy üzenet session küldését vagy fogadását
- Broadcast és Unicast üzenet küldésre

Hasonlóan a CAN-TP-hez a FlexRay protokollnál is megtalálható a SingleFrame fogalma. FlexRay esetén is a szegmentált átvitel blokkosan történik, viszont van pár plusz dolog, amiben a FlexRay többet tud. Hasonlóan képes ismert hosszúságú üzenetek átvitelére, viszont képes ismeretlen hosszúságú véges üzenetek továbbítására is. FlexRay esetén sokkal nagyobb lehetőség van a visszacsatolásra.

FlexRay esetén, egy acknowledge üzenet átvitel, ami StartFrame_ACK-val (STF_ACK) kezdődött, az mindig tartalmaz egy, a fogadó által küldött, FlowControl_ACK (FC_ACK) válaszüzenetet, amely az utolsó beérkezett keret után kerül kiküldésre. Tehát FlexRay esetén akár egy SF után is elvárt lehet egy FC_ACK megerősítő üzenet.

FlexRay esetén amennyiben egy keretben elfér az egész kiküldendő üzenet, akkor csak egy StartFrame-et küld, mint a CanTp a SingleFrame. Különbséget csak az üzenetek visszaigazolásánál találunk.

Az STF PDU-t a kezdő keret protokollvezérlő információ (STF PCI) azonosítja. Az STF PDU-t a küldő hálózati entitás küldi ki, és egy vagy több fogadó hálózati entitás fogadhatja. Ez a keret szolgálhat akár egy egyedülálló keretként vagy szegmentált üzenetek átvitelére is. Ha az STF ML (üzenethossz) ugyanolyan hosszúságot jelöl, mint amennyi adat ebben a PDU-ban átvitelt nyert, akkor nem szegmentált átvitel történik, és az STF befejezi a továbbítást.[15]

Ha az STF ML (üzenethossz) nagyobb hosszúságot jelöl, mint amennyi adat ebben a PDU-ban átvitelt nyert, akkor szegmentált átvitel történik, és a fogadó kommunikációs réteg entitásának meg kell kezdeni a szegmentált üzenet egyesítésére az STF PDU fogadásakor.

Két STF PDU-t határoznak meg annak megkülönböztetésére, hogy az alábbi átviteltípusok között válasszanak.

- STFU_PDU: Visszaigazolatlan átvitel
- STFA_PDU: Visszaigazolt átvitel

Tehát hasonlóan a CanTp-hez a FlexRay is képes szegmentált átvitelre, viszont a FlexRay protokoll nem különböztet meg külön SF-et, hanem az STF-ben megadott üzenethossz alapján folytatja, vagy fejezi be az átvitelt. Ezenfelül az STF-ben 2 üzenethossz érkezik, az egyik a teljes üzenet hossza, valamint az aktuális payload hossza. Ha ismeretlen hosszúságú az átvinni kívánt adat hossza, akkor a teljes üzenethossz helyére 0 érték kerül, viszont az utolsó keretben összegezve van a valóban átvitt adatméret. (Újraküldött adatokat nem számítva) Ismeretlen üzenethosszúság esetén a felsőbb rétegnek a küldő oldalon szinkron vagy aszinkron módon biztosítania kell az adatokat.

Minden keret, ahogyan a CanTp-nél is láttuk, a vezérlő információk által, a PCI mezőben megtalálható adatok alapján azonosítható.

A CF-ek a CanTp-hez hasonlóan tartalmaznak egy SN számot, ami az átvitt keret sorszámát tartalmazza. Ez az érték minden 15-dik után, vagy amennyiben a FC FlowSatus ACK_RET értékű, visszaáll 0-ra.

Ahogyan a CanTp-nél is láthattuk, a FC PDU utasítást ad a küldő hálózati entitásnak például a CF PDU-k továbbításának indítására, leállítására vagy folytatására. A különbség a két protokoll között az, hogy a FlexRay esetén az utolsó elküldött CF után is átvitelre kerülhet egy FC keret. A FC PDU képes tájékoztatni a küldő hálózati entitást a CF PDU-k továbbításának szüneteltetéséről egy szegmentált üzenetátvitel során, vagy egy szegmentált üzenet továbbításának megszakításáról, ha például jelenleg nem áll rendelkezésre buffer az üzenet tárolására. FlexRay esetén a FlowControl FlowStatus értéke jelezhet folytatást, várakozást, túlcsordulást, abortálást, valamint kérhet újraküldést. A FlowControlnak tartalmaznia kell az első helytelen bájt pozíciót, ahonnan a küldőnek folytatnia kell a küldést.

FlexRay esetén definiáltak egy LastFrame típust, amely PCI mezője tartalmazza a teljes üzenet hosszát. Az utolsó keretet a küldő használja a továbbítás befejezésére (ismert és ismeretlen üzenethosszúság esetén egyaránt). Az üzenethossz (ML) paraméter jelzi az üzenet teljes hosszát, nem számítva a többször továbbított feleslegesen redundáns bájtokat. Ismert üzenethosszúságú továbbítás esetén az üzenethossz már az átvitel elejétől ismert, ezért azonos az StartFrame-ben megadott üzenethosszal.

FlexRay üzenetküldés esetén egy üzenetblokk végén kitüntetett ConsecutiveFrame_EOB üzenet érkezik, amelyre FC üzenetben jelzi a fogadó az aktuális állapotát. [15]

Minden kommunikációs layer szolgáltatásnak ugyanolyan az általános struktúrája. Az egyes szolgáltatások definiálásához három szolgáltatás primitív van specifikálva.

- A request primitívet a felsőbbréteg használja a hasznos és vezérlő adatok kommunikációs rétegbe való továbbítására.
- Az indication primitívet a kommunikációs layer használja a státusz és a hasznos adatok továbbítására a felsőbb réteg felé.
- A confirmation primitívet státusz információ továbbítására használják a felsőbbrétegek felé.

A CanTp és FlexRay modulok API mennyiségre és funkcionalitásra szinte megegyeznek. A CanTp modul esetén definiáltak egy ReadParameter API-t, amely a kívánt id-hoz tartozó BlockSize vagy STmin paramétereket képes visszaadni.

A két modulnak hasonló konfigurációs paraméterei vannak, ugyanis mind a kettőnél statikusan konfigurálható a maximum párhuzamosan aktív csatornák száma. Ezen belül külön konfigurálható az FrTp esetén a fogadó és a küldő oldali PDU-k, viszont itt külön létrehozhatók fogadó és küldő csomagok(pool-ok), amiket utána tetszés szerint lehet az egyes csatornákhoz rendelni. Hasonlóan a kontroll paramétereket (pl.: időzítések) is külön lehet konfigurálni és tetszés szerint csatornákhoz rendelni őket. (CanTp esetén ugyebár létre lehet hozni egy csatornát és ahhoz fix Tx és Rx oldali NSDU-k tartoznak, ahol egyesével megadhatóak a kontroll paraméterek)

A CAN-FD maximális adatsebessége 8 Mbit/s, míg a FlexRay maximális adatsebessége 10 Mbit/s. A CAN-FD-nek van hibajavítási képessége, míg a FlexRay-nek

nincs. A FlexRay időosztásos többszörös hozzáférési (TDMA) megközelítést használ a csomópontok közötti kommunikáció szinkronizálására, míg a CAN-FD egyáltalán nem használ szinkronizációs mechanizmust. A FlexRay keretekhez fix prioritást rendelnek, amelyet rendszertervezés során határoznak meg. A CAN-FD esetében a keretek prioritás alapú arbitrációs sémát használnak annak meghatározására, hogy melyik keret kerül először átvitelre.

4 SZOFTVERTESZTELÉS

Napjainkban már rengeteg olyan eszköz vesz körül minket, ami elektronikus, vagy van olyan része, amihez tartozik vezérlés vagy szabályozás. Ezekben a kontrollerekben, applikációkban programok futnak, amelyek megvalósíthatnak biztonságkritikus vagy egészen egyszerű megoldásokat.

A programok helyes megvalósítása mellett a cégeknek nagy hangsúlyt kell fektetniük a szoftverek kiadása előtt a tesztelésére is, hiszen ezek a tesztek mutatják meg az adott kód robosztusságát, hibamentességét. A tesztekkel növelhető a kiadott termék minősége. Az esetek nagyrészében a tesztelési fázisra több erőforrást kell fordítani, mint magára az implementációra, főleg a biztonságkritikus szoftvereknél. Ilyenek például az orvosi műszerek, vagy a járműipar, ahol ASIL-D szintű alkalmazásokat fejlesztenek, ahol egy meghibásodás akár emberéletekbe is kerülhet. ASIL (Automotive Safety Integrity Level) egy biztonsági integritási szintet jelöl az autóiparban. Az ASIL az ISO 26262 nevű nemzetközi szabványban definiált fogalom, amely a járműelektronika biztonságosságának értékelésére és osztályozására szolgál. Az ASIL a kockázat és a lehetséges sérülések súlyossága alapján határozza meg, hogy egy adott járműrendszert milyen biztonsági intézkedésekkel kell ellátni. Az ASIL négy szintet definiál: ASIL A, ASIL B, ASIL C és ASIL D, ahol az ASIL D a legmagasabb szint, azaz a legmagasabb biztonsági követelményeket jelenti. Ezekre az applikációkra külön szabványosított tesztelési módszerek léteznek, melyek betartása kötelező.

Akármilyen egyszerű is az implementáció, a fejlesztés során minden bizonnyal keletkeznek hibák, amiket sorra kijavítunk. Tehát abban biztosak lehetünk, hogy mielőtt letesztelnénk a modult, az tartalmaz hibákat. Viszont a tesztek után is csak abban lehetünk biztosak, hogy a tesztelt részekben nincs hiba, így a tesztek számával nő a program megbízhatósága.

Fontos megemlíteni pár lényeges alapelvet, melyeket szem előtt tartunk tesztelés során, elsősorban azt, hogy egy teszt képes felfedni hibákat, viszont azt nem, hogy nincs hiba. Egy alkalmazás kimerítő tesztelése nem lehetséges vagy legalábbis nem érdemes, ezért főként a magaskockázatú és prioritású részekre célszerű nagyobb hangsúlyt fektetni. Egy program életciklusában érdemes a tesztelést minél előbb elkezdeni, mert annál előbb fedhetünk fel vele hibákat és annál olcsóbb is a javítás. A szoftver életciklusának későbbi részében a frissítések során a teszteseteket ugyanúgy bővíteni kell, mint ahogy az

implementációt fejlesztik. Ellenkező esetben a tesztek egy idő után már nem fednek fel több hibát. A tesztelésnél mindig figyelni kell a termék céljára és a rendelkezésre álló időre, ugyanis másként kell tesztelni, ha pár napunk van, vagy ha pár hónapunk. Alapvetően más tesztek készülnek egy autóba, mint például egy számítógépes játékba.

A tesztelés elengedhetetlen része egy tesztprogram létrehozása, vagy egy olyan szoftver keresése, amivel ennek vizsgálata megoldható. Ehhez sok szoftver áll rendelkezésre különböző tulajdonságokkal és működéssel. Elsősorban érdemes áttekinteni azokat a tesztelési módokat, amelyekkel egy szoftvert meg lehet vizsgálni.[5]

4.1 Szoftvertesztelési lehetőségek

A gyakorlatban kétféle technikát különböztetünk meg: a feketedobozos (black-box) és a fehérdobozos (white-box) teszteket. A két tesztelési lehetőséget a rendelkezésre álló információk alapján különíthetjük el egymástól. Black-box tesztek esetén a specifikáció alapján készítünk teszteket, míg white-box esetén strukturális tesztekről beszélhetünk, ahol a forráskód áll rendelkezésre.

Feketedobozos tesztek esetén a szoftver interfészeit vizsgáljuk, amelyhez szükség van a specifikációra, valamint a lefordított szoftverre is. Ismerjük a bemeneteket és a program elvárt reakcióját is, így nincs más dolgunk, mint megadni a bemeneteket, lefuttatni a programot és összehasonlítani a valós kimenetet az elvárttal.

Fehérdobozos tesztek esetén mindig egy kész struktúrát, azaz programkódot tesztelünk. Ebben az esetben a struktúra lefedettségét vizsgáljuk, miszerint a meglévő teszteseteink mekkora mértékben fedik le a program működését. Általában a következő struktúrák kerülnek tesztelésre: kódsorok, elágazások, metódusok, osztályok, funkciók, modulok.

Egy komplex rendszer tesztelése esetén érdemes először egyenként a komponenseket tesztelni, majd ahogy készülnek a komponensek, úgy integrálva együtt tesztelni, ezután pedig a rendszertesztet elvégezni. A korábban említett tesztek a fejlesztők feladatkörébe tartoznak, majd miután kiadták a terméket, a felhasználók végeznek úgynevezett átvételi teszteket.

A komponenseken általában black- és white-box teszteket egyaránt elvégeznek, tehát a specifikáció alapján készülnek a tesztesetek a forráskód ismeretében. Ezen tesztek gyakori fajtái a modulteszt és a unit-teszt.

A unit-tesztek alapja, hogy ismerjük az adott bemeneti paraméterekre elvárt visszatérési értéket és mellékhatásokat, tehát magát a folyamat metódusát teszi próbára a teszt. A valós értékeket komparáljuk az elvártakkal és amennyiben a megegyeznek, a teszt sikeresnek mondható. Magának a unit-tesztnek nem járhat mellékhatásokkal, azaz a teszt nem befolyásolhatja a modul működését, attól teljesen független kell, hogy legyen. Tehát ezek a tesztek nagy segítséget jelentenek a komponensek fejlesztésében a szoftver életciklusának teljes hosszában, hiszen ezekkel ellenőrizhető, hogy bármilyen újabb változtatás nem hozott be hibát.

Integrációs tesztek során a komponensek együttműködését vizsgálják, vagyis az illesztéskor fellépő hibák kiderítésére és javítására szolgál. Abban az esetben, ha az egyes komponenseket különböző egyének implementálják, a fejlesztők tehetnek feltételezéseket, amikből adódhatnak illeszkedési problémák. Az integrációs teszteket is érdemes komponensenként, folyamatosan, az egyes részek elkészülésével egyidejűleg végezni a hibák egyszerűbb lokalizálása végett.

Előfordul, hogy a teljes rendszertesztet független cég végzi, ilyen például a feketedoboz teszt. Így a tesztmérnökök teljesen elfogulatlanok lesznek és csak a specifikáció alapján vizsgálhatják a teljes rendszert.

Ezek után jönnek az átvételi tesztek, mint alfa, béta, felhasználói és üzemeltetői tesztek. Itt az alkalmazott környezetben fellépő hibákat figyelik meg, pl.: egyes funkciók lassulása, amik a tesztkörnyezetben nem jelentkeztek. Ilyen béta tesztekről manapság videó játékoknál hallani, ahol a felhasználók egy szűk csoportja megkapja a szoftvert és rövid idő alatt sokat foglalkozik vele.

4.1.1 Követelmény alapú tesztelés

A követelmény alapú tesztelés során minden olyan szempontot figyelembe kell venni, melyek a specifikációban szerepelnek. Ezek a követelmények meghatározzák, hogy a tesztelendő szoftvernek milyen fontosabb kikötéseket kell követnie. Ezek mindegyikének teljesülnie kell ahhoz, hogy a modul követelmény szinten hibamentesnek legyen tekinthető. A követelmények külön-külön tesztelhetők előre megadott bemeneti értékekkel, majd az ezek alapján kapott eredményeket vetjük össze az elvárt viselkedéssel. Az eredményeket vizsgálva ellenőrizzük, hogy milyen végeredménnyel zárult le a szoftver tesztelése.

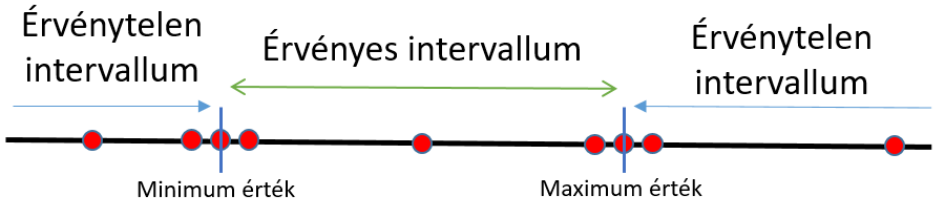
4.1.2 Ekvivalencia osztály alapú tesztelés

A lehetséges bemeneteket szétválogatjuk ekvivalencia osztályokba. A tesztelés előtt az összes követelményt átfogólag meg kell ismerni, hogy az osztályokat hasonló követelmények alapján alakíthassuk ki. Egy ekvivalencia osztály esetén a modul hasonlóan fog működni, mint az osztályon belül megtalálható többi bemenet esetén. Ezekhez az osztályokhoz elég egyetlen tesztesetet írni, mivel hasonló viselkedés várható az osztályba tartozó bemenetek esetén. Ezeket az ekvivalencia osztályokat mi választjuk ki a feladat követelményeinek alapos vizsgálata után. A tesztelés során a kezdeti bemeneti értékeket úgy kell szétválasztani ekvivalencia osztályokra, hogy a bemenet minden eleme csak egyszer szerepeljen egy ekvivalencia osztályban. [7]

Ilyen eset például, amikor csak egy adott intervallumon belül vehet fel helyes értéket egy bemeneti paraméter. Ebben az esetben három ekvivalencia osztályt szükséges elkülöníteni: egyet a helyes tartományban található értékeknek, egyet a kisebb értékeknek és egyet a nagyobb értékeknek. Ezzel a tesztelési módszerrel csökkenthető a szükséges esetek száma, mivel elegendő egy tesztesetet készíteni az adott osztályban.

4.1.3 Határérték analízis (boundary value analysis)

A határérték analízis, angolul boundary value analysis egy black-box tesztelési módszer. A határérték egy ekvivalencia osztály határán helyezkedik el, például egy tartomány maximális vagy minimális értékénél. A határérték analízis a megadott értékhatár közelében elhelyezkedő értékeket vizsgálja meg, mivel magasabb eséllyel a bemeneti- vagy a kimeneti intervallum határértékeinél fordulnak elő hibák. Ezek úgy vizsgálhatók, hogy az intervallum határértékének az elemeit, valamint annál eggyel kisebb és eggyel nagyobb értéket teszteljük, emellett egy olyan közbenső elemet is választunk az intervallum közepéből, amely nagy valószínűséggel hibamentes lesz.



7. ábra: Boundary value analysis

Például, ha egy program bemenete vagy kimenete sorrendezett halmaz (pl.: egy táblázat, egy lineáris lista vagy egy szekvenciális fájl), akkor meg kell figyelni a halmaz első és utolsó elemét. Ahogy a 7. ábra is mutatja, a minimumánál kisebb és a

maximumánál nagyobb értékek helyes működés esetén várhatóan valamilyen hibával térnek vissza. [6] Például egy olyan egyszerű program tesztelésére kell gondolni, amely a számként beírható hónapot megadva az utána következő hónapot adja válaszul, szöveges formában. Ebben az esetben érdemes a 0, 1 és 2 értékeknél, valamint a 11, 12 és 13 értékeknél megnézni a működését. Valamint a köztes hónapokra is megadni egy értéket 3-10-ig, amely bizonyára helyes értékkel tér majd vissza. Érvénytelen eset lehet, ha 0 értéket adunk be. Az elvárt viselkedés egy hiba üzenet küldése vagy a bemenet ignorálása lenne, így, ha a program a "január" értéket adja válaszul úgy felfedeztünk egy hibát a programban. Abban az esetben, ha a programnak érvényes adatot adok meg, mint bemenet, például 12-t és a program helytelenül kezeli, akkor a teszt a program hibás implementációját igazolja.[7]

4.1.4 Code Coverage

A kódfedettség vizsgálat egy olyan white-box vizsgálati teszt, amellyel meg lehet határozni, hogy az adott tesztelendő szoftver mely részét fedte le a tesztelő csomag, és melyeket nem. Ezáltal a kód hibamentes futását tudjuk vizsgálni, valamint azt, hogy nincsenek holt, avagy nem teljesen letesztelt részek a kódban. A végeredmény egy százalékban kifejezett érték. A hibamentes működés annál valószínűbb, minél nagyobb a kódfedettség vizsgálattal megkapott érték.

A kódfedettségen belül is megkülönböztetünk különböző mérőszámokat, ilyen például a function coverage, amely a kód szerkezeti lefedettségével foglalkozik és megadja, hogy a szoftverünkben található függvények mekkora mértékben vannak lefedve a tesztkörnyezet által. Egy program funkciólefedettségének a mérőszámát az a meghívott függvények (alprogramok) és az összes függvény számának aránya adja meg százalékos értékben.

A statement coverage (utasítás fedettség) a teszt által lefedett végrehajtható utasítások százalékos arányát adja meg. A tesztbe beletartozik minden végrehajtható utasítás a tesztelendő kódban. Az utasításlefedettségen keresztül azonosítani tudjuk a végrehajtott utasításokat és azt, hogy mely kódrészletek nem futnak le.

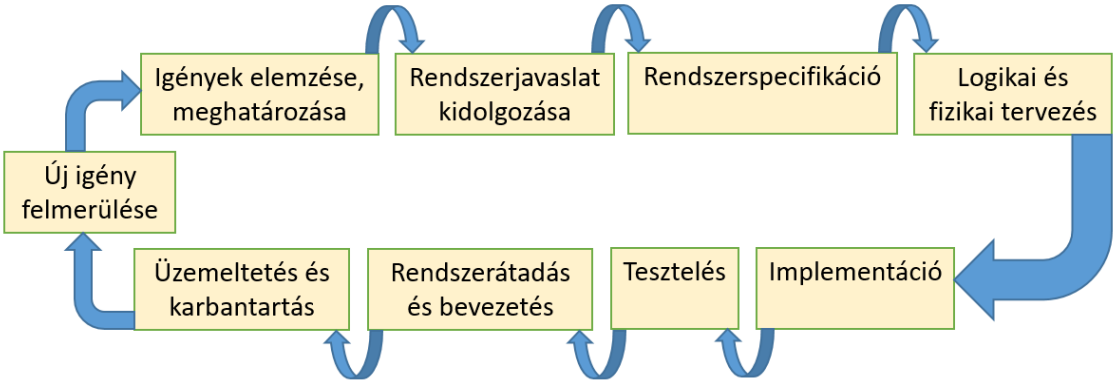
A branch coverage a tesztelendő szoftverben lévő minden lehetséges döntéshez tartozó ágat megvizsgálja. A mérőszám egy százalékos érték, amely a lefutott ágak számát hasonlítja a kód összes ágával.

Az MC/DC (Modified condition/decision coverage) megköveteli, hogy az egyes állapotok összes lehetséges állapotát teszteljék, miközben a többi állapotot rögzítették. Ezenkívül be kell mutatni, hogy az egyéni állapot változása megváltoztatja az eredményt. A teljes MC/DC fedettséghez szükséges a következő feltételek betartása:

- A program minden kilépési és belépési pontját legalább egy tesztesetben végre kell hajtani.
- A teszteknek minden döntést meg kell vizsgálnia az összes lehetséges eredmény szempontjából.
- A döntésekben szereplő minden feltételt az összes lehetséges állapotra futtatnia kell.
- Minden egyes feltételről be kell mutatni, hogy az eredményt függetlenül változtatja.

4.2 Szoftverfejlesztési modellek

A szoftver életciklusa már az igény felmerülésével kezdődik. Minden egyes alkalommal amikor kiadunk egy kész szoftvert, akkor egy idő után újabb igényei lesznek a felhasználóknak, ami a szoftver további fejlesztését igényli. Tehát a program életszakaszai ciklikusan ismétlődnek. Elméletben egy hasznos program végtelen életciklusú, viszont ez a keretrendszerek elavulása miatt nem létezik. A szoftver és a környezet elöregszik. A ciklus egyes szakaszait az alábbi, 8. ábra szemlélteti.



8. ábra: Életciklus modell

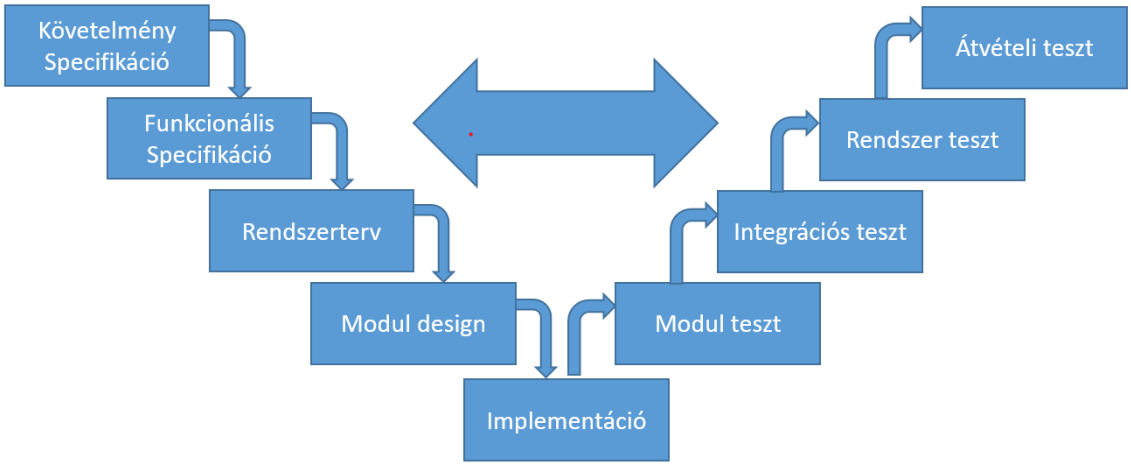
A fejlesztői munka szervezettebbé tételéhez különböző módszertanokat fejlesztettek ki, ezek a módszertanok meghatározzák, hogy a szoftver életciklusainak egyes lépései milyen sorrendben következzenek, valamint, hogy milyen

dokumentumokat, termékeket (szoftver) szükséges létrehozni és azok fejlesztésének mikéntjét.

Az első publikált szoftverfejlesztés folyamatát leíró modell, a Vízesés modell volt. A Vízesés modell egyszerű lépéseket kínál és jól elhatárolható fázisokra bontható. Fázisai nem működhetnek párhuzamosan, csak akkor indulhat a következő, ha a korábbi már befejeződött. Ez egy kezdetleges modell, amely rugalmatlan és már a korai szakaszokban komoly döntéseket igényel.

4.2.1 V-modell

Az autóiparban legtöbbször feltűnő modell a V-modell. Nevét a V betűt formáló szárairól kapta. Két szára a tervezési és a tesztelői részt testesítik meg. A már korábban megjelent vízesés modell alkotja az egyik szárat, amit kiegészít a tesztelési ág, azaz a szoftver életciklusának validációs és verifikációs szakasza. A modell azonos szintjén lévő szakaszok az összetartozó specifikáció és tesztelési fázisokat mutatják. (9. ábra)



9. ábra: V-model

Első lépésben a felhasználók igényeinek a kiderítése a feladat és megtudni, hogy milyen funkciókat szeretnének és mik az elvárásaik. Amennyiben ezek megvannak, az igényeket követelményekké kell formázni, majd ezeket a funkcionalitásra és a minőségre vonatkozó követelményeket össze kell gyűjteni és egy priorizált listába kell szedni. Ezt a listát a verifikálási szakaszban megvizsgálja az adott fejlesztő, hogy teljesültek-e.

Ezek után a felhasználói követelmények technikai követelménnyé alakítása a feladat. Meg kell határozni a külső és belső interfészek követelményeit, valamint azt, hogy mely komponenshez mely követelmények tartoznak.

A követelmény specifikáció befejeztével a különböző forgatókönyvek, koncepciók létrehozása következik. Definiálni kell a funkcionalitást pl.: melyik gomb milyen hatást vált ki. A meglévő követelményeket analizálni, klasszifikálni kell, továbbá be kell sorolni szükséges és elégséges feltételek szerint, valamint validálni kell őket szimulációval vagy prototípussal.

Mivel egy teljes rendszernek több száz vagy több ezer különböző követelménynek kell megfelelnie, ezért a követelmények felosztása után komoly menedzsment feladatokra van szükség, hogy a projektek, követelmények és a tervek között kapcsolat legyen. A menedzsment felelősöket rendel a követelmények mellé, valamint figyelemmel kíséri a követelmény változásokat.

Ha mindezek megtörténtek, meg kell tervezni, hogy az egyes funkciókat milyen metódusokkal, osztályokkal, adatbázisokkal kell megvalósítani. Definiálni kell, hogy ezek a funkciók mely komponensen belül helyezkednek el, valamint meghatározásra kerül, hogy a komponensek miként dolgozzanak együtt. Ezek a lépések és megállapítások alkotják a későbbi unit és integrációs tesztek alapját.

Az implementáció elvégzése után a komponenstesztek következnek, amelyek megvizsgálják a főbb funkciókat és osztályokat a uniton belül, majd integrálják a komponenseket, azaz megvizsgálják az együttes viselkedésüket. Hiba esetén az implementációban vagy a rendszertervben keresik a hiba forrását és javítják azokat.

Az integrációs tesztelést a funkcionális specifikációkon alapuló rendszerteszt követi. Ha a rendszertesztben hibát találunk, akkor visszatérünk a funkcionális specifikáció elkészítéséhez. Ezt követően elvégezzük az átvételi tesztelést a követelményspecifikáció alapján. Remélhetőleg nem lesz több hiba, máskülönben újrakezdhetjük, ami egy félresikerült projektnek felel meg.[5]

A V-modell összességében egy rugalmatlan, a változásokhoz nehezen alkalmazkodó modell, amely egy lineáris folyamatot vázol fel a szoftver fejlesztés folyamatáról.

4.2.2 Agilis szoftverfejlesztés

Maga az agile szó fürgeséget jelent, amely a módszertanok esetén a változásokra való gyors reagálást fejez ki. Agilis fejlesztés esetén, módszertanok egy csoportjáról beszélhetünk, amelyeket 2001-ben foglalt össze az Agile Manifesto kiadvány. Az agilis fejlesztésekben résztvevők rendkívül adaptívnak mondhatók, mivel a lehető legjobban

igyekeznek alkalmazkodni az adott projekthez. Ennek következtében a fejlesztők folyamatos továbbképzése, tanulása fontos szerepet játszik a folyamatok során.

Az agile 12 alapelve:[12]

1. A vevő kielégítése minél előbbi és folyamatosan érkező hasznos szoftverrel.
2. A nagy munka feldarabolása kis feladatokra, amelyek gyorsan elvégezhetőek.
3. Felismerni, hogy a legjobb munka az önszerveződő csapatokból születik.
4. Biztosítani a motivált egyéneknek azt a támogatást, amelyre szükségük van, valamint megbízni bennük, hogy a munkát valóban elvégzik.
5. Olyan folyamatok létrehozása, amelyek fentartható munkavégzést biztosítanak.
6. A követelmények változásának lekezelése, akár a projekt végső szakaszában is.
7. A műszaki kiválóságra és a jó dizájnra való folyamatos figyelem növeli az agilitást.
8. Rendszeres időközönként a csapat belső hatékonyságának a vizsgálata.
9. Rendszeres együttműködés a megrendelő és a kivitelező között.
10. A leghatékonyabb és legeredményesebb módszer az információ továbbítására a fejlesztőcsapatnak és azon belül a személyes beszélgetés.
11. A működő szoftver a fejlődés elsődleges mércéje.
12. Az egyszerűség elengedhetetlen.

Az előbb említett egyszerűbb szoftverfejlesztési modellekhez képest az agilis módszertan esetén a legszembetűnőbb különbség, hogy a fókuszban a változásra való reagálás áll. A klasszikus fejlesztési folyamatok elkerülték a módosításokat, mert extra kiadásokkal jártak, viszont az agilitás kiküszöböli ezt. Agilis fejlesztés esetén a projekteket felbontják kisebb részegységekre, amiket ciklikusan pl.: hétről- hétre készítenek el. Így nincs hosszadalmas tervezési fázis, mint az előzőleg már említett V-modell esetén, hanem apró tervezések történnek a ciklusok elején. Ez azért volt nagy előrelépés, mivel vagy a tervezés során kerül hiba a rendszerbe, vagy az ügyfél igényei változnak.

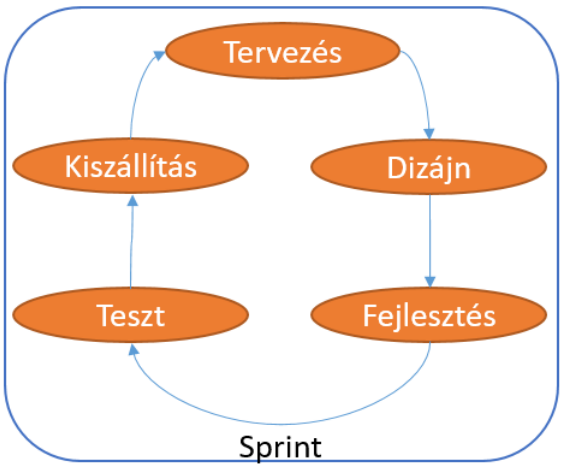
Agilis szoftverfejlesztés esetén az egyén és az interakció fontosabb, mint a folyamatok és az eszközök, mivel az emberek irányítják a folyamatokat és reagálnak az

üzleti változásokra és szükségletekre. Az agilis módszertan rámutat arra is, hogy az aktualitás sokkal fontosabb, mint a folyamatok betartása. Ha a folyamatok és az eszközök irányítják a projekteket, akkor az fejlesztő csapat kevésbé tud adaptálódni a változásokhoz, ezáltal kevésbé fog találkozni az eredmény a vevő elvárásaival.

A működő program nagyobb jelentőséggel bír, mint a dokumentációja. Az agilis fejlesztés előtt hatalmas mennyiségű időt öltek a termék dokumentációjának elkészítésébe a fejlesztés alatt, viszont az agile esetében a dokumentációs folyamat leegyszerűsödik.

A szerződéses tárgyalások helyett előtérbe kerül a vevő bevonása a fejlesztési folyamatokba. Az agile a menedzserek és az ügyfél közötti együttműködésre összpontosít, nem pedig a tárgyalásokra, hogy kidolgozzák a termék leszállításával kapcsolatos részleteket. A vevővel való együttműködés már nem csak a fejlesztés elején és végén történik meg, hanem az egész folyamat során, ezáltal is könnyebben illeszthető a termék a vevő igényeihez. A legjobb, ha a vevő minél előbb lát egy működő szoftvert, ami alapján egyeztethető az elképzelés és a valóság.

Az agilis fejlesztések hatékonyak, amíg a fejlesztő csoport létszáma 5-9 fő körül mozog, viszont nagyobb csoportok esetén nehezebb a csapat egységének megtartása. Az együttműködés és a kommunikáció ugyanolyan fontossá vált, mint a technológia, és mivel az Agile Manifesto nyitott az értelmezésekre, az Agile-t minden méretű és típusú szervezethez igazították és módosították. Bár az Agile megnyitja a kommunikációs vonalakat a fejlesztők és az üzleti oldal között, kevésbé sikerült a tesztelést és a folyamatokat ebbe a kompozícióba bevinni.



10. ábra: Agilis szoftverfejlesztés folyamata

4.3 Unit tesztek

A rendszer legkisebb önálló építőelemei a komponensek, melyek tesztelésére használjuk az unit teszteket. Egy teszt során a unit függvényeit, API-jait (Application Programming Interface) hajtjuk meg különböző bemenetekkel, paraméterekkel, amelyek megfelelő eredményt vagy hibát produkálnak. A unitokat egymástól teljesen függetlenül, izoláltan szükséges tesztelni.

Unit teszt által a hibák sokkal korábban észlelhetőek és ezáltal a fejlesztők biztosak lehetnek benne, hogy minden komponenst alaposan leteszteltek. Mivel a komponenseket szeparáltan tesztelik, ezért a hibák lokalizációja kevésbé bonyolult feladat és javításuk is egyszerűbb. Minden unithoz készül külön dokumentáció, amely példákat biztosít a program funkcióinak rendeltetésszerű használatára.

Egy tesztnek gyorsnak, függetlennek, megismételhetőnek, önellenőrzőnek és automatikusnak kell lennie. A gyorsaság a tesztek gyakori futtatása miatt szükséges, mivel, ha minél hosszabb ideig fut, annál kevesebbszer fogják elindítani a tesztet és annál tovább lappanghatnak a hibák a rendszerben. Minden tesztesetnek bármilyen sorrendben hívhatónak és a többi tesztesettől függetlennek kell lennie, máskülönben egy korábbi teszt hatása egy teljesen új és sztochasztikus szcenáriót hozhat létre.[4]

Az autóiparban az AUTOSAR BSW modulok C programnyelven íródnak az erőforráshatékonyság és az alacsony költségek miatt. Ezeknek a C programoknak a helyességét, valamilyen tesztelést támogató könyvtár segítségével, esetemben CUnit tesztekkel lehet ellenőrizni.

4.3.1 CUnit tesztek

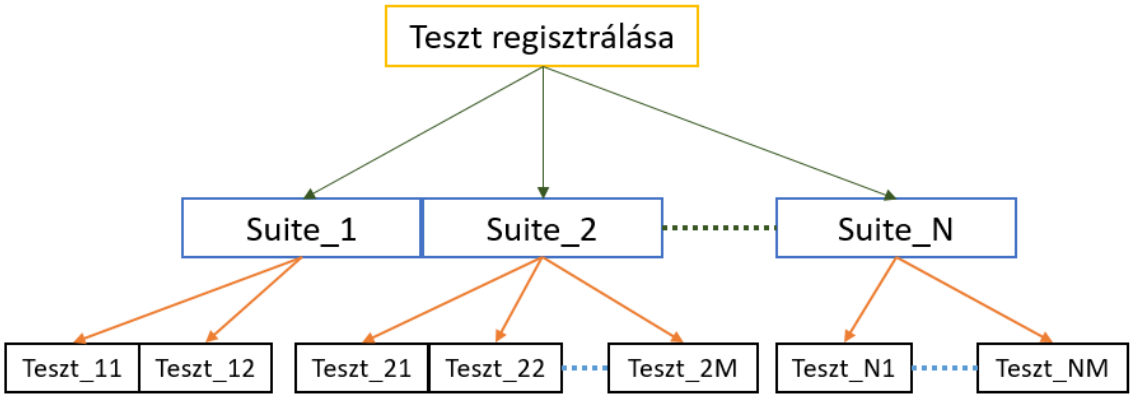
A CUnit egy rendszer egységtesztek írására, adminisztrálására és futtatására C nyelven. A C programozók számára alapvető tesztelési funkciókat biztosít rugalmas felhasználói felület széles választékával.

A CUnit statikus könyvtárként épül fel, amely a felhasználó tesztkódjához kapcsolódik. A CUnit egyszerű keretrendszert használ a tesztszerkezetek felépítéséhez, és ASSERT feltételek gazdag készletét kínálja a gyakori adattípussal rendelkező változók teszteléséhez. Ezenkívül számos különböző interfész biztosított a tesztek futtatásához és az eredmények jelentéséhez. Ezek közé tartoznak a kódvezérelt teszteléshez és jelentéskészítéshez szükséges automatizált interfészek, amelyek lehetővé teszik a felhasználó számára a tesztek futtatását és az eredmények dinamikus megtekintését.[11]

A felhasználói felületek megkönnyítik a keretrendszerrel való interakciót a tesztek futtatásához és az eredmények megtekintéséhez.

A CUnit egy platformfüggetlen keretrendszer kombinációja különféle felhasználói felületekkel. Az alapkeretrendszer támogatást nyújt a tesztnyilvántartás, a csomagok és a tesztesetek kezeléséhez.

A CUnit egy hagyományos egységtesztelési keretrendszerhez hasonlóan van felszerelve:



11. ábra: A tesztek felépítése

Az egyes tesztesetek suitokba vannak csomagolva, amelyeket az aktív tesztnyilvántartásban regisztrálnak. (11. ábra) A suitok rendelkezhetnek setup és teardown függvényekkel, amelyek automatikusan meghívásra kerülnek a suitok végrehajtása előtt és után. Ezek a függvények segítenek az előkészítő és lezáró műveletek végrehajtásában a tesztek előtt és után. A setup függvényt a tesztelendő egység előkészítésére és inicializálására használják. Ez a függvény fut le a teszt előtt, amikor beállítja a tesztelendő egység környezetét, például inicializálja változókat, létrehoz objektumokat vagy konfigurálja a rendszert. A setup függvény biztosítja, hogy a tesztelendő egység megfelelő állapotban legyen a teszt futtatása előtt. A teardown függvényt a teszt utáni takarítási feladatok végrehajtására használják. Ez a függvény fut le a teszt után, és felelős a teszt során létrehozott objektumok, erőforrások felszabadításáért vagy más takarítási feladatokért. A rendszerleíró adatbázisban található összes suit és teszt futtatható egyetlen függvényhívással, de a kiválasztott suitok vagy tesztek külön-külön is futtathatóak.

4.3.2 Generátor tesztek

A modulok fontos részei a konfiguráció függő generált fájlok. Ezeket a fájlokat az én esetemben egy modellezőeszköz és egy a modulhoz specifikusan hozzátartozó generátor program készítette el. Ahogy a modul C kódját, úgy a generátorát is vizsgálni kell helyes és helytelen bemenetekkel.

A generátorvizsgálat fő célja a generátorban jelen lévő eltérések azonosítása a modulkövetelmény dokumentum tekintetében. A generátor végrehajt a modellen konzisztencia ellenőrzéseket, amely ellenőrzéseket érvényes és érvénytelen bemenetekkel tesztelünk. Érvényes bemeneteket el kell fogadnia a generátornak, viszont az érvénytelen bemenetek esetén informatív hibaüzenetet kell adjon.

Tehát a konzisztencia ellenőrzését intenzíven teszteljük érvénytelen bemenetek használatával, ami segít elkerülni a kellemetlen kivételeket, a sérült generált fájlokat stb. az érvénytelen bemeneti konfiguráció miatt.

5 A CAN TRANSPORT LAYER MODUL TESZTELÉSE

5.1 Követelmények kezelése

Az AUTOSAR BSW modulok funkcionalitását a szabvány határozza meg, így a legtöbb követelmény elég jól meghatározott formátumban érhető el az AUTOSAR konzorcium honlapján. Ennek a megközelítésnek a következtében az alapszoftverhalmazzal kapcsolatos követelménykezelési stratégia az alábbiak szerint foglalható össze.

Az egyes követelmények eredetét a nyomon követhetőség javítása érdekében kezelni és tárolni kell. Alapértelmezés szerint minden szabványos dokumentumban meghatározott követelményt be kell vezetni és tesztelni kell.

Mivel lehetnek ellentmondások, gyengén definiált követelmények és az alkalmazási terület szempontjából irrelevánsok is, ezért minden követelménynél világosan meg kell jelölni, hogy elfogadjuk-e vagy elutasítjuk. Az alapértelmezett döntés az elfogadás, minden más döntést megfelelő indoklással kell alátámasztani.

Mivel egy adott elfogadott követelmény túl általános lehet ahhoz, hogy közvetlenül végrehajtható vagy tesztelhető legyen, azaz konkrét kódsorhoz lehessen rendelni, minden egyes követelménynél világosan meg kell jelölni, hogy megvalósítható-e vagy sem. Ugyanígy, szükséges megállapítani a tesztelhetőséget (tesztelhető-e vagy sem). Az alapértelmezett döntés szerint, végrehajtható és tesztelhető az adott követelmény, minden más döntést megfelelő indoklással kell alátámasztani.

A modul fejlesztése során már létrehoztak egy XML sémadefiníciót, amely tartalmazza a modulhoz tartozó összes követelményt. A modulnak egyaránt az AUTOSAR konzorcium által meghatározott, valamint az ISO15765-ös szabvány által megfogalmazott viselkedésnek is meg kell felelnie.

5.1.1 Követelmények osztályozása

Minden egyes követelményhez a BSW szabványból tartozik egy req példány a reqlist.xml-ben.

Minden elem rendelkezik egy eredeti azonosítóval (alapértelmezett), amely a megvalósítás alapjául szolgál az AUTOSAR SWS-ben meghatározottak szerint. Továbbá minden elem tartalmaz egy forrás-referenciát is, amely az adott követelmény származását jelöli, például AUTOSAR SWS, vagy SRS szabványt.

Minden követelmény öt attribútummal rendelkezik. Ezek az "acceptance", "id", "implementable", "testable" és "sourceRef" szimbólumok. Ezeket a modulfejlesztő már kitöltötte, viszont ezeket nekem felül kellett vizsgálni tesztelhetőség szempontjából.[1]

Az elfogadott (accepted) követelményeket megvizsgáltam tesztelhetőség szempontjából. Összesen 432 követelmény tartozik a modulhoz, ebből 236 implementálható és 180 tesztelhető.

A nem tesztelhető (12. ábra) követelmények valamilyen belső állapot változásra vagy valamilyen kívülről (black-box teszttel) direkt vagy indirekt módon nem tesztelhető viselkedésre utaltak. Ilyen például a fájl struktúrára vagy belső folyamatra vonatkozó követelmény.

[SWS_CanTp_00221] ⌈CanTp.c shall include CanTp_Cfg.h. ⌋ ( )

12. ábra: Elfogadott, implementálható, nem tesztelhető követelmény[16]

A fentebb látható követelmény egy belső folyamatra utal, tehát nem tesztelhető, viszont implementálható. A követelmények vizsgálatakor találtam olyan követelményt, amely korábbi verziókból hibásan lett átemelve, ezért ennek a felülvizsgálatát kértem. (13. ábra)

[SWS_CanTp_00269] ⌈After reception of each Consecutive Frame the CanTp shall call the `PduR_CanTpCopyRxData()` function with a PduInfo pointer containing data buffer and 6 or 7 data length (or less in case of the last CF). The output pointer parameter provides CanTp with available Rx buffer size after data have been copied.

13. ábra: Módosítással elfogadott,implementálható és tesztelhető követelmény

After reception of each Consecutive Frame the CanTp module shall call the PduR_CanTpCopyRxData() function with a PduInfo pointer containing data buffer and data length:
- 6 or 7 bytes or less in case of the last CF for CAN 2.0 frames
- DLC-1 or DLC-2 bytes for CAN FD frames (see Figure 5 and SWS_CanTp_00351).
The output pointer parameter provides CanTp with available Rx buffer size after data have been copied.

14. ábra: A SWS_CanTp_00269 módosítása

A fent említett követelményt módosítottuk a következő indoklással; "Ez a követelmény módosult, mert a 6 vagy 7 adathossz nem megfelelő CAN-FD támogatás esetén (a CAN-FD keretek lehetnek nagyobbak, mint 8 bájt). Ezen kívül, az AUTOSAR szabvány pontosan ezt a korrekciót tartalmazza a 4.3.1 verziótól kezdődően."

Minden tesztelhető követelményt behivatkoztam a tesztelés helyén, minél specifikusabban hozzárendelve a megfelelő kódrészlethez.

[SWS_CanTp_00272] ⌈The API `PduR_CanTpCopyTxData()` contains a parameter used for the recovery mechanism – 'retry'. Because ISO 15765-2 does not support such a mechanism, the CAN Transport Layer does not implement any kind of recovery. Thus, the parameter is always set to NULL pointer.⌋ ( )

15. ábra:Követelmény a retry paraméter értékéről

Például a SWS_CanTp_00272-es követelmény (15. ábra) a PduR_CanTpCopyTxData hívásakor beállított paraméter értékéről ír, aminek az ellenőrzését a PduR_CanTpCopyTxData stub függvényének a törzsében végeztem el a CUnit teszt könyvtár egy makrójának a segítségével. (16. ábra)

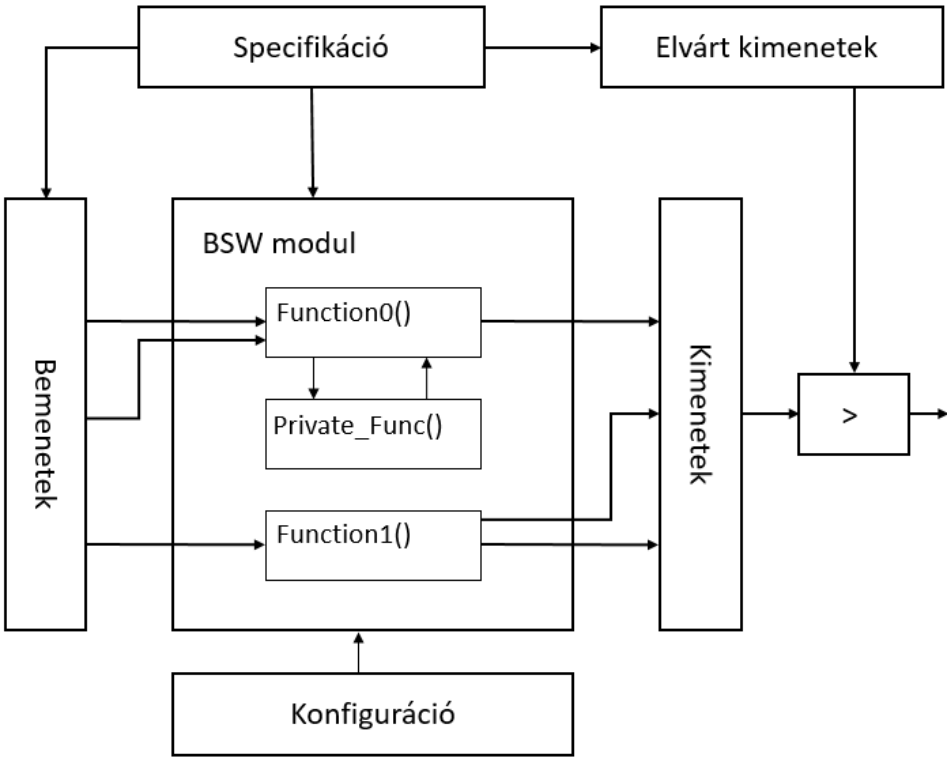
```
/*
 *  The API PduR_CanTpCopyTxData() contains a parameter used for the recovery mechanism - 'retry'.
 *  Because ISO 15765-2 does not support such a mechanism, the CAN Transport Layer does not implement
 *  any kind of recovery. Thus, the parameter is always set to NULL pointer.
 *  @reqtest{SWS_CanTp_00272}
 */
CU_ASSERT_PTR_NULL_FATAL(retry);
```

16. ábra: SWS_CanTp_00272

5.2 Tesztkörnyezet felépítése

A black-box teszteléshez szükséges egy tesztkörnyezet, ami megfelelően reagál a tesztelendő modul viselkedésére és képes naplózni a környező modulokhoz tartozó API hívásokat. Valamint a környezet segítségével a konfigurációfüggő viselkedést is vizsgálni lehet.

Tesztelés során a specifikáció alapján megkapja a modul a megválasztott bemeneteket, valamint a konfigurációt. A specifikáció alapján ismerjük az adott bemenetekhez tartozó kimenetet, így a megfelelő be- és kimenetek összehasonlításával megbizonyosodhatunk a modul helyes működéséről. (17. ábra)



17. ábra: Egy modul tesztelési struktúrája

A környező modulok API függvényeit, amelyeket a CanTp hívna, stub függvényekként hoztam létre, amely tartalmazza az adott hívás adatainak naplózásához szükséges kódot. Stub függvényként vettem fel Det, CanIf és PduR-hez tartozó függvényeket, valamint ezekhez vettem fel segéd függvényeket, amelyek a naplózást és a különböző interakciók beállításait hivatottak segíteni. Minden stub függvényhez tartozik egy külön forrásfájl a teszt átláthatóságának érdekében. Nem csupán fogadni kell az egyes API hívásokat, de naplózni is kell az egyes hívásokhoz tartozó argumentumok értékét és a szimulált interfésznek megfelelő választ és visszatérési értéket kell biztosítani a CanTp számára. A korábban már ismertetett CUnit tesztekkel vizsgálom a modul helyes működését, azaz ennek segítségével hozok létre különböző teszt suite-okat és eseteket a loggolt és az elvárt értékek összehasonlítására.

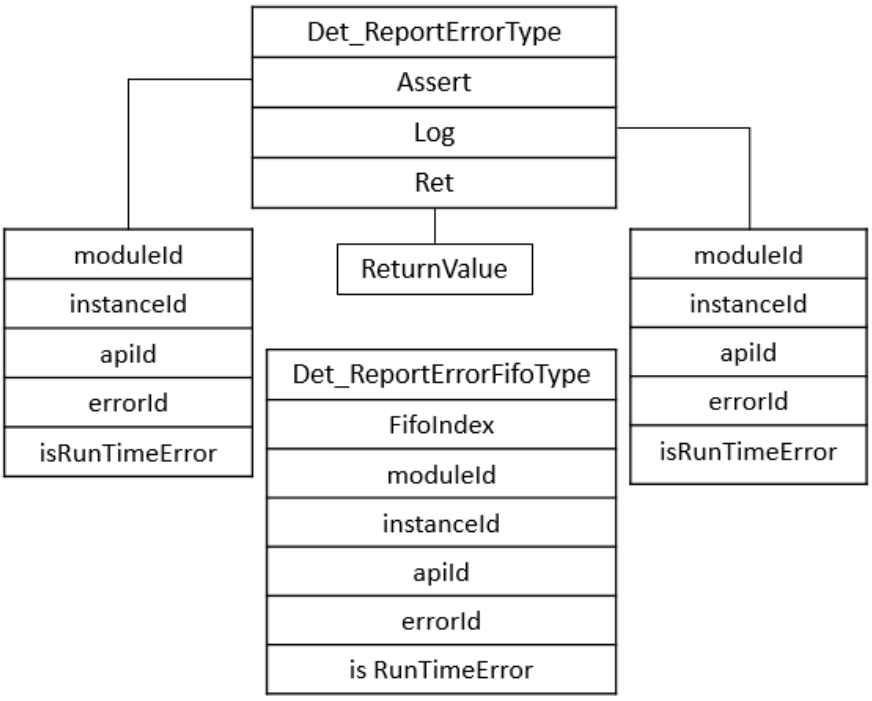
5.2.1 Tesztkörnyezet struktúrák

Minden hívott API függvény argumentumainak értékét naplózni kell és az egyes tesztesetek végén össze kell hasonlítani a kapott értékeket az elvártakkal. Ehhez el kellett készítenem a szükséges adatstruktúrát. Minden stub függvény argumentumának elmentéséhez statikus memória területeket foglaltam le az egyes tesztek elején. A Det riportokhoz a CanTp két különálló függvényt hívhat meg, egyet a futás idejű és egyet a

fejlesztés idejű hibák jelentésére. Ebben az eseteben azonos struktúrába naplóztam és mentettem az elvárt adatokat, viszont egy boolean változóba elmentettem az adott hívás típusát, azaz, hogy futás vagy fejlesztés idejű az adott hiba üzenet.

Később a tesztek során láthatjuk majd, hogy némely esetben az elvárt értékek hozzáadása késleltetést igényelt, ezért létrehoztam egyes esetekben egy FIFO struktúrát, amelybe elmenthetjük az elvárt értéket, és egy tetszőleges időpontban hozzáadhatjuk a kívánt értékek listájához, amelyeket a teszt figyelembe vesz a futás során.

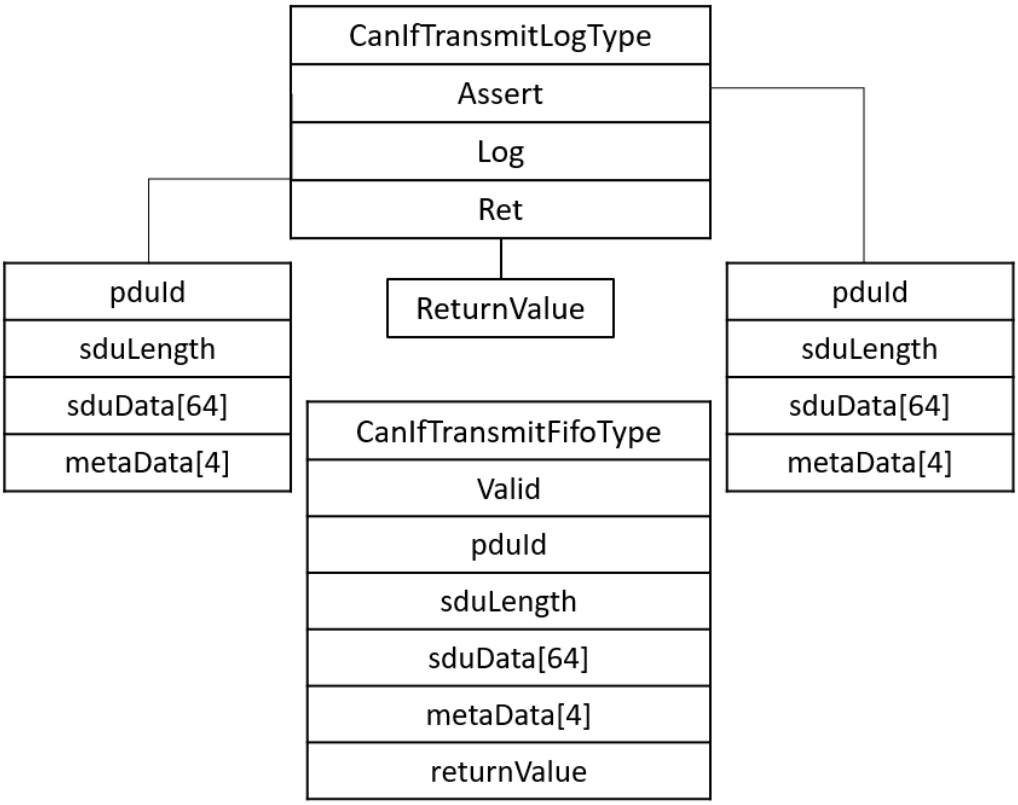
Egy stubbolt függvény mellé szükség van továbbá egy inicializáló függvényre, amely a teszteset elején alapállapotba állítja az elvárt és az aktuális változók értékét. Szükség van továbbá egy függvényre, amely segítségével a teszteset tetszőleges számú elvárt adatot képes hozzáadni az elvárt elemek listájához, és egy függvényre, ami a teszteset végén elvégzi az összehasonlítást az elvárt és a naplózott eredmények között. Alább láthatjuk a Det_ReportRuntimeError() és Det_ReportError() függvények kezeléséhez szükséges adatstruktúrát.(18. ábra)



18. ábra: Det API hívás naplózásához és asszertálásához szükséges adatstruktúra

Vizsgáljuk meg a CanIf-hez tartozó stub függvényeket is. (19. ábra) Hasonló módon épül fel a CanIf-hez tartozó adatstruktúra is, mint ahogy a Det esetén leírtam, viszont itt nagyrészt tömbök szerepelnek. Általánosságban az egyes sduData elemeknek egyaránt 64 bájtnyi memóriát foglaltam le, mivel ekkora a maximális kerethossz CAN-

FD esetén. A CanIf-nek továbbított metaData fixen, maximum 4 bájt hosszúságú lehet, ami ugyancsak a szabványból következik.

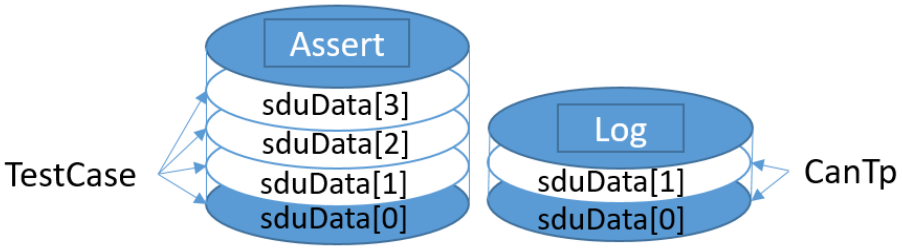


19. ábra: CanIf_Transmit API naplózásához és asszertálásához szükséges adatstruktúra

Hasonlóan ahogy a Det paramétereinél, itt nagy hangsúlyt kell fektetni az egyes keretek vizsgálatára, ugyanis a tesztnek pontosan el kell várnia, hogy melyik keret milyen tartalommal és milyen sorrendben érkezik. Ha egy keretet előbb vesz fel a teszt az elvárt üzenetek listájához, akkor annak a keretnek időben is előbb kell megérkeznie a CanTp-től. (20. ábra) A teszteset végén az elvárt elemek listája alapján és sorrendjében kerülnek összehasonlításra a keretek.

Az egyes tesztesetek végén CUnit függvényekkel asszertálom az értékeket, amely azt jelenti, hogy összehasonlításra a CU_ASSERT_EQUAL_FATAL() CUnit makrót használom fel. A komparálás a függvényhívások számának ellenőrzésével kezdődik. Amennyiben megegyezik az elvárttal, azaz ugyanannyiszor hívták meg az a CanIf_Transmit () függvényt, ahányszor a teszt elvárta, akkor folytatom a további értékek összehasonlítását. Ha nem, akkor megbukik a teszt és egy informatív üzenet segítségével jelzem a bukás helyét és okát. A teszt ettől függetlenül nem szakad meg és a többi stub

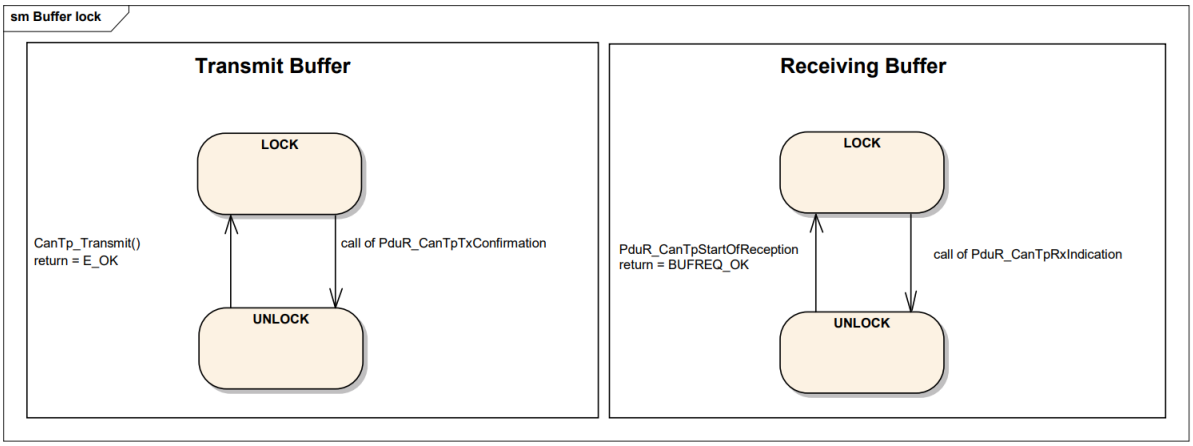
függvény esetén is végbemegy a komparálás. Ha a valós hívások száma megegyezik az elvárttal, akkor végig iterálva az összes híváson, összehasonlítom a PduId-t és az aktuális keret hosszt. Amennyiben ezekben eltérés észlelhető akkor a teszteset bukik és megszakad. Végül a payload és a metaData vizsgálata történik. A payload-on, a naplózott SDU hossz alapján iterálok végig, míg a MetaData fix hosszal rendelkezik, ezért minden bájtját megvizsgálom minden alkalommal.



20. ábra: Az elvárt és a naplózott keretek

A PduR-nek 5 API függvényét hívhatja a CanTp. Ezek felsorolás szinten az üzenetek fogadásához felhasznált API függvények a: PduR_CanTpStartOfReception, PduR_CanTpCopyRxData és PduR_CanTpRxIndication, valamint a küldéshez felhasznált API függvények a: PduR_CanTpCopyTxData és a PduR_CanTpTxConfirmation. A PduR_CanTpTxConfirmation és PduR_CanTpRxIndication esetén egyszerűen csak el kellett tárolni a PduId-t és a hozzá tartozó eredményt. Ebben a két függvényben jelzi a CanTp modul, hogy sikeres volt-e a küldés vagy a fogadás, valamint a buffer elengedését jelzik a PduR számára. Nem a CanTp kezeli a memóriát, amely az üzenetek küldéséhez vagy fogadásához szükséges. A felsőbb réteg szolgáltatja és zárolja a megfelelő memória buffert, amíg egy adatátvitel folyamatban van, és a CanTp csupán pointereken keresztül fér hozzá ezekhez a területekhez.

Küldés esetén a felsőbb réteg a CanTp_Transmit meghívása előtt már feltételezhetően zárolta a küldendő üzenetet tartalmazó buffert, míg fogadás esetén a PduR_CanTpStartOfReception szolgál a felsőbb réteg értesítésére. Ha van elegendő hely, akkor megkapja a CanTp a fogadásra képes zárolt memória buffert és elkezdődik a fogadás. (21. ábra)



21. ábra: Buffer zárolás és felszabadítás[16]

Minden további stubbolt API interfészt a CanIf-hez hasonlóan készítettem el, hogy minden bemeneti argumentumot megőrizzen és minden kimeneti argumentumot amire a CanTp-nek szüksége lehet szolgáltatni tudjon (visszatérési érték, bufferméret).

5.2.2 Teszt konfiguráció

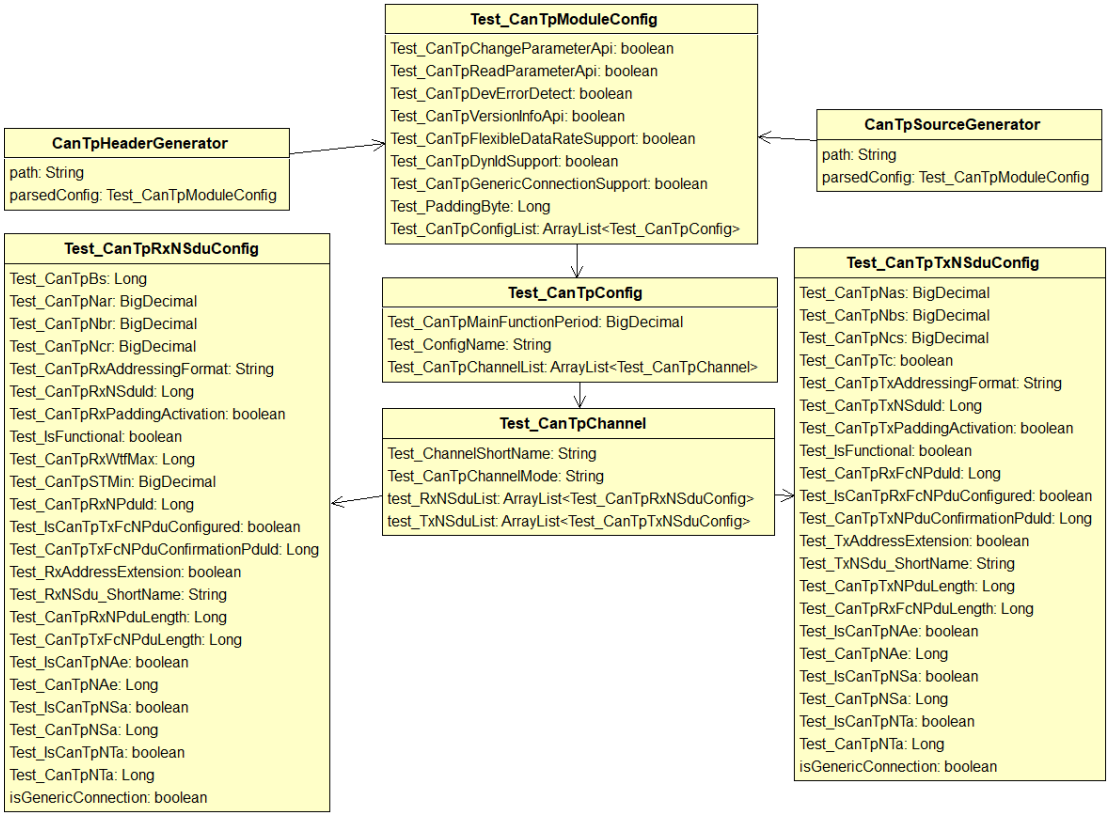
A konfigurációfüggő modulrészek letesztelésére egy modellezőeszköz állt rendelkezésemre, ami támogat tetszőleges számú különböző konfigurációt. Korábban már volt szó arról, hogy milyen széleskörűen konfigurálható a CanTp. A különböző beállítások hatására a CanTp működése is megváltozik, tehát a tesztnek is követnie kell ezeket a változásokat a megfelelő tesztesetek megvalósításához.

A konfiguráció eltárolására létrehoztam egy struktúrát, amely eltárolja az egyes paramétereket és biztosítja a megfelelő elérést a teszt számára. A struktúra nagyon egyszerű. Egy konfigurációhoz egy header és egy source fájlt hoztam létre. A precompile kapcsolókat egy header fájlba szerveztem, amelyek a CanTp szabványában megtalálható CanTpGeneral konténer alatt szereplő elemeket jelenti. Ilyen például a padding bájt értéke, az egyes API függvények kapcsolói stb. Ezen felül a CanTpConfig-ok számát is eltároltam itt, hiszen a modul képes több CanTpConfig kezelésére, tehát a tesztnek is képesnek kell lennie mindet letesztelni.

A CanTpConfig egy olyan konfigurációs adatszerkezet, amely meghatározza a modul működésének paramétereit. Ez a konfigurációs halmaz tartalmazza az AUTOSAR modulokban használt beállításokat, például az időzítést, az adatcsomagok formátumát, a prioritást, a kommunikációs interfészeket és egyéb paramétereket.

A CanTpConfig lehetővé teszi a modulok paramétereinek dinamikus beállítását futás közben anélkül, hogy a forráskódot módosítani kellene. Ezáltal lehetővé válik a modulok konfigurálása és tesztelése különböző környezetekben és alkalmazási esetekben anélkül, hogy a forráskódot újra kellene fordítani.

Fontos megjegyezni, hogy ugyan a modellezőeszköz a rendelkezésemre állt, viszont az értékek feldolgozása, szervezése és megfelelő generálása c és h fájlokba az én feladatom volt. A modellezőeszözhöz írtam egy generátort Java nyelven, ami segítségével az imént említett feladatokat elvégeztem. A Java kód implementációja a tesztelés szempontjából nem tölt be jelentős szerepet, ennek készítése közben is az egyszerűségre törekedtem. A tesztgenerátor elkészítése közben is igyekeztem minél magasabb minőségben dolgozni, ezért az Eclipse által biztosított SpotBugs analízist futtattam a kódra, amely nem eredményezett hibát. A CanTpConfig lekérdezéséhez használt Java kód osztálydiagramja a 22. ábra látható. A diagram az egyes műveleteket nem tartalmazza.



22. ábra: Tesztkonfig generátor osztály diagramja

Egy konfiguráció esetén el kell tárolni minden CanTpConfig esetén az összes csatornát és a hozzájuk tartozó küldő (Tx) és fogadó (Rx) oldali NSdu-kat. A Tx és Rx

oldal paraméterei hasonlóak, amint már korábban említettem, de nem ugyanolyanok, ezért a teszt adatstruktúrában nem különítettem el őket, viszont eltároltam egy boolean változóban, hogy az adott NSdu Tx oldali vagy sem. Rx oldal esetén azoknál a változóknál, ahol Tx oldali paramétert várt volna a teszt generátor, oda NaN értéket definiáltam, ami nulla értéket jelentett, viszont a generátumba írt érték sokkal beszédesebb lett. A Tx oldalnál ugyanígy jártam el, csupán az Rx oldali paraméterek nem kaptak értéket. Ahogy a 23. ábra is mutatja, felvettem mind az N_As és N_Ar időzítési paramétereket, viszont Tx esetben csak N_As(as) kap releváns értéket, N_Ar(ar) NaN értéket vesz fel.

23. ábra: Tesztkonfigurációs adatok modellje

5.3 Tesztesetek készítése

A modul egyes funkcióhoz teszt suitokat hoztam létre, ahol elsőként egy hibamentes átvitelt és fogadást teszteltem. Ezután a modul viselkedését vizsgáltam hibás bemenetek esetén, végül a különböző szolgáltatásokhoz kapcsolódó egyéb API függvények tesztelését végeztem.

5.3.1 Segédfüggvények

A korábban már említett üzenet típusok alapján egy átvitel lehet SingleFrame, vagy nagyobb mennyiségű adat esetén szegmentált kiküldés. Minden adatátvitel során a keretek tartalmaznak egy PCI (Protocol Control Information) mezőt, amely meghatározza, hogy az adott keret milyen típusú (SF, FF, CF, FC) és azt, hogy milyen paraméterekkel rendelkezik az adott keret vagy akár a teljes üzenet. A tesztelés során létrehoztam olyan segédfüggvényeket, amelyek képesek a megfelelő kereteket előállítani az üzenet hossza és a konfiguráció alapján.

SF esetén szükségem van az I-PDU-ra, azaz a nyers adatra, amit a CanTp a PduR modultól kap, valamint az N-PDU-ra, amit a CanTp kimenetén, a CanIf bemenetén várok, mint elküldendő adat. A segédfüggvényem a teszteset által biztosított payload hossz és a kiválasztott csatorna konfigurációs adatainak alapján eldönti, hogy lehetséges-e egy SF konstruálása, ezután összeállítja a szükséges N-PDU-t, valamint a nyers adatokat tartalmazó I-PDU-t. Küldés során az összehasonlítandó adatok a CanTp által kiküldött keretek lesznek, amelyeket a tesztkörnyezetben implementált CanIf bemenetén fogadok. A CanTp-nek, mikor az üzenetek fogadását tesztelem, a CanIf-től érkező N-PDU lesz a bemenete és a PduR bemenetén lévő hasznos adat lesz a kimenete, azaz az I-PDU.

Minden keret tartalmazza a már említett PCI mezőt. A segédfüggvényeknek, amelyek összeállítják a teszthez szükséges kereteket, ahogyan a modulnak is, be kell állítaniuk ezt a PCI mezőt. A szegmentált üzenetek kiküldésének tesztelésére először is szükségem volt egy függvényre, ami képes összeállítani a megfelelő FirstFrame-et és annak PCI mezőjét, valamint a fennmaradó szabad bájtokra elhelyezi az üzenet első bájtjait.

A küldéshez, a ConsecutiveFrame-ek esetén az I-PDU-k és az N-PDU-k előállítását két függvényben valósítottam meg. N-PDU esetben egy helyes átvitel teszteléséhez az átvitel utolsó CF-jét másként kell kezelni, mint a többit, ugyanis a SingleFrame-ekhez hasonlóan a konfigurációnak megfelelően lehet szó keret kitöltésről

(padding), vagy éppen optimalizálásról. Minden CF-nél megfelelően kell kitölteni a PCI mezőt, amely tartalmazza a keret típusát és sorszámát (SequenceNumber (SN)). Ezenfelül a megfelelő payload-dal kell feltölteni a kereteket. Átküldendő üzenetként egy 0-tól 63-ig tartó számsorozatot küldtem át ismétlődve, mint hasznos adat, ami segített a hibák felkutatásában a modulban, vagy akár teszt fejlesztése esetén is. A tesztből a segédfüggvény bemeneti paraméterként vár egy boolean értéket, amely eldönti, hogy az adott keretek egyből a stublog-ba, vagy egy köztes FIFO-ba kerülnek. A köztes FIFO segítségével az adatok késleltetve kerülhetnek bele a stublogba a teszt bármely pontján. Így legenerálhatjuk az összes keretet, de a teszt végén csak olyan adatokat várunk el, mint kimenet, vagy használunk fel, mint bemenet, amely ténylegesen bekerült a stublogba.

A SF-ek tesztelését tovább bontottam sikeres és sikertelen esetekre. Sikeres átvitel esetén is megkülönböztettem a klasszikus CAN és a CAN-FD-hez tartozó üzenetek kiküldését, mivel abban az esetben, ha a CAN kerethossza meghaladja a 8 bájtot, onnantól kezdve a SF-hez tartozó követelmények kibővülnek.

Minden API-hoz készítettem egy analízist, ahol elsőként megállapítom, hogy milyen bemenetek és kimenetek szerepelnek a tesztek során, amelyek alapján felállítom az ekvivalencia osztályaim. Ezek után az egyes ekvivalenciaosztályoknál határérték analízist végeztem és megállapítottam, hogy milyen bemeneti paraméter értékekre érdemes teszteket futtatni.

Minden egyes továbbított PDU esetén össze kellett állítanom a konfiguráció alapján, hogy milyen címzési paramétereket tartalmazzon az adatmező, valamint, hogy milyen értékek kerüljenek a MetaData-ba, amely alapján a CanIf összeállítja a CAN ID mezőt. Amennyiben a CanTpDynIdSupport engedélyezett a konfigurációban, úgy a CanTp-nek, tehát a teszteknek is fel kell töltenie a meta adatokat a konfigurációban meghatározott értékek alapján. Viszont, amennyiben a CanTpGenericConnectionSupport is aktív, a meta adatok a felsőbbrétegtől származó meta adatokból kerül összeállításra. A funkció teszteléséhez létrehoztam két segédfüggvényt, egyben az adatmezőbe bekerülő címzési paramétereket kezelem, a másikban pedig a CanTp-be beérkező és a CanTp-től elvárt meta adatokat állítom össze.

5.4 Transmit tesztesetek

A CanTp_Transmit függvényhez is készítettem egy külön teszt analízist. A CAN Transport Layer viselkedése ennek az API-nak a meghívásakor ekvivalencia osztályokra

oszthatók a következők alapján: a függvény argumentumai, a modul belső állapota, a modul konfigurációja és a környező modulok válaszai.

Bemenetek:

Függvényargumentumok:

- TxPduId: Az elküldendő PDU azonosítója
  - valid: Konfigurált érték egy csatornán.
  - invalid: Nem konfigurált érték.
- PduInfoPtr: Az SDU-t, a hosszt és a meta adatokat tartalmazza
  - MetaData: A PDU metaData mutatója.
  - SduLength: Az SDU mérete bájtban.
    - Teszt értékek: 1-től 66-ig, 140, 200, 500, 4096
  - SduData: Az értékét mindig érvényesnek tekintjük ezért az analízis során nem vizsgáljuk.

A modul belső állapota:

- CanTpState:
  - CANTP_OFF: A modul nincs inicializálva.
  - CANTP_ON: A modul inicializálva van.
  - TX_WAIT: A modul várakozik az átviteli kérelemre.
  - TX_PROCESSING: A CanTp inicializálva van, és a CanTp_Transmit sikeresen hívódott meg, de az üzenet még nem került teljes egészében átvitelre.

Konfigurációs paraméter:

- DevErrorDetect: konfigurációs kapcsoló
  - ON: A CanTpDevErrorDetect bekapcsolva.
  - OFF: A CanTpDevErrorDetect kikapcsolva.
- DynIdSupport: konfigurációs kapcsoló
  - ON: A CanTpDynIdSupport bekapcsolva.
  - OFF: A CanTpDynIdSupport kikapcsolva.
- CanTpTxTaType: Az addott TxNSdu-hoz tartozó kommunikációs típust tartalmazza.
  - CanTpPhysical (Phy): 1:1 kommunikációra használt.
  - CanTpFunctional (Fun): 1: n kommunikációra használt.

Kimenetek:

ReturnValue: Visszatérési érték

E_OK: Az átviteli kérelem elfogadásra került.

E_NOT_OK: Az átviteli kérelem nem került elfogadásra.

ErrorNotification: A modul egy hiba üzenetet generál a Det modul számára.

A fenti ekvivalencia osztály meghatározása szerint a következő határértékeket választottam a tesztesetek meghatározásához:

Függvényargumentumok:

- TxPduId: ennek a bemenetnek nincs intervallum szemantikája, minden érvényes és egy nem konfigurált értékre kell tesztelni.
- PduInfoPtr: ennek a bemenetnek nincs intervallum szemantikája, a NULL és egy nem NULL mutató vizsgálata szükséges.
  - MetaData: ennek a bemenetnek nincs intervallum szemantikája, a NULL és egy nem NULL mutató vizsgálata szükséges.
  - SduLength: ennek a bemenetnek nincs intervallum szemantikája, minden felsorolt lehetséges értéket meg kell vizsgálni.

A modul belső állapota:

- CanTpState: ennek a bemenetnek nincs intervallum szemantikája, minden felsorolt lehetséges értéket meg kell vizsgálni.

Konfigurációs paraméterek:

- DevErrorDetect: ennek a bemenetnek nincs intervallum szemantikája, minden felsorolt lehetséges értéket meg kell vizsgálni.
- DynIdSupport: ennek a bemenetnek nincs intervallum szemantikája, minden felsorolt lehetséges értéket meg kell vizsgálni.
- CanTpTxTaType: ennek a bemenetnek nincs intervallum szemantikája, minden felsorolt lehetséges értéket meg kell vizsgálni.

A fent azonosított ekvivalencia osztályok és a megfelelő határértékanalízis alapján az alábbi táblázat felsorolja a szükséges teszteket a CAN Transport Layer viselkedésének vizsgálatához ezzel a tsTransmission tesztkészlettel kapcsolatban.

A táblázat minden sora egy inger és az elvárt válasz párját mutatja be úgy, hogy az ingerek és a válaszok figyelembe veszik a modul belső állapotát is. A táblázat első oszlopa a megfelelő tesztet azonosítja. (24. ábra)

Test Case	CanTpState	TxPduId	PduInfoPtr	TxTaType	MetaData	SduLength	DynIdSupport	DevErrorDetect	Return Value	Error Notification
tcCanTpOff	CANTP_OFF	Don't care	Don't care	Don't care	Don't care	Don't care	Don't care	ON	E_NOT_OK	CANTP_E_UNINIT (dev)
tcCanTpOff	CANTP_OFF	Don't care	Don't care	Don't care	Don't care	Don't care	Don't care	OFF	E_NOT_OK	-
tcError_ Notification	TX_WAIT	Invalid	Set	Don't care	Don't care	Don't care	Don't care	ON	E_NOT_OK	CANTP_E_PARAM_ID (Dev)
tcError_ Notification	TX_WAIT	Valid	NULL_PTR	Don't care	-	Don't care	Don't care	ON	E_NOT_OK	CANTP_E_PARAM_POINTER (Dev)
tcTransmit_ MetaData_ NullPtr	TX_WAIT	Valid	Set	Don't care	NULL_PTR	SM	ON	ON	E_NOT_OK	CANTP_E_PARAM_POINTER (Dev)
tcTransmit_ MetaData_ NullPtr	TX_WAIT	Valid	Set	Don't care	NULL_PTR	SM	OFF	Don't care	E_OK	-
tcTransmitSF_ ShortMsg	TX_WAIT	Valid	Set	Don't care	Set	SF	Don't care	Don't care	E_OK	-
tcTransmit_ SegmentedMsg_ Long	TX_WAIT	Valid	Set	Physical	Set	SM	Don't care	Don't care	E_OK	-
tcTransmit_ SegmentedMsg	TX_WAIT	Valid	Set	Physical	Set	SM	Don't care	Don't care	E_OK	-
tcTransmit_ SegmentedMsg_ Long	TX_WAIT	Valid	Set	Functional	Set	SM	Don't care	Don't care	E_NOT_OK	CANTP_E_INVALID_TATYPE
tcTransmit_ SegmentedMsg	TX_WAIT	Valid	Set	Functional	Set	SM	Don't care	Don't care	E_NOT_OK	CANTP_E_INVALID_TATYPE
tcTransmitSF_ ShortMsgs_ parallel	TX_ PROCESSING	Valid	Set	Don't care	Set	SF	Don't care	Don't care	E_OK	-
tcError_ Notification	TX_WAIT	Valid	NULL_PTR	Don't care	-	Don't care	Don't care	ON	E_NOT_OK	CANTP_E_PARAM_POINTER (Dev)

24. ábra: A CanTp_Transmit API-hoz tartozó tesztesetek

Az üzenetek kiküldése során három időzítési paraméternek kell megfelelnie az átvitelnek, az N_As, N_Bs és N_Cs paramétereknek. Ehhez összesen 4 tesztesetet állítottam fel. Az időzítők vizsgálatához eljuttattam a modult a megfelelő állapotba és ott addig hívtam a modul main függvényét, ameddig az időzítő le nem járt. A main függvényt normál esetben az RTE hívja periodikusan a konfigurációban beállított időközönként. Mivel ismerem a periódus időt és az egyes időzítők értékeit, ezáltal pontosan ismerem, hogy hányszor kell meghívni a main függvényt, hogy az adott időzítő lejárjon. Ha egy időzítő lejár akkor a modulnak értesítenie kell a felsőbb réteget a sikertelen küldésről és error riportot kell küldenie a Det modulnak. Az időzítőket megvizsgáltam minden lehetséges szituációban, például az N_Cs esetén megvizsgáltam a működést a FlowControl és Consecutive Frame, valamint két Consecutive Frame között is. Az egyes időzítők elhelyezkedését a 6. ábra jól szemlélteti.

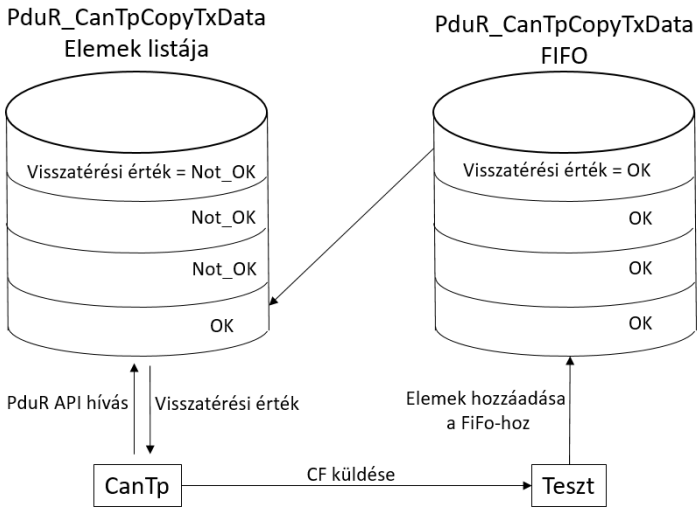
A küldés során a fogadó fél a FlowControl segítségével képes folyamszabályozásra, tehát teszteket készítettem, amiben a FlowStatus, STmin és BlockSize paraméterek változtatásának hatását ellenőriztem. A FlowStatus esetén

egyszerű dolgom volt, mivel annak függvényében megszakítja, várakozik, vagy küldi tovább az üzenetek a küldő, azaz a tesztelendő CanTp modul. Ennek következtében olyan teszteseteket készítettem, ahol a fogadó fél, azaz a tesztkörnyezet, az említett státuszokkal FlowControl üzeneteket kézbesít a küldő számára. Ezek után figyeltem, hogy folytatódik-e a küldés, hogy megszakítás esetén érkezik-e értesítés a PduR modul számára a sikertelen átvitelről, vagy várakozás esetén történik-e timeout.

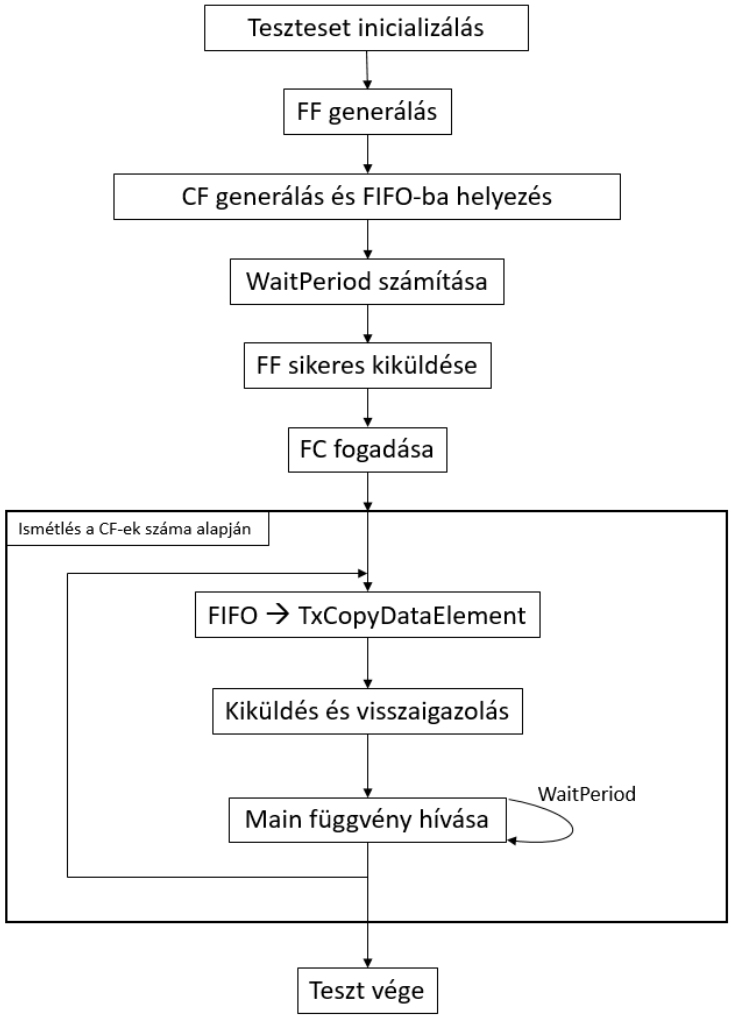
A BlockSize esetén a küldőnek a megadott méret alapján kell küldenie a kereteket és várnia a következő FlowControl-ra, ha elküldte a megfelelő mennyiségű keretet. Tehát ha a küldő nem kap FC üzenetet a megadott blokk méretenként akkor timeout keletkezik.

Az STmin paraméter teszteléséhez határérték analízist végeztem az érvényes és érvénytelen értékek határain, ami azt jelenti, hogy az értékhatárokon és a tartományok közepéről választottam bemeneti teszt értékeket. A konfigurációs tool-ban másodpercben lehet megadni az értékeket, ezért a teszt konfiguráció generálásakor átskáláztam minden időértéket mikroszekundumba, ezáltal nem volt szükséges a lebegőpontos változók használata a tesztben. Az STmin paraméter és a main függvény periódus idejének a hányadosa adja, hogy hányszor kell meghívni a main függvényt a CanTp_TxConfirmation hívása után, mielőtt az új keret kiküldésre kerülne. Nevezzük ezt az értéket WaitPeriod-nak. (26. ábra)

Ennél a tesztnél felhasználtam a stublog FIFO-t, ami nem egyből teszi az felhasználni kívánt keretek listájába a kívánt keretet, hanem közvetlenül az STmin lejárata után. Ugyebár a CanTp a működés során hívja a PduR-t amelynek egy API függvénye (PduR_CanTpCopyTxData) szolgáltatja a CanTp számára a kiküldendő adatokat. (25. ábra) Szegmentált átvitel során a modul két keret kiküldése között meghívja a PduR_CanTpCopyTxData függvényt amelynek visszatérési értéke a felsőbb réteg állapotát jelzi. Ez az érték lehet BUFREQ_OK, BUFREQ_E_BUSY, BUFREQ_E_NOT_OK vagy BUFREQ_E_OVFL. Alapvetően, ha meghívják a PduR_CanTpCopyTxData függvényt akkor az inicializált BUFREQ_E_NOT_OK visszatérési értékű elemet fogja elérni a bemenetek közül, ami megszakítaná az átvitelt. Ha helyesen működik a modul és kivárja a megfelelő STmin időtartamot, akkor a teszt közvetlenül mindig egy olyan elemet tesz be a PduR_ CanTpCopyTxData stub log elemei közé, aminek a visszatérési értéke BUFREQ_OK lesz. (25. ábra) Tehát az átvitel nem szakad meg a felsőbb réteg negatív válasza miatt és a teszt sikeresen átmegy.



25. ábra: Az elemek átkerülése a FIFO-ból a felhasznált elemek listájába



26. ábra: FlowControl STmin paraméterének tesztelése (folyamat ábra)

Leteszteltem, hogy küldés során Single, First vagy Consecutive Frame esetén mi történik, amennyiben az alsóbb réteg (CanIf) negatív visszatérési értéket ad. Ezek a tesztesetek elég hasonló felépítésűek. El kellett juttatni a modult a megfelelő állapotba. A stublogba előre felvettem, hogy milyen várható keretek érkeznek és milyen sorrendben, majd a megfelelő pillanatban negatív visszatérési értéket állítottam be az adott híváshoz. Amikor meghívja a CanTp a CanIf modul API-ját, akkor ezt a beállított értéket kapja vissza. Ugyanígy teszteltem a PduR minden lehetséges visszatérési értékére a viselkedést.

Létrehoztam tesztesetet, amelyben különböző hiba üzeneteket vártam a Det interfészén.

- Átviteli kérelmet indítottam párhuzamosan ugyanazon a csatornán.
- Null pointerrel hívtam meg a CanTp_Transmit függvényt.
- Invalid (nem konfigurált) azonosító értékkel indítottam átvitelt

Az üzenetek küldésével kapcsolatban elmondható, hogy minden teszt sikeresen lefut és a modul a követelményekben megfogalmazottak alapján működik. A teszt suite-ban a teszteseteket dokumentáltam, azaz leírtam a teszt működését és a vizsgált követelményeket. Az egyes esetek előtt doxygen kommentben rögzítettem az adott eset összegző leírását, valamint vázlatpontokban leírtam az eset felépítését. Ahol az egyes követelményeket teszteltem, ott kommentbe beírtam a teljes követelmény szövegét, vagy adott esetekben, ahol a követelmény több mindent magába foglalt, ott a releváns részletet. Minden követelményhez tartozik egy azonosító, ami alapján tag-ekkel hivatkozom a teljes követelményt a már említett xml fájlból.

Üzenetek kiküldéséhez szorosan kapcsolódóan, összesen 25 tesztesetet készítettem, amelyek minden konfiguráció esetén előállítanak különböző bemeneteket és a hozzájuk tartozó kimeneteket.

5.5 Reception tesztek

Egy üzenet fogadása során a CanTp modulnak jóval több dolga van, mint egy üzenet kiküldése esetén. A CanTp-nek el kell döntenie, hogy az adott üzenet érvényes és fogadhatja-e az adott pillanatban, ezt viszont rengeteg paraméter befolyásolja.

A CanTp_RxIndication API függvény röviden, az alsó szintű kommunikációs interfészmodulból hívható, amely fogadja a beérkezett keretet.

A CAN Transport Layer viselkedése ennek az API-nak a hívásakor az alábbiak szerint osztható fel ekvivalencia osztályokra: függvényargumentumok, a modul belső állapota, a környező modulok viselkedése és a modul konfigurációja alapján:

Bemenetek:

Függvényargumentumok:

- RxPduId: A fogadott PDU azonosítója.
  - valid: Konfigurált érték, mint CanTpRxNPduId egy csatornán.
  - invalid: Nem konfigurált érték.
- PduInfoPtr: Tartalmazza a fogadott PDU hosszát (SduLength), egy mutatót a PDU-t tartalmazó bufferre (SduDataPtr), valamint a MetaData-t ehhez a PDU-hoz.
  - SduLength: Az SDU hossza bájtban.
  - MetaData: A PDU meta adatának mutatója.
  - SduData: Az értékét mindig érvényesnek tekintjük ezért az analízis során nem vizsgáljuk.

A modul belső állapota:

- CanTpState:
  - CANTP_OFF: A modul nincs inicializálva.
  - CANTP_ON: A modul inicializálva van.
  - TX_PROCESSING: A CanTp inicializálva van, a CanTp_Transmit sikeresen meghívásra került és az átvitel folyamatban van.
  - RX_WAIT: A CanTp inicializálva van, de nincs fogadás folyamatban.
  - RX_PROCESSING: A konfigurált fogadó csatorna használatban van.

Konfigurációs paraméter:

- DevErrorDetect:
  - ON: A CanTpDevErrorDetect be van kapcsolva.
  - OFF: A CanTpDevErrorDetect ki van kapcsolva.
- ChannelMode:
  - FullDuplex: Full-duplex-csatorna.
  - HalfDuplex: Half-duplex-csatorna.

Felső réteg modulok válaszai:

- PduR_CanTpStartOfReception: Ha a fogadott keret elfogadásra került, akkor a felső réteg biztosítja a buffert.
  - Visszatérési érték: A felső réteg státusza.
  - BufferSize: A biztosított buffer mérete.

Kimenetek:

Látható viselkedés a felső rétegeken:

- PduR_CanTpRxIndication: A CanTp jelzi a felső rétegnek, hogy a fogadás befejeződött.
  - eredmény: A fogadás eredménye.
    - BUFREQ_OK, BUFREQ_E_NOT_OK, BUFREQ_E_OVFL
  - Hibaértesítés: A modul egy Det hibát jelent.

Határérték analízis

Az előzőleg meghatározott ekvivalencia osztályok alapján az alábbi határvizsgálati értékeket választottam ki a tesztesetek definiálásához:

Függvény argumentumok:

- RxPduId: ennek a bemenetnek nincs intervallum szemantikája, minden lehetséges érvényes és egy érvénytelen értéket meg kell vizsgálni.

- PduInfoPtr: ennek a bemenetnek nincs intervallum szemantikája, vizsgáljunk meg egy NULL és egy nem NULL pointerre is.
    - MetaData: ennek a bemenetnek nincs intervallum szemantikája, vizsgáljunk meg egy NULL és egy nem NULL pointer-t is.
    - SduLength: ennek a bemenetnek intervallum szemantikája van, az alábbi pontokat vizsgáljuk:
        - Minden lehetséges CAN-FD keret hosszúság értéket. (8, 12, 16, 20, 24, 32, 48, 64)
        - Érvénytelen: Nem lehetséges CAN-FD értékek: 0, 11, 13
    - SduData: Az értékét mindig érvényesnek tekintjük ezért az analízis során nem vizsgáljuk.

A modul belső állapota:

- CanTpState: ennek a bemenetnek nincs intervallum szemantikája, minden felsorolt lehetséges értéket meg kell vizsgálni.

Konfigurációs paraméter:

- DevErrorDetect: ennek a bemenetnek nincs intervallum szemantikája, minden felsorolt lehetséges értéket meg kell vizsgálni.

Az alsó réteg moduljainak válaszai:

PduR_CanTpStartOfReception

- ReturnValue: ennek a bemenetnek nincs intervallum szemantikája, minden felsorolt lehetséges értéket vizsgáljunk meg.
- BufferSize:
    - A fogadott keret payload hossza (I-PDU hossz).
    - A szegmentált üzenet hossza.
    - Kisebb, mint a fogadott keret payload hossza

Tesztek:

Az előzőleg azonosított ekvivalencia osztályok és a megfelelő határvizsgálati elemzés alapján az alábbi táblázatban felsoroljuk azokat a teszteket, amelyek szükségesek

a CAN Transport Layer viselkedésének vizsgálatához a tsReception tesztkészletre vonatkozóan.

Minden táblázat sor egy inger és a várható válasz párost tartalmazza úgy, hogy az inger és a válaszok figyelembe veszik a modul belső állapotát is. A táblázat első oszlopa megadott teszteset nevét tartalmazza. (27. ábra)

Test Case	CanTpState	RxPduId	PduInfoPtr	SduLength	MetaData	ErrorDetect	ChannelMode	StartOfReception ReturnValue	StartOfReception BufferSize	RxIndication Result	Error Notification
tcCanTpOff	CANTP_OFF	Don't care	Don't care	Don't care	Don't care	ON	Don't care	-	-	-	CANTP_E_UNINIT(dev)
tcCanTpOff	CANTP_OFF	Don't care	Don't care	Don't care	Don't care	OFF	Don't care	-	-	-	-
tcReceive_ InvalidIdSF	RX_WAIT	invalid	Set	Don't care	Don't care	ON	Don't care	-	-	-	CANTP_E_PARAM_ID
tcRxIndication_ NullPtr	RX_WAIT	valid	NULL_PTR	-	-	ON	Don't care	-	-	-	CANTP_E_PARAM_POINTER
tcReceive_ InvalidLen_ SF	RX_WAIT	valid	Set	Invalid	Set	Don't care	Don't care	-	-	-	-
tcReceive_ MetaData_ NullPtr	RX_WAIT	valid	Set	CANFD Value	NULL_PTR	Don't care	Don't care	BUFREQ_OK	SduLength	E_OK	-
tcUnexpectedPDU SFarrival_ DuringTransmission_ HalfDuplex	TX_ PROCESSING	valid	Set	CANFD Value	Set	ON	HalfDuplex	BUFREQ_OK	SduLength + 1	-	-
tcUnexpectedPDU SFarrival_DuringTrans mission_FullfDuplex	TX_ PROCESSING	valid	Set	CANFD Value	Set	ON	FullfDuplex	BUFREQ_OK	SduLength + 1	E_OK	-
tcReceiveSF_ BufReqNotOk	RX_WAIT	valid	Set	CANFD Value	Set	Don't care	Don't care	BUFREQ_E_ NOT_OK	Don't care	E_NOT_OK	-
tcReceiveSF_ BufReqNotOk	RX_WAIT	valid	Set	CANFD Value	Set	Don't care	Don't care	BUFREQ_E_ OVFL	Don't care	E_NOT_OK	-
tcReceiveSF_ BufReqNotOk	RX_WAIT	valid	Set	CANFD Value	Set	Don't care	Don't care	BUFREQ_OK	SduLength - 1	E_NOT_OK	-
tcReceiveSF_ BufReqOk	RX_WAIT	valid	Set	CANFD Value	Set	Don't care	Don't care	BUFREQ_OK	SduLength	E_OK	-
tcUnexpectedPDU SFArrival_ DuringReception	RX_ PROCESSING	valid	Set	CANFD Value	Set	ON	Don't care	BUFREQ_OK	SduLength + 1	E_OK	-

27. ábra: A CanTp_RxIndication API-hoz tartozó tesztesetek

A tesztelés során a fogadásnál, ahogyan a küldésnél is, el kell készíteni a bemeneteket és az elvárt kimeneteket. Miután a küldéshez már külön segédfüggvényeket készítettem az egyes Single Frame, First Frame stb. előállítására, ahol előállítottam mind az I-PDU-t és az N-PDU-t, így a fogadás vizsgálatához csak meg kellett cserélnem a bemenetet és az elvárt elemeket. A küldésnél az I-PDU szolgált bemenetként és a kimenetet hasonlítottam össze az általam legenerált N-PDU-val, míg fogadásnál az N-PDU a bemenet és az I-PDU kerül bele a stublogba az elvárt elemek listájához. Természetesen a modul más függvényeket hív felfelé és más API-n keresztül érhetjük el a CanTp-t is, de az argumentumok listája a függvényeknél nagyon hasonló, így a segédfüggvényeket ismét fel tudtam használni.

Hasonlóan, ahogyan a küldés vizsgálatának megkezdésekor, a fogadásnál is a Single Frame vizsgálatával foglalkoztam elsőként. Ezúttal is jó kiindulási pontnak bizonyult, hogy egy helyesen összeállított üzenetet képes-e lekezelni modul. Tehát az első tesztesetemben a Single Frame összeállító segédfüggvényemnek megadtam, hogy milyen

hosszú üzenetet szeretnék vizsgálni, majd az visszaadta az I- és N-PDU-kat. Az I-PDU-t megadtam a PduR interfészen, mint elvárt kimenet, az N-PDU-t pedig megadtam a CanTp_RxIndication argumentumaként, mint bemenet, azaz mint beérkező üzenet. Ezenfelül beállítottam, hogy a felsőbb réteg fogadja és biztosítson megfelelő buffer méretet az érkező üzenet számára. A teszthez sokrétű konfigurációkat állítottam össze, ahol különböző hosszúságú PDU-kat rendeltem a csatornákhoz, ki-be kapcsolatam a Padding paramétert, változtattam a címzési módot és minden lehetséges konfigurációs paramétert.

Ezután elkezdtem letesztelni a modul viselkedését a PduR től kapott visszatérési értéktől függően. Fogadásnál három PduR API függvényt hív meg a CanTp a folyamat alatt, egyel elindítja, egyel másolja a bejövő vagy kimenő adatokat, majd egy harmadikkal nyugtázza a folyamat sikerességét, avagy sikertelenségét a felsőbb réteg irányába. Amikor beérkezik egy helyes üzenet akkor a felsőbb rétegnek is nyilatkoznia kell, hogy képes-e a fogadásra és ha igen, akkor meg kell adnia az adott üzenethez biztosított buffer méretet is. A PduR státuszától függően a CanTp várakozik, abortálja vagy fogadja a beérkező üzenetet.

Ha a PduR kész a fogadásra, akkor a biztosított buffer méret alapján a CanTp még mindig visszautasíthatja a fogadást. Tehát teszteltem a modult különböző visszaadott státuszokra és méretekre. Ha a visszaadott buffer méret, kisebb, mint az aktuális keretben lévő hasznos adat, akkor a CanTp elutasítja a fogadást, viszont, ha nagyobb a biztosított méret, mint a beérkezett hasznos adat, de szegmentált átvitel során nem lenne elegendő a következő keret fogadására, akkor a modul elindítja az N_Br időzítőt és ha lejárt, akkor FlowControl üzenetet küld FC.WAIT státusszal és újra indítja az időzítőt.

A konfigurációban beállítható paraméter a WftMax, ami megadja, hogy hány darab FC küldhető FC.WAIT státusszal mielőtt a modulnak abortálnia kéne a fogadást. Minden egyes main függvényhíváskor a CanTp rákérdez a PduR-nél, hogy van-e már elég hely. Amennyiben megfelelő buffer méretet kap a felsőbb rétegtől, egy FC üzenetben engedélyezi a küldőnek az újabb keret kiküldését.

A tesztesetem eljuttatta a megfelelő állapotba a modult, tehát a modul megkapta a FF-et, majd nem szolgáltattam elegendő buffert további keretek eltárolására. A CanTp N_Br * WftMax-szor hívta meg a PduR API függvényét. Miután kiment az utolsó FC üzenet és az N_Br időzítő is lejárt, egy Det hiba értesítést várok a modultól. A tesztelés során pontosan tudom, hogy melyik időpillanatokban kell, hogy interakciót

kapjak a modultól az interfészeken, ezáltal meg tudok bizonyosodni a helyes működésről. Itt gyakorlatilag az időzítő és a WftMax szerinti helyes működést egyszerre tudtam tesztelni. Megvizsgáltam a viselkedést az említett FF utáni, valamint egy blokk utolsó CF utáni szituációjában is.

Megvizsgáltam az N_Ar és N_Cr időzítők helyes működését is. Az N_As, N_Cs időzítőkhöz hasonlóan eljuttattam a megfelelő állapotba a modult, majd meghívtam a main függvényt pontosan annyiszor, hogy elérje az időkorlátot, ahol egy Det error üzenetet várok, így ellenőrizve a helyes működést.

Egy szegmentált átvitel során a fogadó félnek nagyon sok mindent kell ellenőriznie a keretek helyességével kapcsolatban. Az üzenetek fogadásának tesztelésekor a FlowControl és FirstFrame generáló segédfüggvényeim sikeresen fel tudtam használni, viszont az üzenetekhez tartozó Consecutive Frame-ek generálását újra kellett alkotnom. Ezt nem szerveztem ki külön függvénybe, mivel egységében, csak egy teszteset használja, ahol a helyes üzeneteket megvizsgálom. Egyéb esetekben valamilyen hibát kellett juttatni valamely részébe a kódnak, ezért a kódot érdemesebb volt ismételten leírni, mint minden alkalommal egy külön függvényt létrehozni.

Be kell állítani minden keretnél a címzési információkat, valamint a ConsecutiveFrame-eknél a SequenceNumber számot is. Ezenfelül még az átvitt adatoknak is meg kell felelniük, amelyet továbbra is 0-tól 63-ig egy folytonos, ismétlődő számsorozat szolgáltat. Az utolsó keret esetén méret optimalizáció vagy a maradék bájtok kitöltése(padding) szükséges.

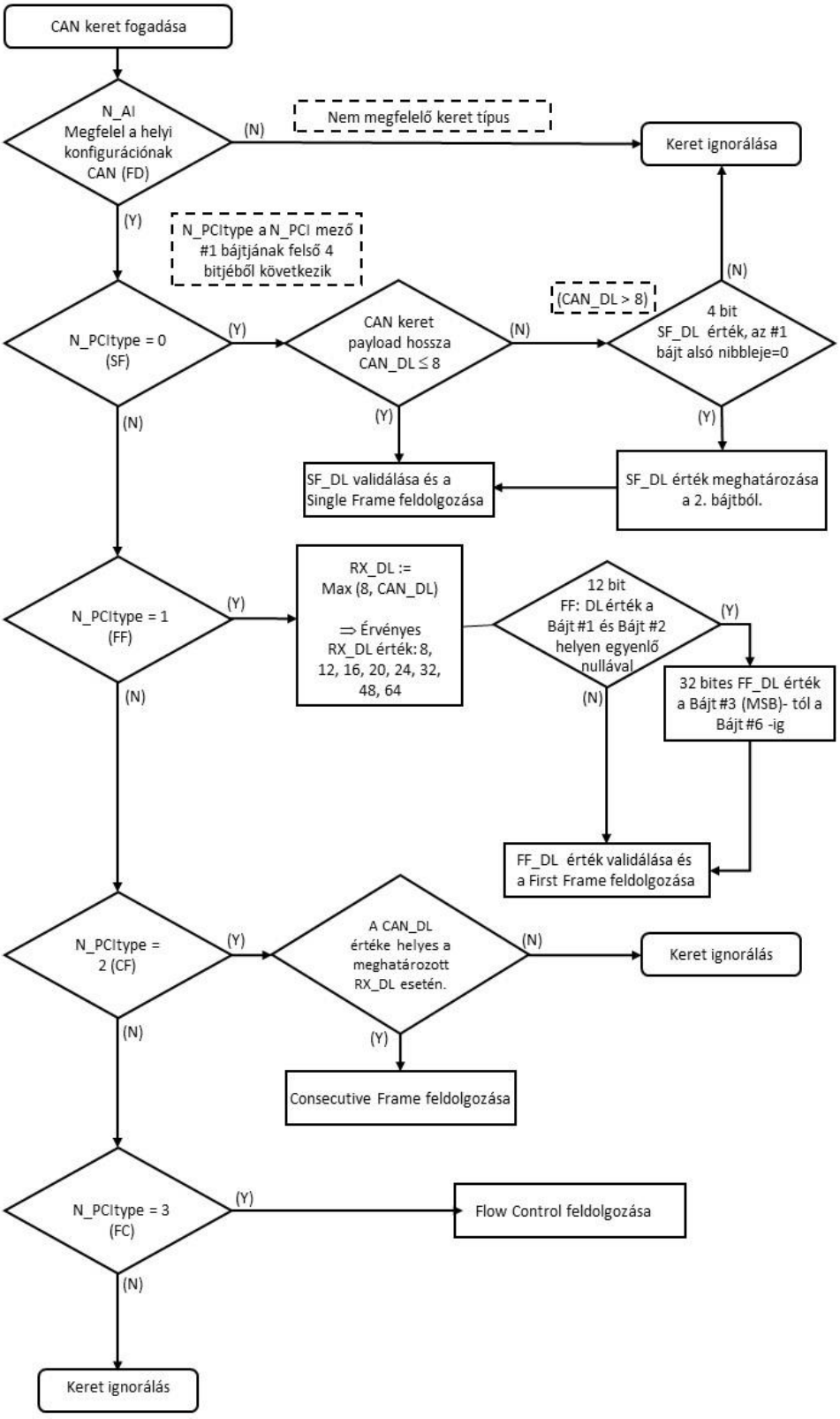
Ezeknél a teszteknél a teszteseteknek alkalmazkodniuk kell a modul konfigurációban előírt BlockSize és STmin szerinti működéshez, hiszen most a teszt szimulálja a küldő felet. Tehát minden egyes BlockSize elérésekor várunk a fogadó oldaltól egy FlowControl üzenetet, aminek a konfigurációban meghatározott értékeket kell tartalmaznia.

Egy keret beérkezésekor a CanTp-nek számos ellenőrzést kell elvégeznie, mire azt lehessen mondani, hogy az adott üzenet érvényes és továbbítható a felsőbb réteg számára. A továbbítási adatkapcsolati réteg adathossza (TX_DL) konfigurálja a hálózati réteget megvalósító alkalmazás maximálisan használható adatkapcsolati réteg terhelésének hosszát, ahogy azt az ISO 15765-2 ezen részében meghatározták. Mivel a

küldő csomópont TX_DL konfigurációja nem ismert a fogadó számára, a fogadó csomópontnak mindig alkalmazkodnia kell a küldő TX_DL beállításaihoz.[17]

A helyileg konfigurált N_TA (Network Target Address) típus lehetővé teszi a kapott CAN-keret típusának (CLASSICAL-CAN/CAN-FD) ellenőrzését, és arra használják, hogy figyelmen kívül hagyják a helytelen N_TA típusú kereteket. Amint az N_TA típus helyes, ellenőrizhetők a különböző N_PCI típusok, és feltételezéseket lehet tenni a RX_DL (a küldő TX_DL) kapcsán.[17]

A CAN_DL (CAN keret hossza, N-PDU hossz) értéke helyes, ha az érték megegyezik RX_DL (hasznos adat hossza) értékével az összes CF-re vonatkozóan az utolsó CF kivételével. Az utolsó CF akkor felel meg ennek a feltételnek, ha a CAN_DL kisebb vagy egyenlő, mint az RX_DL. Az RX_DL a FF-ből származik és rögzített a PDU fogadási folyamatához. Egy CAN keret fogadásának folyamatát és elbírálását mutatja a 28. ábra.



28. ábra: CAN keret fogadása [17]

A következő tehát az érvénytelen esetek tesztelése volt. Elsőként a SingleFrame-ekből állítottam össze érvénytelen eseteket, ami annyiban egyszerű volt, hogy a segédfüggvényekkel könnyedén összeállítottam a különböző N-PDU-kat, majd mindig megváltoztattam (elrontottam) egy bizonyos részt, aminek eredményeként különböző módon kellett, hogy elutasítsa a modul a beérkező üzenetet. A különböző hibák bevitelét külön tesztesetekbe válogattam a könnyebb átláthatóság miatt. Egy beérkező SF esetén az N-PDU hossza, valamint a PCI mezőben megtalálható hasznos adat hossza erősen összefüggenek.

N_PDU	N_PCI bájt mező							
	Byte #1		Byte #2	Byte #3	Byte #4	Byte #4	Byte #5	Byte #6
	Bits 7-4	Bits 3-0						
SingleFrame (SF) (CAN_DL ≤ 8)	0000_2	SF_DL	----	-----	----	----	----	----
SingleFrame (SF) (CAN_DL > 8)[a]	0000_2	0000_2	SF_DL	-----	-----	-----	-----	-----
FirstFrame (FF) (FF_DL ≤ 4095)	0001_2	FF_DL		-----	-----	-----	-----	-----
FirstFrame (FF) (FF_DL > 4095)[b]	0001_2	0000_2	$0000\ 0000_2$	FF_DL				
ConsecutiveFrame (CF)	0010_2	SN	-----	-----	-----	-----	-----	-----
FlowControl (FC)	0011_2	FS	BS	ST_{min}	N/A	N/A	N/A	N/A

a: Az üzenetek amik CAN_DL > 8-nak megfelelnek olyan escape szekvenciát kell használniuk, amelyben a Byte #1 alsó nibble-je 0-ra van állítva (érvénytelen hosszúság). Ez azt jelzi a hálózati réteg számára, hogy a SF_DL értékét a keret következő bájtja (Byte #2) alapján határozzák meg. Mivel CAN_DL nagyobbnak van definiálva, ez a meghatározás csak CAN_FD típusú keretekre érvényes
b: A 4095 bájtnál nagyobb üzeneteknek olyan escape szekvenciát kell használniuk, amelyben a Byte #1 alsó nibble-je és a Byte #2 összes bitje 0-ra van állítva (érvénytelen hosszúság). Ez azt jelzi a hálózat számára, hogy az FF_DL értékét a keret következő 32 bitje alapján határozzák meg (Byte #3 az MSB és a Byte #6 az LSB).

29. ábra: A PCI mező [17]

A 28. ábra minden lehetséges útvonalát leteszteltem, tehát a kötelező és a konfigurálható padding működését is egyaránt. Ezt lényegében olyan N-PDU hosszúságokkal értem el, amelyek nem fordulhatnának elő helyes működés alatt, mint N-PDU hossz. Minden bekonfigurált kapcsolaton végig próbáltam a helytelen kombinációkat. SF esetben, ha a CAN_DL nagyobb mint 8, akkor már a PCI 2. bájtján helyezi el payload hosszát (SF_DL). (29. ábra) Ez viszont magával vonja, hogy akár 255 érték is kerülhet ide, ami abszolút helytelen egy SF esetében, ahol a maximum payload hossz legfeljebb 62 lehet. A címzéstől és a bekonfigurált PDU hossztól függően az

SF_DL mérete maximum CAN_DL-2 vagy CAN_DL-3 lehet. A helyes működésnél már leteszteltem a határértékekre és további helyes értékekre a SF-eket, ezért a címzés szerint megközelítettem az érvénytelen oldalról is a határértéket, hogy megbizonyosodjak a helyes működésről. Főként a helyes feltételek és elágazások meghatározása jelentett kihívást a teszteset összeállításában, mivel a konfigurációban szereplő minden kapcsolatra lefuttattam a teszteket.

Folytatásként a FF érvénytelen eseteinek tesztelése volt a cél. Itt nem csak a felső, de az alsó határra is tesztelni kellett a hosszokat. Ha a szegmentált átvitel első keretében megadott payload hossz (FF_DL) belefér egy SF-be, akkor helytelen szegmentált átvitelt folytatni. Ezenfelül, ha az FF_DL értéke meghaladja a 4095 bájtot, akkor a FirstFrame PCI mezőjébe bekerül egy escape sequence, ami azt jelenti, hogy a PCI mező első és második bájtja is 0 értéket vesz fel. (29. ábra) Ez jelzi a hálózati rétegnek, hogy a FF_DL értéke a következő 32 biten alapul. Viszont, ha escape sequence van jelen a PCI mezőben, akkor az FF_DL nem lehet rövidebb mint 4095 bájt. Így hát kipróbáltam minden lehetséges kombinációt a hibás esetek előállítására.

ConsecutiveFrame–ek esetén a címzésnek, az SN számnak és az N-PDU hossznak kell megfelelnie egy helyes keret fogadásához. A CAN_DL-nek meg kell egyeznie a FF PDU hossz értékével, kivéve, ha az utolsó keretről van szó, mivel ott a padding-től függően optimalizálható. Következésképp létrehoztam teszteket, amelyek téves SN számot helyeznek egy átvitel egy ConsecutiveFrame PCI mezőjébe, valamint manipuláltam az N-PDU hosszokat a köztes és az utolsó keret küldésekor.

Egy modul csatornája kétféle képpen lehet konfigurálva, Full- vagy Half-duplex módban. Az egyes módokban az újonnan beérkező (váratlan) üzenetekre másként kell, hogy reagáljon a modul. Alapvetően elválasztottam a két mód tesztelését, azon belül is külön teszteltem a küldés és fogadás folyamata alatt beérkezett nem várt PDU-kat. Tehát egy folyamatban lévő fogadás vagy küldés alatt érkezik egy FF vagy egy SF fogadási kérelem, amit modulnak megfelelően kell lekezelnie.

A tesztesetekben elindítottam egy fogadási folyamatot, amit eljuttattam különböző fázisokba, például a FC kiküldése utáni állapotba, ahol a modul a következő CF-re számítana, ehelyett váratlan módon egy SF-et vagy egy FF-et kap esettől függően. Tesztelés során felfedeztem egy apró hibát a modul működésében. A modul várakozik a FC üzenet elküldése után, hogy meghívják a CanTp_Confirmation callback függvényt, ami a FC üzenet visszaigazolására szolgálna, azonban váratlanul egy PDU érkezik a

modulhoz, aminek hatására a modul egy téves állapotba került és helytelen működést produkált. A beérkezett PDU-t ahelyett, hogy továbbította volna a felsőbb réteg számára, inkább ignorálta. Ez a hiba, mint kiderítettem egy korábbi verziójában is jelen volt a modulnak. Egyeztetés után a hiba javításra került, mind az aktuálisan tesztelt, mind a korábbi verzióban is, azaz a modul feldolgozta a beérkezett PDU-t és megfelelően továbbította a PduR számára.

Váratlan PDU érkezését tovább teszteltem a modul küldési állapotában is. A FirstFrame kiküldése után a teszt összeállított egy megfelelő, normál állapotban fogadható üzenetet, amelyet a modul megkapott. A modul helyesen működött, miszerint Full-duplex módban feldolgozta az adott (váratlan) FF-et vagy SF-et, míg Half-duplex módban ignorálta azt.

5.6 CanTp_ChangeParameter és CanTp_ReadParameter függvények tesztelése

A CanTp képes futási időben egyes paraméterek módosítására és visszaolvasására. Ilyen paraméter a BlockSize és az STmin értékek. Ezeket az értékeket bármely konfigurált kapcsolatnál olvashatjuk és írhatjuk, bizonyos feltételek mellett. A CanTp_ChangeParameter API, ha a konfiguráció engedi és nincs folyamatban fogadás, akkor képes egy hívás során egy paraméter megváltoztatására, ezért lefuttattam érvényes és érvénytelen bemeneti paraméterek esetén a függvényeket. STmin érvényes értékeknek a 0, 0x3F, 0x7F, 0xF1 és 0xF9 számokat választottam, mivel ezek esnek az érvénytelen és érvényes értékek határára. A helyes működés ellenőrzését az értékek visszaolvasásával végeztem, valamint, hogy megbizonyosodjak, hogy tényleg a megváltoztatott értékeket használja fel a modul, ezért egy szegmentált átvitelt indítottam, ahol vizsgáltam a FC-ban érkező értékeket. Ezen felül, a tesztek során megpróbáltam olyan értékeket is beállítani a modulban, amelyek nem felelnek meg a helyes specifikációnak. Ezzel azt szerettem volna kideríteni, hogyan reagál a modul az érvénytelen adatokra, és milyen hatást gyakorolnak a modul állapotára és működésére. Érvénytelen adatok esetén a modul API hívása negatív eredménnyel tért vissza és nem használta fel a beadott értékeket.

A CanTp_ReadParameter API esetén, a tesztem lekérte a megfelelő kapcsolathoz tartozó konfigurációban lévő értéket és ezt hasonlította össze az API által visszaadott értékkel. Amennyiben érvénytelen kapcsolat azonosítóhoz (I-PDU Id) tartozó értéket próbált lekérni a teszt, akkor negatív eredménnyel tért vissza az API, tehát megtagadta a kérést és a működése helyesnek bizonyult.

5.7 CanTp_CancelTransmit és CanTp_CancelReceive függvények tesztelése

A CanTp képes a fogadási és küldési folyamat megszakítására egyaránt. A CanTp elutasítja a fogadás megszakítási kérelmét abban az esetben, ha egyetlen keretet fogad, vagy ha a megszakítási kérelem éppen az utolsó két egymást követő kerete közben érkezik. (Azaz a szolgáltatás hívása az N_Cr időzítő elindítása után történik, az utolsó egymást követő keretekre). Valamint a kérés elutasításra kerül, ha a megadott N-PDU azonosítóhoz nem tartozik aktív kapcsolat. Ezek alapján készítettem két tesztet, amelyeknél sikertelen megszakítást vártam. Egy esetben az átvitelt az utolsó két keret közötti állapotba vezéreltem, a másik esetben ismeretlen kapcsolat azonosítót adtam meg. Ezeken felül egy sikeres megszakítást is leteszteltem, ahol egy szegmentált üzenet fogadását indítottam el és szakítottam meg a FlowControl kiküldése után. A megszakítás után a CanTp-nek folytattam a küldés szimulációját, a következő ConsecutiveFrame-et el kellett, hogy utasítsa a modul. SingleFrame fogadásának a megszakítása nem lehetséges, mivel a CanTp_RxIndication API törzsén belül történik meg a feldolgozás, ami megszakíthatatlan.

A küldés megszakítása esetén egy plusz konfigurációs paraméter is érvényben van. A csatornákon belül a küldéshez tartozó kapcsolatok megszakíthatósága konfigurálható, tehát amennyiben a megszakítás nem engedélyezett az adott kapcsolaton, minden esetben a megszakítási kérés elutasítását várom el. Ha megszakítási kérelem érkezik egy olyan aktív kapcsolatra, amelynél a megszakíthatóság nem engedélyezett, akkor Det hiba üzenetet várok el. Leteszteltem SingleFrame és szegmentált üzenet átvitelének a megszakítását is. SingleFrame esetén a beérkezett átviteli kérelem és a main függvény meghívása között eszközöltem a megszakítást, míg szegmentált átvitel esetén a FirstFrame kiküldése után. A tesztek sikeresek voltak, a modul a követelményeknek megfelelő eredményt szolgáltatott.

5.8 CanTp_Init és CanTp_ShutDown függvények tesztelése

A CanTp modulnak két fő állapota van. Inicializált (bekapcsolt), avagy leállított (nem inicializált). Leállított állapotban az API függvények (a verzió lekérdezést megvalósító API-t kivéve) nem működnek, azaz hívásuk esetén negatív választ kell szolgáltatniuk, ezért minden egyes API-t meghívtam leállított állapotban. A konfigurációban beállítható számos CanTpConfig konténer, amely alapján a modul specifikusabb működése definiált. Itt vannak konfigurálva a csatornák és a hozzájuk

tartozó kapcsolatok. Az init függvényben egy ilyen CanTpConfighoz tartozó adatstruktúra pointere kerül átadásra, amivel inicializálódik a modul. Csak leállított üzemmódban inicializálható a modul, viszont ez akár futásiidőben is megtehető.

Ezek alapján állítottam fel teszteseteket, amelyek inicializált és leállított állapotban próbálják leállítani és újra inicializálni a modult. Bekapcsolt állapotból bekapcsolni és kikapcsolt állapotból kikapcsolni nem lehet, ezért hiba üzeneteket várok el a modultól ezekben a szituációkban.

5.9 Meta adatok tesztelése

Meta adat kezelés esetén a teszt bemenő paraméterei maga a MetaData, amely a PDU-val érkezik a felsőbb vagy alsóbb rétegtől a PduInfo részeként, valamint a konfigurációs paraméterek, azaz a CanTpDynIdSupport és CanTpGenericConnectionSupport kapcsolók. Ezeket a teszteseteket csak meta adat kezelés esetén futtatom, tehát ha legalább a CanTpDynIdSupport engedélyezve van. A tesztesetekben a korábbiakhoz hasonlóan külön vizsgáltam egy üzenet fogadását és kiküldését meta adatokkal. A fejlesztés során a korábbi teszteseteimet átalakítottam, hogy a normál működés során, amennyiben meta adatokat kéne használnia egy tesztesetnek a konfiguráció alapján, akkor azok helyes működést eredményezzenek és minden esetben a konfigurációból vett címeket használják fel mint bemenet és mint elvárt értékek. Ez azért fontos mert minden teszt lefut az összes konfigurációra és a CanTpGenericConnectionSupport kapcsoló aktivált állapotában a felsőbb rétegtől érkező meta adatokat használná fel a modul kiküldéshez, viszont a tesztek ebben az esetben is a konfigurációból vett értékeket használják fel és adják át a CanTp-nek a PduR-en keresztül. Ebből adódóan a meta adatok helyes kezeléséről számos tesztesetem megbizonyosodik, tehát ezután a helytelen vagy nem várt beérkező üzenetek tesztelésére fókuszáltam.

Megvizsgáltam, hogy ha a felsőbb rétegtől olyan meta adatok érkeznek amelyek egyáltalán nem találhatóak meg a konfigurációban, akkor miként viselkedik a modul különböző konfigurációk esetén. A CanTp továbbra is helyes működést mutatott, tehát a kimeneten a megfelelő értékeket adta vissza, azaz a konfigurációtól függően a konfigurált csatornán megtalálható vagy a PduR-től érkezett meta adatokban megtalálható címzési paramétereket. Amennyiben a CanTp a felsőbb rétegtől várta volna a meta adatokat, de

nem kapott, akkor a modul helyesen elutasította az adott üzenet kiküldését és hiba üzenetet szolgáltatott a Det modulnak.

Ha be van kapcsolva a meta adatok kezelése akkor generikus kapcsolatok között egy üzenet fogadása során azonos N-PDU id esetén a meta adatok alapján tesz különbséget a modul az egyes kapcsolatok között. Tehát a konfigurációban készítettem olyan csatornákat ahol az N-PDU id-k megegyeztek, viszont a címzési paraméterek nem, ezáltal vizsgáltam, hogy melyik kapcsolatot rendeli a modul a beérkező üzenethez.

A fogadó modulnak el kell mentenie a beérkező üzenethez tartozó meta adatokat az átvitel során és azok alapján kell kiküldenie a FC üzenetet. A FC üzenetben a Source Address és a Target Address felcserélődik ezért ezt fogadói és küldő oldalról is megvizsgáltam. A tesztekben üzenetek fogadása során a felcserélt cím paramétereket vártam a FC üzenetben, valamint összeállítottam helytelen FC üzenetet, hogy vizsgáljam a CanTp reakcióját hiba esetén. A modul nem várt üzenetként kezelte a helytelenül összeállított FC üzenetet, tehát a CanTp működése helyesnek bizonyult.

5.10 Generátor tesztek

A generátor modellellenőrzését intenzíven kell tesztelni érvényes és érvénytelen bemenetek használatával. Ha a modell konfiguráció hibás, akkor a modul konfigurációs fájljainak nem szabad létrejönniük, ezáltal is meggátolva a helytelen beállítás használatát. A modul specifikációból következik, hogy mely beállítások együttese, vagy mely értékek beállítása esetén érvénytelen egy bemenet. A modul fejlesztéséhez hozzátartozik konzisztenciaellenőrzések létrehozása, melyek hibás bemenetek esetén meggátolják a kódgenerálást és hibaüzenetben tájékoztatnak a hiba okáról. A generátortesztelés érdekében létrehoztam modelleket, amelyek segítségével elválasztottam az egyes hibás esetek tesztelését. Egy teszt akkor sikeres, ha a kimeneti fájlok tartalma megegyezik a referencia fájlokéval. A hibás esetekhez tartozó referencia fájlok nem tartalmaznak generált modell és C forrás fájlokat. Referenciaként felvételre kerül egy szöveges fájl, amely tartalmazza a konfiguráció generálásakor keletkező elvárt konzisztencia hiba üzeneteket.

A tesztelés során elsőként létrehoztam egy funkcióhoz tartozó modellt, ahol megadtam hibás bemeneteket. Az első futtatás alkalmával megvizsgáltam a szolgáltatott hiba üzenetet és amennyiben releváns és informatív volt az üzenet, akkor felvettem a referencia fájlba az elvárt konzisztencia üzenetekhez.

Összesen 12 modell fájl volt szükséges a hibás bemenetek lefedéséhez, amelyekben különböző értékeket kaptak az általános (CanTpGeneral) valamint a specifikus (CanTpConfig) paraméterei a konfigurációnak.

A modul általános beállítási között megtalálható a CanTpDynIdSupport és a CanTpGenericConnectionSupport kapcsolók. Amennyiben a CanTpGenericConnectionSupport kapcsolót engedélyezni szeretnék, akkor szükséges követelmény a CanTpDynIdSupport kapcsoló engedélyezése is. Ennek tudatában létrehoztam egy tesztet, amelyben a CanTpGenericConnectionSupport-ot engedélyezem, viszont a CanTpDynIdSupport kapcsolót nem. A konfigurációhoz tartozó generálásakor, ahogy vártam, egy hiba üzenetet kaptam és semmi mást. (Nem kaptam generált kódot.) A kapott hiba üzenet a következő volt: "CanTpDynIdSupport shall be set if CanTpGenericConnectionSupport is true.". Ez a hibaüzenet teljesen megfelel, mivel reflektál a hiba okára, ezért ezt az üzenetet bevettem a referencia log-ba így sikeres lett a teszt.

Ahol bizonyos intervallumban volt értelmezhető az adott bemeneti érték, ott a határérték analízisnek megfelelően a határértékre és a szomszédos értékekre teszteltem, amely egy biztosan érvényes és egy biztosan érvénytelen bemenetet jelent. Ilyen értékek voltak a main függvény periódusának értéke, az érvényes PDU hossz értékek vagy a Source Address, Target Address és Address Extension értékek.

A követelményekben megszabták, hogy a választott címzési mód alapján mely cím fajtákat szükséges bekonfigurálni. Például az ECUC_CanTp_00284-es követelmény szerint, ha a CanTpNSdu-nak CANTP_MIXED vagy CANTP_MIXED29BIT a címzési formája, akkor a CanTpNAe-t be kell állítani.

A tesztelés eredményeképpen egy új ellenőrzés került be a modulba, amely vizsgálja a konfigurált Source Address, Target Address és Address Extension értékeket, amelyek csak 0-255 tartományon vehetnek fel érvényes bemenetet. Mivel ezek az értékek 1 bájton kerülnek eltárolásra, az érvényes intervallumon kívül felvett értékek nem várt működést eredményeznének.

A követelményeknek megfelelően további generátor teszteket vettem fel a modulhoz, amely működése megfelelt az elvártnak, így a tesztek sikeresen futottak.

Az ellenőrzéseket megvalósító osztályra 100% a branch fedés, azaz kaptam visszajelzést arról, hogy minden ellenőrzést kipróbáltam érvényes és érvénytelen

bemenettel. A helyesen összeállított konfigurációk generálása a teljes konfigurációs kódgenerátor 96%-át lefedi, tehát a helyes konfigurációkat tartalmazó modellek is megfelelően diverzifikáltnak tekinthetők.

5.11 Integrációs tesztek

A CanTp modul hardveren való teszteléséhez szükséges volt az egész CAN stack bekonfigurálása. A rendelkezésemre álló ECU 3 CAN-csatornával rendelkezett, amelyből én kettőt használtam fel. Az alapvető ötlet az, hogy a Dcm modul API függvényei helyett saját logikát helyezek el, amelyben megvalósítom a PDU-k indítását, fogadását és összehasonlítását. A CanTp robusztusságának vizsgálatára a CanIf modulban helyezek el saját kódrészleteket, amelyek üzenettől függően a PCI mezőben valamilyen hiba jelenséget generálnak.

A modulok konfigurációját harmonizáltan elkészítettem, miszerint az egyes azonosítók és összeköttetések egymással összhangban voltak. Ezenfelül a hardver két CAN csatornáját összekötöttem egymással, tehát a különböző üzenetek küldése és fogadása ugyanazon az eszközön valósulhatott meg.

A teszt során új szoftvereket és eszközöket ismertem meg, amelyek a CAN üzenetek buszon történő nyomkövetését és a program futásának megfigyelését tették lehetővé a céleszközön. A CANoe program és egy Vector doboz segítségével lehetőségem volt a CAN-busz megfigyelésére.

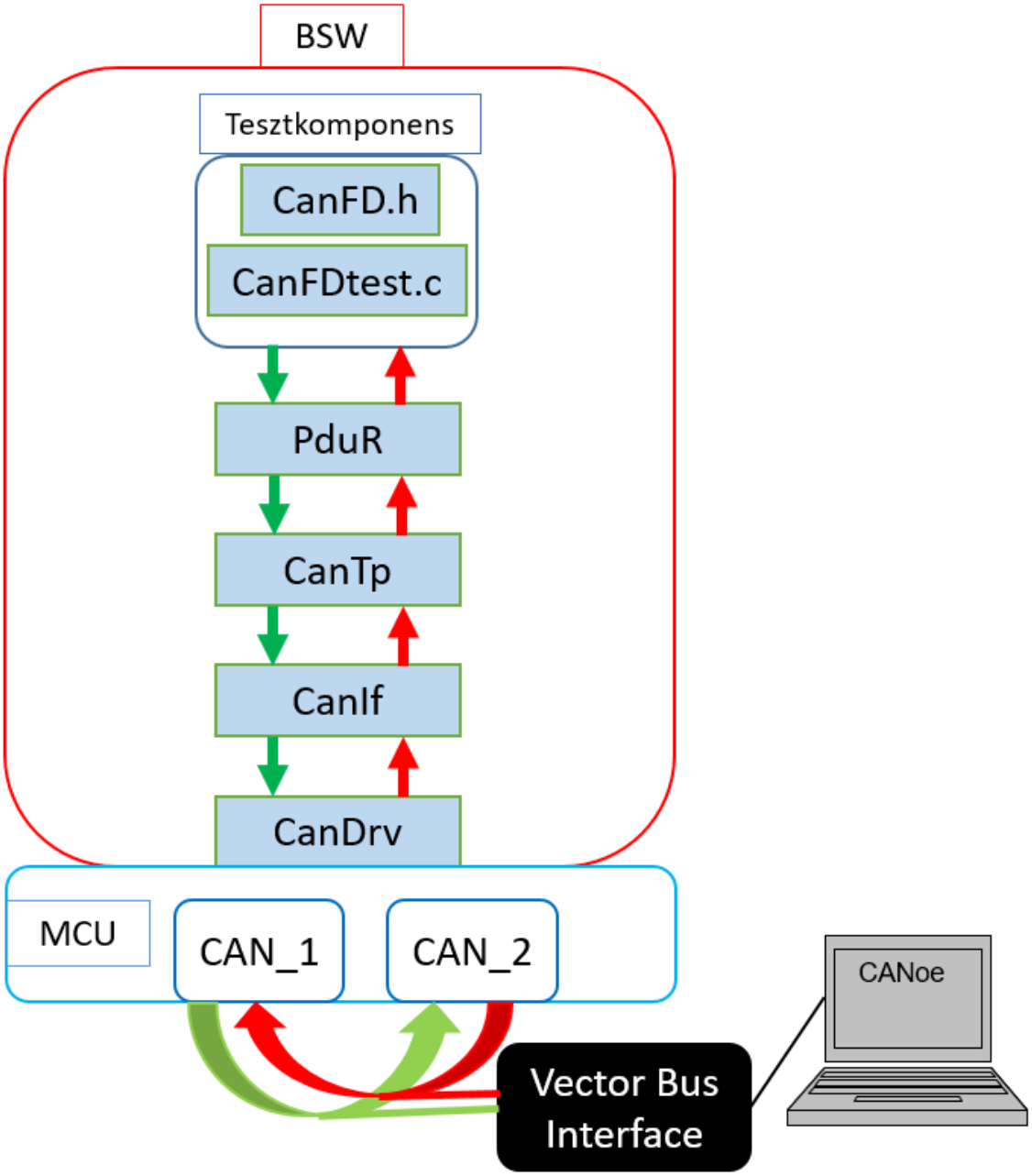
A CANoe program egy olyan szoftvereszköz, amelyet a Vector Informatik fejlesztett az autóipari hálózatok tesztelésére, fejlesztésére és diagnosztizálására. A program képes a Controller Area Network (CAN) buszon alapuló kommunikációs rendszerek széles körének támogatására.

Maga a Vector doboz egy specifikus hardvertermék, amelyet ugyancsak a Vector Informatik cég készít. Ez a termék egy FlexRay és CAN busz analizátor, amelyet például az autóipari tesztelési és diagnosztikai alkalmazásokban használnak.

A Vector doboz és a CANoe lehetővé teszi a FlexRay és CAN buszok kommunikációjának monitorozását, az adatok rögzítését és analizálását. A segítségükkel az autóipari fejlesztők képesek azonosítani és diagnosztizálni a hálózati kommunikációs problémákat, megfigyelni az adatforgalmat, elemezni a buszüzeneteket, és felismerni a hibákat vagy rendellenességeket a rendszer működésében. Ezenkívül ezeket a terméket olyan esetekben is használhatják, amikor az autóipari hálózatokat valós idejű

rendszerekkel, például szimulátorokkal vagy tesztforgatókönyvekkel kell kapcsolni, vagy akár az autóelektronika fejlesztésének folyamatában is.

A tesztekhez az összekötött CAN-csatornát hozzákapcsoltam a Vector dobozhoz, így PC-n keresztül képes voltam az üzenetek valós idejű követésére. (30. ábra) A hardveren futó szoftver nyomonkövetésére egy Lauterbach által fejlesztett Trace32 eszközt használtam, amely egy fejlett hibakereső és fejlesztőrendszer.



30. ábra: A hardver teszt felépítése

A Trace32 szoftver és a hozzá tartozó hardvereszközök együttműködésével lehetőséget nyújt a beágyazott rendszerek fejlesztésére, hibakeresésére és teljesítményelemzésére. A Trace32 kiterjedt támogatást nyújt különböző processzorarchitektúrákhoz és beágyazott rendszerekhez, beleértve az ARM, PowerPC, MIPS, x86 és más platformokat is. A rendszer széles körben használatos az ipari, telekommunikációs, gépjárműipari és más beágyazott rendszerfejlesztési területeken, ahol a megbízható és hatékony hibakeresés és fejlesztés kritikus fontosságú.

Az üzenetek kiküldésének és fogadásának megvalósítását a Dcm modul interfészének segítségével végeztem. A Dcm modul számomra hasznos funkcióit és kapcsolatait megtartva a környező BSW modulokkal implementáltam egy tesztlogikát, aminek segítségével a CanTp hardveren való tesztelése megvalósítható. Minden alkalommal amikor egy üzenet kiküldéséről visszaigazolást kap a teszt, 2 másodperc időkülönbséggel elindítja a következő küldését. Először minden üzenethez 8 bájtos PDU-t konfiguráltam, amely lehetőséget adott a konfiguráció és egyszerűbb üzenetek tesztelésére. Hasonlóan a szoftveres modulteszthez, az üzenetek méretét minden elküldött üzenet után eggyel növeltem, ahol az üzenetek tartalma egy 1-től növekvő számsorozat volt. Tehát a teszt elején SF-ek, majd ahogy az üzenet mérete meghaladja konfigurált PDU méretet, úgy szegmentált átvitel valósul meg. Hasonlóan leteszteltem az összes FD hosszúságra, amely során hasonlóan a 8 bájtos mérethez, először SF-ek majd szegmentált átvitelt folytatott a modul, teljes mértékben helyesen.

A hiba injektálást a CanIf modulban tett módosítással oldottam meg közvetlenül mielőtt a CanIf továbbítaná az üzenetet a CanTp modul számára. A hibák injektálást minden második beérkező üzenet esetén eszközöltem, mivel így továbbra is megbizonyosodhattam arról, hogy injektálás nélkül az üzenetek átvitele sikeresen végbe megy. A CAN stack felkonfigurálásakor normál címzési módot alkalmaztam, tehát 11 bites címzést. Továbbá külön CAN id-t konfiguráltam minden PDU hossz esetén, mivel így láthattam, hogy adott csatornán a megfelelő hosszúságú üzenet érkezik be. Ezenfelül külön CAN ID-t konfiguráltam az egyes csatornákon válasz üzenetként érkező FC üzeneteknek is, ami így összesen a CanTp-ben 8 küldő és 8 fogadó kapcsolatot jelentett egy csatornán.

A hibainjektálás során a fogadott üzenetek típusát (SF, FF, CF, FC) először azonosítottam a programban majd különféle módon rontottam el, azaz ismeretlen PDU id-t továbbítottam, valamint a PCI mezőt módosítottam olyan módon, hogy azt a CanTp

ne fogadja el. Például FF esetén a PCI mezőben adathossznak 5 értéket adtam meg, ami helytelen mivel ez az adathossz méret biztosan belefér egy SF-be is, tehát nem kerülhet bele egy FF-be. A PCI mezőben, a szoftveres tesztekhez hasonló módon eszközöltem változtatásokat a hibás működés előidézése érdekében. A helytelen üzenetek segítségével az egyes időzítők működését is le tudtam tesztelni, mivel a hibás üzenetek beérkezése után a vevő és a küldő oldal is hiba üzenetet küld a Det modulnak. Mivel helyes üzenet nem érkezik és szegmentált átvitel esetén FC sem kerül visszaküldésre, így a modulokban timeout keletkezik, ami kiváltja a hiba üzenetet. Egy üzenet elvesztését is szimuláltam egy szegmentált átvitel során a ConsecutivFrame-ben lévő SN érték megnövelésével. A későbbi tesztek során különböző címzési típusokat és címparamétereket adtam meg a konfigurációban, így tesztelve a meta adatok megfelelő használatát.

Összességében a modul integrált tesztelése során nem találtam hibát, minden bevitt hibára a specifikációnak megfelelő módon ragált a modul. Az integrációs tesztet tovább lehetne fejleszteni további tesztesetek létrehozásával, amelyekben a modulok diverzifikáltabb konfigurációját alkalmaznám.

6 EREDMÉNYEK ÉS ÖSSZEFOGLALÁS

A diplomamunkám során átfogóbban megismerhettem az AUTOSAR BSW felépítését, valamint az egyes modulok együtt működését. Részletes betekintést nyertem egy ECU-n megtalálható kommunikációs stack-be. Megismertem különböző kommunikációs protokollokat, köztük a FlexRay és CAN-FD protokollokat behatóbban is. A CAN-FD és a FlexRay protokollok összehasonlítása során kevésbé előnyöket és hátrányokat véltem felfedezni, inkább csak protokollok közti funkcionális különbségeket találtam.

Megismertem a tesztelés elhelyezkedését, módjait és fontosságát egy szoftver életciklusában. Különböző teszt folyamatokkal és technikákkal találkoztam, amelyeket a feladattól függően lehet érdemes felhasználni szoftvertesztelésre.

A modulteszt során megismertem az AUTOSAR CanTp modul minden funkcióját és működését. A modul tesztelése során részletesen megvizsgáltam a követelményeket, felépítettem egy tesztkörnyezetet és a funkcióktól függően teszt suite-okat hoztam létre, amelyekben a tesztesetek a követelmények alapján különböző konfigurációkra letesztelték a modult. A tesztelendő funkciókhoz határérték analízist készítettem, amely a teszt teljességét is igazolja.

A tesztelés során felfedeztem hibákat és optimalizációs lehetőségeket, amelyek alapján a modul fejlesztőjével egyeztetve sikerült javítani, továbbfejleszteni a modul működését. Találataim voltak mind a követelményekben, mind a modul működésében és a konfiguráció generálásában is. Minden tesztesethez, majd minden teszt suite-hoz doxygen kommentet fűztem, amely dokumentálja az egy kódrészek tartalmát, valamint a minden tesztelhető követelményt referáltam a teszt kódbázisban. A tesztek minden konfiguráció esetén sikeresen átmennek, ezáltal kimondható, hogy a modul megfelel a követelményeknek, valamint a teljes tesztelendő kód minden ágát elértem. Ezt egy a cégen belül használt mérő eszköz igazolja, amely 100%-os branch-coverage értéket mutat.

A modul generátorának letesztéléséhez összeállítottam hibás bemeneti konfigurációkat, amelyekre a modul megfelelően informatív hiba üzenetekkel reagált, valamint megtagadta bármilyen konfigurációs fájl létrehozását.

A diplomamunkám során végezetül a modul tesztelését végeztem hardver eszközön. Megismertem új eszközöket, amelyeket az iparban is széleskörűen használnak

fejlesztésre és tesztelésre. A modulok felkonfigurálása és integrálása egy működő szoftverré igen komoly feladat, amely az összes kapcsolódó modul alapismeretét megköveteli. A tesztelést elvégeztem, viszont további lehetőségeket látok új tesztesetek létrehozására és az eddigiek fejlesztésére.

Összességében kimondhatom, hogy a CanTp modul tesztelését mind szoftveresen mind hardveresen elvégeztem és megállapítható, hogy a modul a követelményeknek megfelel.

IRODALOMJEGYZÉK

[1] Vajay Levente - AUTOSAR diagnosztikai Log and Trace-modul megvalósítása

[2] AUTOSAR - Layered Software Architecture: https://www.autosar.org/fileadmin/standards/classic/4-3/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf (2022.12.10)

[3] AUTOSAR - Specification of Diagnostic Communication Manager: https://www.autosar.org/fileadmin/fileadmin/standards/classic/20-11/AUTOSAR_SWS_DiagnosticCommunicationManager.pdf (2022.11.29)

[4] Dr. Szendrei Rudolf - Unit teszt: https://swap.web.elte.hu/2018191_pt2e/ea08.pdf (2022.12.04.)

[5] Ficsor Lajos, Kovács László, Kusper Gábor, Krizsán Zoltán - Szoftvertesztelés: https://dtk.tankonyvtar.hu/xmlui/bitstream/handle/123456789/13039/0046_szoftverteszteles.pdf?sequence=2&isAllowed=y (2022.11.29.)

[6] Széchenyi István Egyetem: Szoftver - tesztelési módszerek: http://www.sze.hu/~heckenas/okt/swmin3.pdf (2022.11.29.)

[7] Majzik I., Micskei Z.: Specifikáció alapú teszttervezési módszerek: https://inf.mit.bme.hu/sites/default/files/materials/category/kateg%C3%B3ria/oktat%C3%A1s/msc-t%C3%A1rgyak/szoftverellen%C5%91rz%C3%A9si-technik%C3%A1k/12/SZET-2012_07a_specifikacio_alapu_teszteles.pdf (2022.11.30.)

[8] techopedia - Code coverage: https://www.techopedia.com/definition/22535/code-coverage (2022.11.30.)

[9] Vector - FlexRay: FlexRay | Vector (2022.12.04.)

[10] J.Pradeep , S. Richerd Sebasteen , R. Dineshkrishna - COMPARISON OF CAN AND FLEXRAY PROTOCOL FOR AUTOMOTIVE APPLICATION: https://acadpubl.eu/hub/2018-119-14/articles/3/48.pdf (2022.12.04.)

[11] Dimitri van Heesch - CUNIT: https://cunit.sourceforge.net/ (2022.12.04.)

[12] Agile Essential - The 12 Principles behind the Agile Manifesto: https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/ (2022.12.11.)

[13] Csselectronix – CANFD Intro: https://www.csselectronics.com/pages/can-fd-flexible-data-rate-intro (2023.02.10.)

[14] Kvasher – Comparing CAN-FD with Classical CAN: comparing-can-fd-with-classical-can.pdf (kvaser.com) (2023.02.10.)

[15] ISO 10681-2:2010 - Road vehicles - Communication on FlexRay - Part 2: Communication layer services: ISO 10681-2:2010 - Road vehicles —

Communication on FlexRay — Part 2: Communication layer services (2023.05.29)

[16] AUTOSAR - Specification of CAN Transport Layer (4.3.0): Home AUTOSAR (2023.05.29)

[17] ISO 15765-2 - Road vehicles - Diagnostic communication over Controller Area Network (DoCAN)- Part 2: Transport protocol and network layer services: ISO 15765-2:2016 - Road vehicles — Diagnostic communication over Controller Area Network (DoCAN) — Part 2: Transport protocol and network layer services (2023.05.29)

[18] AUTOSAR - Specification of FlexRay ISO Transport Layer (4.3.0): Home AUTOSAR