DIPLOMATERVEZÉSI FELADAT

Róth Ádám (ILSXQ3)
szigorló villamosmérnök hallgató részére

Autóipari vezérlők memóriasérülésének felderítése és vizualizációja

Egy modern gépjármű biztonsági és komfortfunkcióit számos beágyazott vezérlőegység (ECU) támogatja. Annak érdekében, hogy az ECU a jármű teljes élettartamán át elviselje a fellépő jelentős fizikai igénybevételt (szélsőséges hőmérséklet, rázkódás, páratartalom, ingadozó tápellátás stb.), a desktop rendszerektől eltérően tartós adattárolásra nem merevlemezeket, hanem tipikusan EEPROM vagy flash alapú tárolókat használnak. A tartós memória elsősorban diagnosztikai információk tárolására használatos, vagyis egyfajta fekete dobozként viselkedik a járműben. Az autógyártó tesztjei során felmerülő egyes problémák esetén (pl. kormányrendszer rendellenes működése) az érintett beszállítónak elemeznie kell a hibajelenség okait. Ilyen esetekben előfordul, hogy addigra olyan súlyosan megsérül az ECU (logikai vagy fizikai memóriakárosodás), hogy a szoftver önvédelmi mechanizmusai az újraindulást is megakadályozzák, emiatt hagyományos módszerekkel szoftveresen nem lehetséges kiolvasni a hibatárolókat. Ilyen esetekben legtöbbször az ECU megbontásával még ki lehet nyerni a tartós memória utolsó tartalmát, amelynek adatszerkezete azonban nem triviális, megértése jelenleg hosszas elemzést igényel.

A hallgató feladata részletesen a következő:

- Ismerje meg a vállalatnál használt mikrovezérlő memóriastruktúráját, értse meg a hibatűrő adattárolás elveit és megvalósulását az AUTOSAR memória moduljaiban.
- Készítsen desktop alkalmazást, amely a mikrovezérlő bináris file-ba mentett memóriatartalmát elemzi, kinyeri a még olvasható adatokat és ember számára könnyen áttekinthetően vizualizálja a fizikai vagy logikai sérüléseket (célszerűen egy HTML riport formájában).
- Demonstrálja a rendszer működését néhány valós helyzetben gyűjtött memória dump elemzésével.

Tanszéki konzulens: Dr. Sujbert László, docens
Külső konzulens: Knoll Tímea (thyssenkrupp Components Technology Hungary Kft.)

Budapest, 2018. március 8.

.....
Dr. Dabóczi Tamás
tanszékvezető

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

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

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

Róth Ádám Gábor

AUTÓIPARI VEZÉRLŐK MEMÓRIASÉRÜLÉSÉNEK FELDERÍTÉSE ÉS VIZUALIZÁCIÓJA

Diplomaterv

KONZULENS

Knoll Tímea (thyssenkrupp);
Dr. Sujbert László (MIT);
Teveli Zoltán (thyssenkrupp)

BUDAPEST, 2018

Tartalomjegyzék

Kivonat **5**

Abstract **7**

1 Bevezetés **9**

2 Beágyazott mikrovezérlők memóriastruktúrája **11**

2.1 Flash memória áttekintés 11

2.2 Flash és EEPROM összehasonlítása 11

2.3 Hibatűrő adattárolás 12

3 AUTOSAR áttekintés **16**

3.1 Komponensorientált modellezés 16

3.2 AUTOSAR szoftverarchitektúra 18

3.2.1 Basic Software réteg 19

3.2.2 BSW modulok tulajdonságai 21

4 AUTOSAR memóriaverem **23**

4.1.1 NVRAM Manager 23

4.1.2 Flash EEPROM Emulation 25

4.1.3 Flash driver (FLS) 26

5 Felhasznált technológiák, eszközök **28**

5.1 JAVA 28

5.2 Xtend 29

5.3 HTML 30

5.4 Eclipse 31

5.5 JAVA verifikációs technikák és eszközök 32

5.5.1 JUnit 33

5.5.2 „Mockito” verifikáció 37

6 ECU integrált Flash memóriaanalízis **39**

6.1 Elemző szoftver felépítése 40

6.1.1 Bemenetek kezelését végző osztályok 40

6.1.2 Elemző adatstruktúrák és osztályok 41

7 Elemzés vizualizációja **43**

7.1 Interaktív HTML riport 43

7.2 Dinamikus riportgenerálás Xtend Template Expressions osztályokkal 48

7.2.1 HTML riportgeneráló Xtend osztályok 49
8 Verifikáció és validáció 52
8.1 Elemző szoftver moduláris (unit) tesztelése 52
8.1.1 Tesztkészlet és tesztesetek bemutatása 54
8.2 Teljes szoftver validációja 58
9 Eredmények értékelése 60
Irodalomjegyzék 61

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2018. 12. 14.

.....
Róth Ádám Gábor

Kivonat

Egy modern gépjármű biztonsági és komfortfunkcióit számos beágyazott vezérlőegység (ECU) támogatja. Annak érdekében, hogy az ECU a jármű teljes élettartamán át elviselje a fellépő jelentős fizikai igénybevételt (szélsőséges hőmérséklet, rázkódás, páratartalom, ingadozó tápellátás stb.), a desktop rendszerektől eltérően tartós adattárolásra nem merevlemezeket, hanem tipikusan EEPROM [1] vagy Flash [1] alapú tárolókat használnak.

A tartós memória elsősorban diagnosztikai információk tárolására használatos, vagyis egyfajta fekete dobozként viselkedik a járműben. Az autógyártó tesztjei során felmerülő egyes problémák esetén (pl. kormányrendszer rendellenes működése) az érintett beszállítónak elemeznie kell a hibajelenség okait. Ilyen esetekben előfordul, hogy addigra olyan súlyosan megsérül az ECU (logikai vagy fizikai memóriakárosodás), hogy a szoftver önvédelmi mechanizmusai az újraindulást is megakadályozzák, emiatt hagyományos módszerekkel szoftveresen nem lehetséges kiolvasni a hibatárolókat. Ilyen esetekben legtöbbször az ECU megbontásával még ki lehet nyerni a tartós memória utolsó tartalmát, amelynek adatszerkezete azonban ember számára kevésbé jól olvasható, megértése jelenleg hosszas elemzést igényel.

A feladat tehát egy olyan asztali alkalmazás készítése, amely képes a mikrovezérlőből kinyert memóriatartalom elemzésére és a fizikai vagy logikai sérülések ember számára könnyen áttekinthető vizualizációjára.

A fejlesztés elkezdése elött megismerkedtem az autóipari szoftverfejlesztés módszertanaival és szabványaival. Feladatom elvégzéséhez szükséges volt utánajárnom a beágyazott mikrovezérlők memóriastruktúrájának tulajdonságainak. Részletesen megismerkedtem a hibatűrő adattárolás ismérveivel, módszereivel különösképpen a thyssenkrupp Components Technology Hungary kft. vállalatnál használt nemfelejtőmemória-kezelést biztosító szoftvermodulok tulajdonságaival.

A diplomatervezés tantárgy fejlesztési fázisában elkészítettem a cégnél használt vezérlőegységek memóriatartalmának elemzésére és az elemzés eredményeinek vizualizációjára képes alkalmazást JAVA programozási nyelven. Az elemző szoftvert kiegészítettem az Xtend Template Expressions funkciójának segítségével megvalósított

HTML riport generátorral. Az elemző szoftvert a cég által fejlesztett AUTOSAR Architect névre keresztelt Eclipse alkalmazásba integráltam bővítmény formájában.

A jól elkülöníthető funkcionális egységeken modultesztelést hajtottam végre, amivel sikerült a tervezési fázisban meghatározott követelményeim teljesülését ellenőriznem.

Abstract

The safety and comfort features of a modern vehicle are supported by many embedded control units (ECU). To ensure that the ECU tolerates significant physical stress (extreme temperature, vibration, humidity, fluctuating power, etc.) throught the whole life of the vehicle, they typically use EEPROM or Flash based storage instead of hard drives used in general customer systems.

These non-volatile memories are primarily used to store diagnostic information about the ECU, meaning a kind of black box in the vehicle. For some problems during the tests of car manufacturers (eg. abnormal behaviour of steering system) the related supplier must analyse the cause of the malfunction. In these cases, ECU could be seriously damaged (logical or physical memory impairment), if the self-protection mechanisms of the software deny the restarting, that means the memory content cannot be read by using standard software processes. In such cases, most of the time the last content of the non-volatile memory can be reached, if ECU is disassembled. The data structure of the reached content is less clear for humans and the understanding of it requires a lengthy analysis.

The goal is to create a desktop application which can analyse memory content extracted from the microcontroller and provide informative visualization of physical or logical injuries for people.

Before starting the development, I have acquainted with the standards and methodologies of automotive software development. In addition it was necessary to expand the scope of my knowledge about memory structures of embedded microcontrollers. I got detailed knowledge about fault-tolerant storage, especially the features of the software modules that used by the company to reach non-volatile memories.

After all the theoretical information gathering and designs phase, I made the memory analysis software developed in JAVA which has extended with a HTML report generator used Template Expression of Xtend programing language. The memory analysis software has integrated as plugin to the Eclipse based integrated development tool called AUTOSAR Architect. This IDE is developed by thyssenkrupp company.

I performed modular testing on well distinguished functional units, which enabled to check the fulfilment of my requirements defined in the design phase.

1 Bevezetés

A XXI. századi technológiai fejlettség ismeretében teljes mértékben elfogadott a mai modern járművekkel szemben támasztott minőségi, biztonsági és kényelmi követelményeink növekedése. A mindenkori rendelkezésreállás és megbízhatóság már szinte az összes, kereskedelemben fellelhető jármű alapvető tulajdonságai közé tartozik. Emiatt a gyártók közti versenyben az egyéb szolgáltatások színvonalában lehet előnyre szert tenni.

A növekvő igények kielégítése érdekében növekszik az autó, mint elektronikus rendszer és az egyes vezérlőegységek komplexitása is. Ami kézenfekvő, ha a mai modern járművekben található többmagos, többprocesszoros rendszerekre, vagy a vezérlőegységek számára gondolunk. Ezek alapján könnyen tekinthetünk a járműre, mint olyan elosztott rendszerre, melyben minden vezérlőegység csak a jól definiált feladatával foglalkozik. A feladatuk ellátásához szükséges adatok közül a funkciójukhoz szorosan kötődőket mérik és számítják. A többi információt, a többi –autóban helyet kapó- vezérlőegységtől szerzik meg. Ezzel a megoldással komoly mennyiségű szenzor és számítási kapacitás takarítható meg.

A járműiparban alkalmazott beágyazott vezérlőegységek (ECU) a dedikált feladataik végrehajtásához szükséges adatok és rendellenes működés esetén öndiagnosztikai információk tartós tárolására is fel vannak készítve. Memória tekintetében tipikusan tartós adattárolásra képes, EEPROM és Flash alapú memóriákat alkalmaznak. A járművek tesztjei során előfordulhat, hogy a már beszerelt vezérlőegységgel kapcsolatban problémák lépnek fel, a hibajelenség kivizsgálása rendszerint a beszállítók feladata. Ezekben az esetekben, a vezérlőegység diagnosztikai információi nagy segítséget nyújtanak a hiba okainak feltárásához. Sarkalatos helyzetekben előfordulhat, hogy az ECU olyannyira megsérül, hogy hagyományos módon nem lehet elérni a memóriában tárolt diagnosztikai információkat. Ezzel szemben, a vezérlőegység megbontásával a tartós adattárolók tartalma még kinyerhető. A hibatűrő adattárolási módszerek miatt, a memóriában található adathalmaz értelmezése nem teljesen egyértelmű feladat, komolyabb erőbefektetéssel járó elemzést igényel.

Jelenlegi feladatom, egy olyan desktop alkalmazás, amely képes a memóriában található adathalmaz elemzésére, kiértékelésére és informatív formában való megjelenítésére. Ezzel segít a fejlesztők hatékonyságát növelni a vezérlőegységek memóriájában található adatok értelmezéséhez.

Jó megoldás a cégnél fejlesztett Eclipse alapú fejlesztőkörnyezet kiterjesztése az erre a feladatra készített plugin alkalmazással. Az ilyen kiterjesztések implementálására a magaszszintű és temérdek előre megírt funkciót megvalósító könyvtárt tartalmazó JAVA nyelv ad lehetőséget.

Az alkalmazásnak képesnek kell lennie a bináris állományban kimentett memóriatartalom és memóriamenedzsment-szoftvermodulok konfigurációs paramétereinek kezelésére. Célszerű, hogy könnyen kezelhető és interaktív felületen jelenítse meg az elemzés eredményeit platform független formában. Ezeknek az elvárásoknak a HTML riportfájl minden szempontból megfelelő megoldással szolgál.

Fontos elvárás, hogy az elemzés eredményei hitelesnek tekinthetők legyenek, vagyis ne fordulhasson elő valótlan információ szolgáltatása. Ehhez szükséges az elemző funkciókat megvalósító részegységek moduláris tesztelése. Ezzel minimalizálva a szoftver hibás működésének esélyeit.

2 Beágyazott mikrovezérlők memóriastruktúrája

2.1 Flash memória áttekintés

A Flash memória egy nem felejtő (non-volatile) típusú informatikai adattároló technológia, amely elektronikusan törölhető és újraprogramozható. A Flash az EEPROM (Electrically Eraseble and Programable ROM, melyről később még szó lesz) egy speciális változata. Az EEPROM és a Flash egyik legfontosabb tulajdonsága, hogy nincs szükségük állandó tápellátásra ahhoz, hogy a benne tárolt adatokat megőrizzék, innen kapták a nem felejtő, tartós tároló jelzőt. A Flash memóriákat két típusra oszthatjuk: NAND és NOR. A NAND típusú memóriák soros hozzáférést tesznek lehetővé, ezért kizárólag kisebb-nagyobb (n * Kbyte) blokkokban olvashatók és írhatóak, ezzel szemben a NOR típusú memóriák párhuzamos hozzáférést tesznek lehetővé, ami miatt a lefoglalható legkisebb alapegységenként (n * Byte) lehet írni és olvasni őket. A Flash tulajdonságai közé tartozik továbbá, hogy nem tartalmaz mozgó alkatrészeket (a merevlemezektől eltérően), ami lehetővé teszi, hogy jobban ellenálljon a mechanikai behatásoknak, mint például a rázkódás. Ennek következtében kiváló adattárolást biztosít beágyazott rendszerekben.

2.2 Flash és EEPROM összehasonlítása

A Flash memória ahogy már korábban is említettem az EEPROM egy speciális válfaja. Amikor Flash-ről beszélünk NAND technológiával készült memóriát értünk, amikor EEPROM-ról akkor pedig NOR kapukat alkalmazó memóriáról. Mi is ezek között a technológiák között a különbség. A kapu technológia miatt a Flash lassabb, viszont nagyobb kapacitású (jellemzően több 10-10.000 kilobájtos a beágyazott rendszerekben használatos processzorokban), mint a néhány 10-100 kilobájtos kapacitású EEPROM. A technológia maga után vonja, hogy gyártási szempontból az EEPROM előállítása drágább mint a Flash-é. Az EEPROM bájtonkénti hozzáférést és újraírást, míg a Flash csak blokkonkénti hozzáférést biztosít. Emellett, a Flash-nél kizárólag csak a blokk törlése után tudjuk az adatot újraírni. A Flash életciklusa alatt nagyságrendileg 100-10.000 újraírást, az EEPROM több mint 1.000.000 újraírást képes hibamentesen elviselni. A két tárolótípus tulajdonságainak köszönhetően az ipari alkalmazások a Flash-t ROM-ként használják programkód, vagy nagy mennyiségű

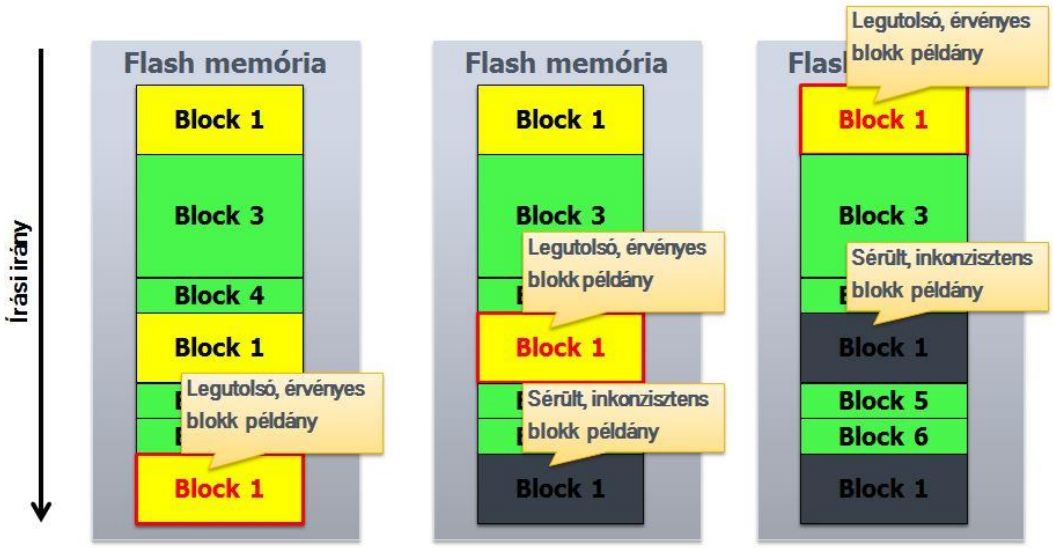
alkalmazásadat számára, amelyeket ritkán kell felülírni. Ezzel szemben az EEPROM-ot a RAM kiterjesztésére alkalmazzák az eszköz kikapcsolt állapotában is megőrizendő alkalmazás adatok tárolására. [3]

2.3 Hibatűrő adattárolás

Egy általános módszer a Flash memóriában tárolandó adatok kezelésére, hogy úgynevezett „lap”-ba foglalunk egy vagy akár több Flash szektort (legkisebb törölhető egység). Az eltárolandó adataink újabb és újabb példányait (blokkokba szervezve) ebbe a „lap”- ba írjuk folyamatosan amíg az be nem telik. Amint a „lap” betelik az adataink legutolsó példányait kiolvassuk, töröljük a lapot és az elejére visszaírjuk az előbb kiolvasott példányokat, majd később kiírjuk az újabb példányokat míg a lap újra be nem telik. Az egész eljárás célja a Flash memória egyenletes terhelése.

A Flash memóriában letárolandó adatblokkok integritásának ellenőrizhetősége érdekében szükséges a blokkadat mellett a blokk azonosítását és állapotát leíró blokk-menedzsmentinformáció tárolása is. Mivel ezeknek a blokk-menedzsmentinformációknak a konzisztenciája nagyon fontos a helyes működés biztosítása érdekében ezen információkat külön védelemmel kell ellátnunk. Erre a feladatra számos megoldás alkalmazható. Letárolhatjuk a blokk-menedzsmentinformációt redundánsan az eredeti invertált változatával, viszont ez felesleges memória használatot jelentene. Továbbá, elláthatjuk a blokk-menedzsmentinformációt XOR, vagy egyszerű ellenőrző összeggel, illetve CRC-vel, attól függően, hogy pl. a Flash memória milyen beépített adatvédelmi mechanizmusokkal rendelkezik. Az előbbi feladat elvégzésére alkalmazni lehetne még magasabb szintű konzisztenciaellenőrző eljárásokat is, amelyek lehetőséget biztosítanak a hiba vagy hibák helyének detektálására és javítására.

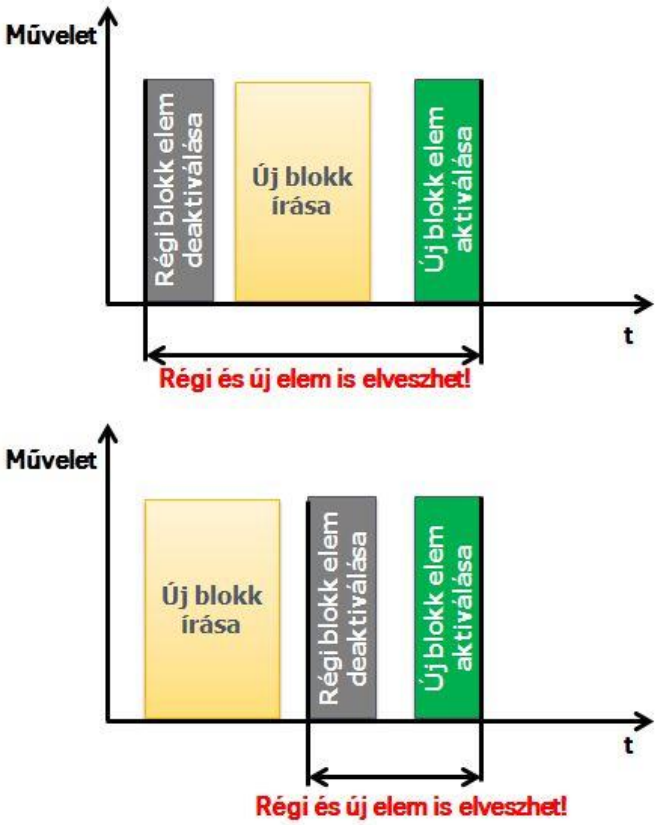
A blokkos adattárolás másik fontos feladata az egyes blokkokból a legújabb, legfrissebb információt tartalmazó blokk megtalálása. Erre két különböző módszer is alkalmazható az adattárolást használó alkalmazás igényei szerint. Ha szükséges az, hogy minden egyes blokkból lehetőség szerint a régebbi információt is tároljuk arra az esetre ha a legújabb blokk sérülne akkor egyszerűen egy algoritmus segítségével mindig meghatározzuk a legutolsó nem sérült blokk pozícióját, ahogy az 1. ábrán látható.



1. ábra - Legutolsó érvényes blokk példány kiválasztása egy hibátlan és két hibás esetben

Másik megoldás a legutolsó érvényes blokkpéldány felismerésére, ha a korábban már ismertetett blokkleíróban egy „Érvényes” és egy „Elavult” flag-et definiálunk, amelyek alapértelmezetten törölve vannak. Amikor egy új blokk kerül kiírásra a Flash memóriában, akkor az „Érvényes” flag-et beállítjuk és ha van a blokkból egy korábbi példány, akkor annak az „Elavult” flag-jét is beállítjuk. Ezzel elérjük, hogy két logikai vizsgálattal (a két flag) meghatározható egy blokk érvényessége. Ezáltal könnyen megtalálható a legutolsó érvényes blokkpéldány, nem kell végigmenni az összesen.

Ez a megoldás sem teljesen veszélytelen ha nem megfelelően választjuk meg az új blokk írásánál elvégzendő műveleteket. A következő ábrán két ilyen esetet láthatunk, művelet-idő diagramon ábrázolva.



2. ábra - Új blokk írás veszélyei

Jól látszik az ábrán, ha a felső műveletsorrendet alkalmazzuk nagyságrendileg nagyobb a valószínűsége, hogy esetleges áramkimaradással járó zavar adatvesztést okozna. Ezért értelemszerűen a hiba valószínűségének minimalizálása érdekében célszerű az alsó műveletsorrendet választani az új blokk írásakor.

Ennek a módszernek, hogy folyamatosan egymás után írjuk az új blokkokat a lapba része, hogy mi a teendő amikor a lap betelik. Logikus döntés lehet, hogy a lap tartalmát átmásoljuk a RAM-ba, ezután töröljük a lapot és a régi, elavult blokkok elhagyásával visszaírjuk a lap tartalmát. Ezzel felszabadítva a teljes területet. Ennek a módszernek a veszélye az, hogy egy ilyen törlő művelet közben, amikor a RAM-ban tároljuk az összes adatunkat, a tápfeszültség elvesztése az összes adat elvesztésével járhat.

Ennek a veszélynek az elkerülése érdekében érdemes két azonos méretű lapot fenntartani a tárolás céljára. Normál működés közben mindig csak az egyik lapot használjuk az előbb ismertetett blokkos adattárolási módszerrel, viszont amikor a lap betelik másként járunk el. A betelt lap legutolsó érvényes blokkjait nem a RAM-ba, hanem a másik, még teljesen üres lapba másoljuk át. Jogos lehet a feltevés, hogy miért

ne definiáljunk több ilyen lapot, viszont egyszerű belátni, hogy minél kisebb egységekre osztjuk a memóriát, annál kisebb kihasználtsággal használjuk normál működés esetén, mivel egyszerre mindig csak egy lapra kerülnek be az új információkat tartalmazó blokkok. A Flash memória terhelését/öregedését növelő törlések számának egyenletes elosztása érdekében az egyes lapváltásokhoz érdemes a fenti Round-robin (körforgó) algoritmust alkalmazni.

3 AUTOSAR áttekintés

A beágyazott szoftverek komplexitásának növekedésével egyre nagyobb problémát jelentett a szoftverek hatékony specifikálása, implementálása, tesztelése és integrálása. A hatékonyság növelése érdekében az autógyártók a szabványosítást tartották a legmegfelelőbb megoldásnak az általános funkciókat megvalósító saját platform szoftvercsomagok helyett. Ezért 7 európai autógyártó és beszállító megalapította az AUTmotive Open System ARchitecture (AUTOSAR) [4] konzorciumot, amely ma több száz taggal rendelkezik. Legfőbb célja az autógyártók és beszállítók közti együttműködés könnyítése a beágyazott szoftverarchitektúra, illetve a szoftvermodellezési és integrációs technológia egységesítésével.

A szabványcsalád fő elemei a komponensorientált modellezési nyelv, a gazdag alapszoftver könyvtár és a fejlesztési folyamat leírása.

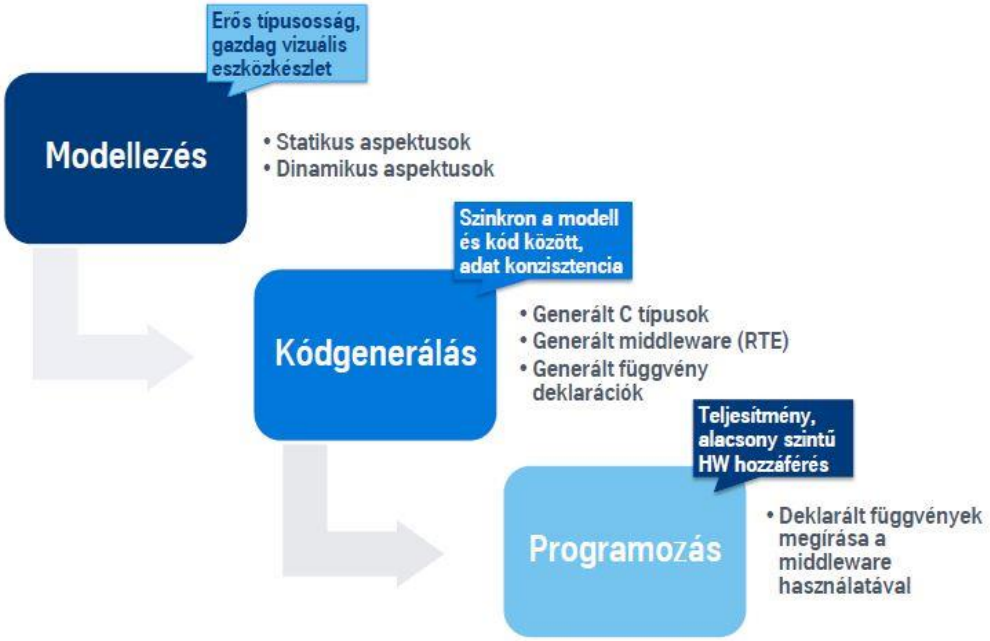
3.1 Komponensorientált modellezés

A komponensalapú modellezési nyelv egyik legfőbb feladata a komplexitás uralása, amelyet a szoftver strukturált felépítése tesz lehetővé.

A funkciókat megvalósító függvényeknél fontos a műveletek egységbezárása, függvényhierarchiák kialakítása, melyek tipikusan a forráskódban néhány tíz soros formában állnak rendelkezésre. A műveleteket végző függvények nem rendelkeznek saját állapottal, kizárólag adat be- és kimeneteket tartalmaznak.

Az adatok és műveletek egységbe zárásához alkalmazott osztályok, már rendelkeznek állapotokkal, örökléssel megvalósított újrahasznosíthatósággal, tipikusan néhány adattag és függvény formájában jelennek meg a forráskódban egy nyelvi egységet alkotva. Ezzel lehetővé téve, hogy a belső adatok a modell részeivé váljanak.

Egy rendszer valamely moduláris részének egységbe zárásához komponensek definiálása szolgál. Több komponens összekapcsolása interfészeken keresztül, segíti az újrahasoníthatóságot lényegében módosítások nélkül. A komponensek tipikusan több osztály vagy függvény, illetve néhány port és interfész formájában jelennek meg.



3. ábra - Komponens alapú szoftver felépítése [5]

A strukturált szoftver felépítéséhez támasztott kategorikus követelmények kielégítéséhez fontos szempont a programozási nyelvek által nyújtott eszközkészlet. A jól ismert Java, és C# magas szintű programozási nyelvek logikusnak tűnhetnek a feladat elvégzéséhez. Nyelvi elemeik teljes mértékben kielégítik a szükségleteinket (osztályok, adattagok, interfészek, stb.), ezzel szemben teljesítményük tipikusan nem elegendő a beágyazott rendszerekben. A függvényeket, osztályokat és névtereket tartalmazó C++ programozási nyelv teljesítménye már alkalmassá tenné beágyazott fejlesztésekre, viszont komplexitását tekintve nem alkalmazzák biztonságkritikus autóipari rendszerekben. Végül, a „kizárólag” függvényeket tartalmazó C programozási nyelv teljesítmény szempontjából megfelelő a fejlesztésekre, viszont számunkra fontos nyelvi elemeket nem tartalmaz. Nincs se osztályfogalom, se névtér.

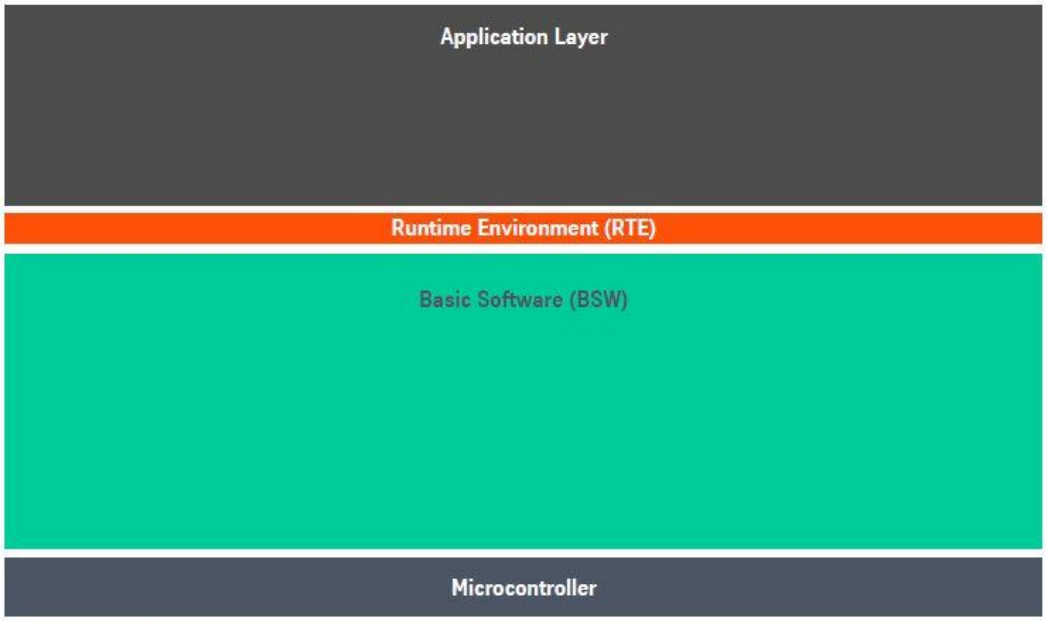
A modellezési és programozási nyelv közötti szakadék áthidalásához használjuk a gazdag modellezési eszközkészletet, de programozzunk C-ben és a hiányzó elemeket helyettesítsük konvenciókkal és kódgenerálással. Grafikus modellezési nyelvben definiálhatjuk statikusan a komponenseket, portokat, interfészeket, adattípusokat, kapcsolatokat, dinamikusan a futtatható entitásokat és eseményeket. Kódgeneráláshoz szükséges: a modell adattípusainak leképezése C adattípusokra, a modell komponensei közti kommunikációt megvalósító middleware (RTE-RunTime Enviroment, lásd 2.2. fejezet), illetve az elnevezési konvenciók szerint generált C függvény deklarációk, melyeket kézzel kell megírni. Végezetül a futtatható entitásokat megvalósító, a

kódgenerátor által deklarált függvények megvalósítása az RTE middleware szolgáltatásai alapján.

Ezen szabályok betartása mellett, jó teljesítményű szoftvert fejleszthetünk modellezetten.

3.2 AUTOSAR szoftverarchitektúra

A 4. ábrán látható az AUTOSAR rétegzett architektúrája [6] legfőbb egységekbe rendezve.



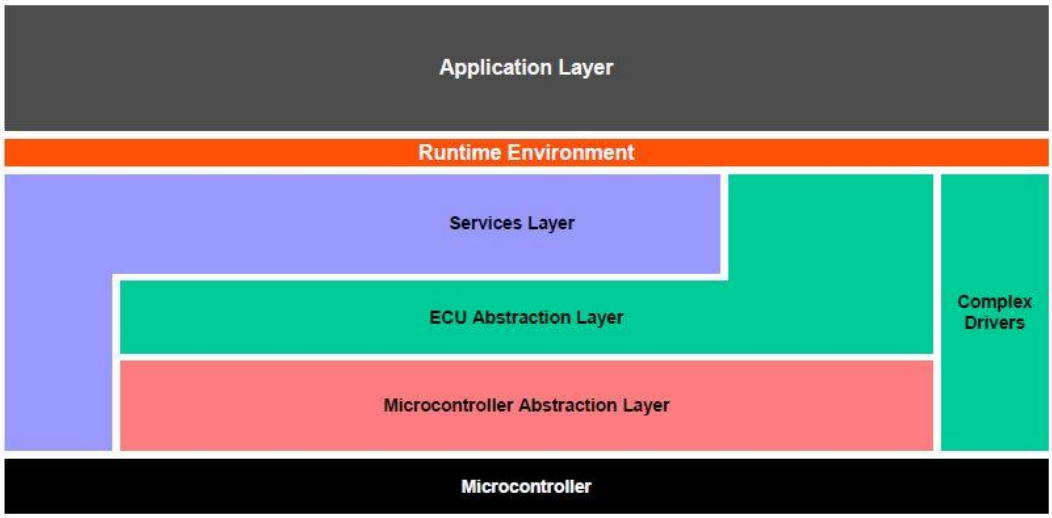
4. ábra - Rétegzett szoftver architektúra [6]

A legfelül elhelyezkedő alkalmazási rétegben helyezkednek el a „hasznos” funkciókat megvalósító AUTOSAR szoftverkomponensek. Közvetlenül alatta a futtató környezet (RTE-RunTime Enviroment). Ez a réteg valósítja meg a komponensek közötti kommunikációt, valamint a komponens és a BSW (Basic Software) réteg által nyújtott magas szintű szolgáltatások közti kapcsolatot. Ezáltal a komponensek megvalósítása teljesen függetlenné válhat az őket hordozó vezérlőegység jellemzőitől. A Basic Software réteg feladata az alkalmazás réteg kiszolgálása az RTE-n keresztül.

A dolgozatom témája a BSW réteget alkotó szoftvermodulokhoz kapcsolódik, ezért a következő fejezetekben ennek a rétegnek a felépítését mutatom be, részletesen kitérve a memóriakezelést végrehajtó modulokra.

3.2.1 Basic Software réteg

Az AUTOSAR Basic Software rétege lényegében egy szabványos interfészekkel rendelkező, moduláris felépítésű függvénykönyvtár, melynek belső alrétegek szerinti felépítését az 5. ábra részletezi. A réteg feladata a mikrovezérlő szolgáltatásainak elérhetővé tétele az alkalmazásréteg komponensei számára. Gazdagon konfigurálható modulok alkotják, melyeknek működése az adott vezérlőegység jellemzőinek, és a vevői követelményeknek megfelelően testre szabhatóak. A BSW modulok többsége egyaránt tartalmaz „kézzel írt” (statikus), valamint az aktuális követelményeknek megfelelő konfigurációs modell alapján generált forráskódot. Ezen kódrészletek aránya mindkét véglet felé elcsúszhat az adott modul konfigurálhatóságának megfelelő mértékében. Modellalapon konfigurálunk és kódot generálunk.



5. ábra - BSW réteg architektúrája [6]

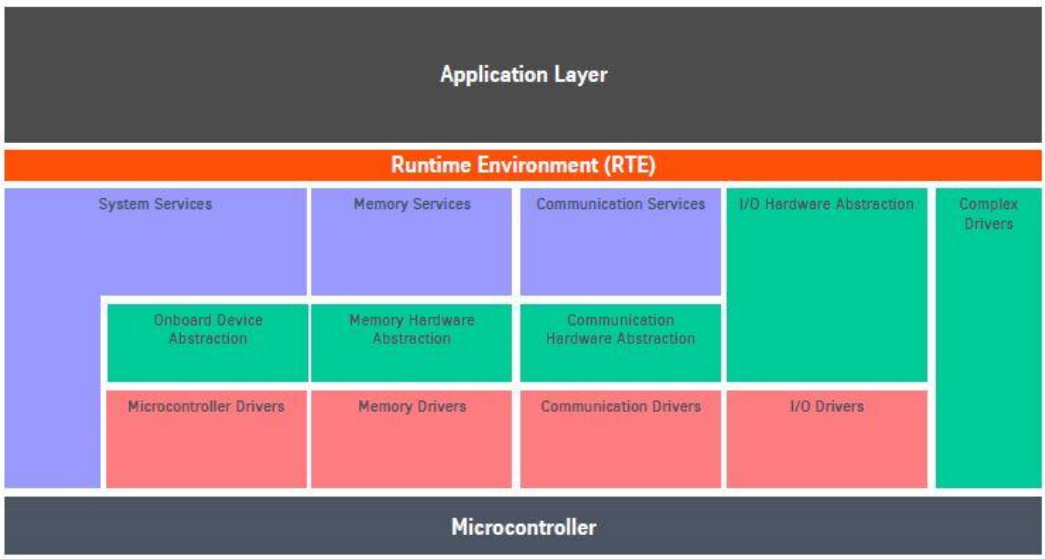
Ahogy az ábrán is látható a BSW réteg legalsó alrétege a Mikrokontroller Absztrakciós Réteg. Ez a mikrokontroller perifériáihoz való hozzáférést biztosító driver modulokból áll. Például: MCU Driver, CAN Driver, FlexRay Driver, Flash Driver, stb. Feladata a felette található ECU Absztrakciós réteg moduljai számára mikrokontrollertől független hozzáférést biztosítani annak funkcióihoz és perifériáihoz. Ennek a rétegnek a megvalósítása erősen mikrovezérlő-típusfüggő, míg az általa nyújtott interfész szabványosított mikrovezérlő-független.

Az ECU Absztrakciós alréteg az alatta található driverek szolgáltatásaira támaszkodva nyújt hozzáférést a vezérlőegység funkcióihoz, függetlenül attól, hogy a funkciót megvalósító periféria épp belső vagy külső. Ez az alréteg tartalmazza az

esetleges külső perifériákat meghajtó driver modulokat is. Feladata, hogy a Szolgáltatási alréteg moduljai számra a vezérlőegység felépítésétől független interfészt biztosítson annak funkcióihoz. Az alréteg megvalósítsa, erősen függ a hardver felépítésétől, míg az általa nyújtott interfész szabványosított és független a vezérlőegység konkrét megvalósításától.

A BSW legfelső alrétege a Szolgáltatási alréteg, melynek feladata, hogy alapvető szolgáltatásokat nyújtson az RTE-n keresztül a szoftverkomponensek számára. Például: Operációs rendszer funkciók, Memória menedzsment, Vezérlőegységek közötti kommunikáció és hálózatmenedzsment, Diagnosztikai funkciók, Vezérlőegység állapotmenedzsmentje stb. a Szolgáltatási Réteg moduljainak megvalósítása és az általuk az Alkalmazás Réteg felé nyújtott interfész teljes egészében független a futtató hardvertől.

Végezetül a Komplex Driver egy olyan BSW modul, mely az AUTOSAR által nem definiált funkcionalitást valósít meg. Az AUTOSAR ezen driverek segítségével teremti meg a lehetőséget arra, hogy a speciális (nem szabványosított, vagy nem szabványosítható) funkciókat megvalósító perifériák szolgáltatásaihoz is hozzáférhessen az alkalmazás. A Komplex Driverek a rétegzett architektúrát átugorva felül közvetlenül az RTE-vel, míg alul közvetlenül az adott perifériával állnak kapcsolatban.



6. ábra - BSW stack funkcionális felosztása [6]

A 6. ábrán láthatók BSW réteg funkcionális stack-jei, melyből a jelenlegi feladatom szempontjából a tartós memória használatát megvalósító szolgáltatások, HW

Absztrakciós réteg, Driver-ek fontosak. Ezekben a csoportokban fognak elhelyezkedni a mikrovezérlő belső Flash memóriájának elérésére szolgáló modulok.

3.2.2 BSW modulok tulajdonságai

Az AUTOSAR BSW [5] modulokkal szembeni követelményeket az adott modulhoz tartozó AUTOSAR specifikáció tartalmazza. Például, a feladatomhoz kapcsolódó:

AUTOSAR_SWS_FlashDriver.pdf, AUTOSAR_SWS_FlashEEPROMEmulation.pdf és AUTOSAR_SWS_NVRAMManager.pdf.

Ezeknek a dokumentumoknak a felépítése egységes, fontosabb fejezeteik a következőek [7]:

Más modulokkal kapcsolatos függőségek (5. Dependencies to other modules) fejezetben az van leírva, hogy az adott modul milyen további moduloknak milyen szolgáltatásaira épít és milyen interfészeket vár el.

Funkcionális specifikáció (7. Function specification) fejezet a modul részletes belső működésének és felépítésének leírását tartalmazza.

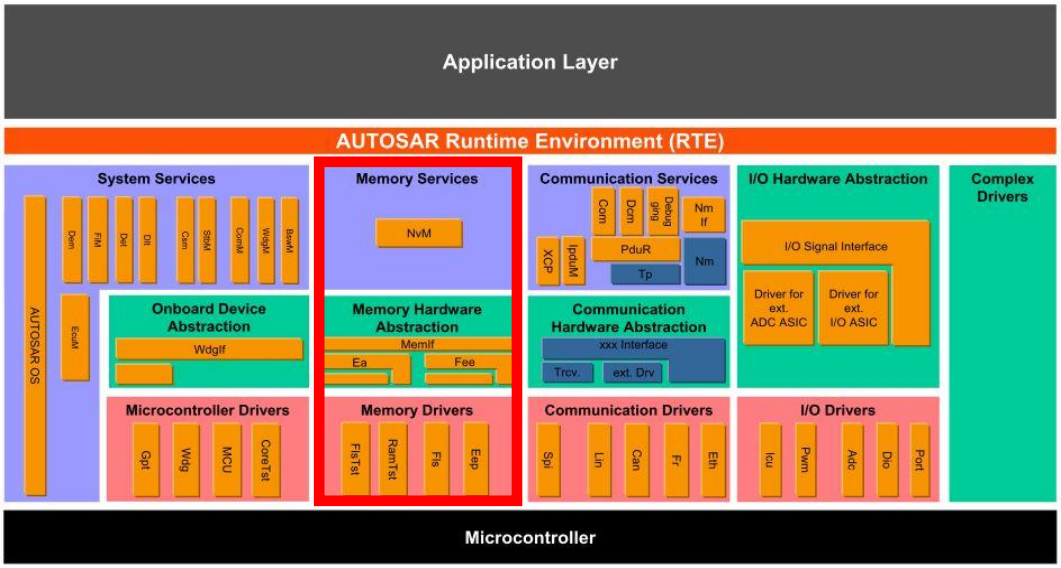
API specifikáció (8. API specification) fejezetben a modul által nyújtott API függvények és típusok meghatározása olvasható.

Konfiguráció specifikáció (10. Configuration specification) fejezet tartalmazza a modul konfigurálhatóságának szempontjából lényeges előírásokat.

Az AUTOSAR BSW fejlesztést a V modell alapján 3 fő területre lehet osztani: a követelmények analízise, az implementáció és a tesztelés. A fejlesztés első lépésének a bemeneteként az AUTOSAR specifikációban taglalt követelmények szolgálnak, illetve az ebben referált szabványok. A fejlesztés első lépéseként ezeket a követelményeket a fejlesztési folyamatnak megfelelő módon egy „.xml” kiterjesztésű fájlban összegyűjtve kategorizáljuk. A követelmények klasszifikálásának követéséhez használt dokumentumban az egyes követelményekről el kell döntenünk, hogy elfogadjuk-e a követelményt, ez a követelmény implementálható-e a forráskódban, illetve hogy tesztelhető-e. Ezeknek az információknak a nyilvántartása fontos a fejlesztéshez, mivel az itt elfogadott követelmények alapján tudjuk az implementáció és a teszt helyességét ellenőrizni.

A fejlesztés lényeges részét képzi a kézzel megírt kódok elkészítése. Itt komoly figyelmet kell fordítanunk a specifikáció 10. pontjában szereplő konfigurációs előírásokra. A szoftver széles konfigurálhatósága fontos, mivel ezzel a modul működését hangolhatjuk az aktuális felhasználási módhoz, amely magában hordozza, hogy több alkalmazás számára is felhasználható modult kapunk. A konfigurációs paraméterek összegyűjtésére arxml kiterjesztésű modell leíró fájlokat használunk. Ezek az xml modellezési nyelv gazdag eszközkészlete révén kiválóan egységbe zárják az egyes szoftvermodulokat a szükséges paramétereikkel együtt.

4 AUTOSAR memóriaverem

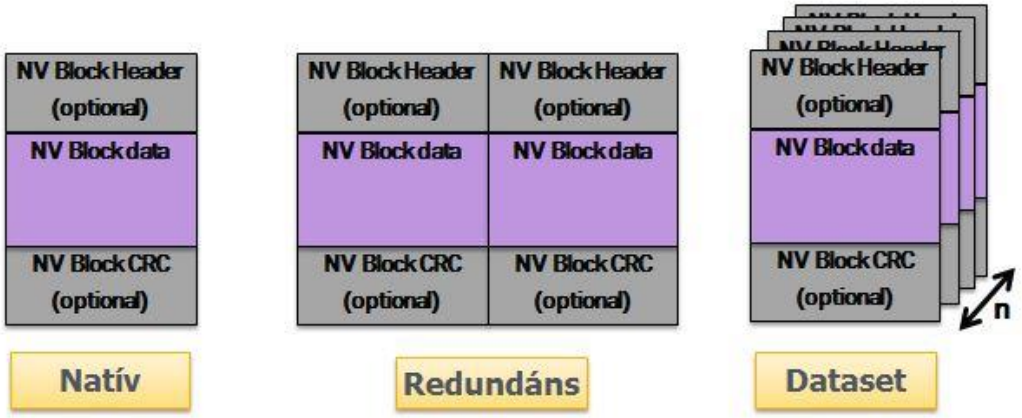


7. ábra - Memória stack az AUTOSAR-ban [6]

4.1.1 NVRAM Manager

Az NvM Manager [8] (Non-volatile Memory Manager) feladata a memóriaveremben egységesített hozzáférés biztosítása az alkalmazásrétegben elhelyezkedő szoftverkomponensek és más BSW (Basic Software) modulok számára. A feladatát blokkszintű adatkezeléssel valósítja meg. Ez azt jelenti, hogy az írás, olvasás és invalidálás (törlés) műveleteket adatblokkok által egységbe zárva végzi el. Az egyes adatblokkok integritásának védelmére, az egyedi azonosító ellenőrzése, a blokk CRC és a redundancia alkalmazható, az egyes blokk-konfigurációk szerint. A blokkok fizikai elhelyezkedéséért kizárólag az alsóbb rétegekben elhelyezkedő modulok felelnek.

Az egyes NvM blokkok lehetnek natív, redundáns és dataset típusúak, ahogy a 8. ábrán látható.



8. ábra - NvM blokk típusok

Ezen blokkok mindegyike opcionálisan rendelkezhet blokk-menedzsment információval, melyet az NvM a blokk adattal együtt tárol és az adott blokk magas fokú integritásának ellenőrzésére szolgál. A blokkadatot megelőzve tárolhatja az NvM a blokk egyedi azonosítóját (NvMNvramBlockIdentifier), amelyet a blokk egyértelmű azonosítására használhat, ha ezt a magas szintű védelmet a blokkot használó felsőbb réteg megköveteli. Mivel ez opcionális ellenőrzés, ezért ezt a blokk konfigurációjában engedélyezettnek vagy tiltottnak kell beállítani a megfelelő (NvMStaticBlockIDCheck) paraméterrel. Az NvM blokk (a blokk végén) rendelkezhet CRC mezővel az adatok sérülésének ellenőrzéséhez. Ez a CRC mező elhagyható (NvMBlockUseCrc), illetve 8, 16 vagy 32 bites szélességgel (pl. a blokk méretének megfelelően) konfigurálható.

A blokkok közül a natív a legegyszerűbb, mivel ebben az opcionális mezők mellett kizárólag az eltárolandó adat szerepel. Ezzel ellenkezőleg a redundáns blokkban az adat védelmének érdekében kétszer tároljuk le ugyanazt az adatot, ezzel az opcionális védelemmel egy biztonságosabb adattárolást megvalósítva. A dataset típusú blokkokat tekinthetjük blokktömbnek. A tömbelemek maximális számát az „NvMDatasetSelectionBits” konfigurációs paraméterrel állíthatjuk be, amelynek konkrét értékét a következő képlet adja meg. A képletben a k változó, az adatkiválasztó bitek számát, az n változó pedig a dataset blokktömb maximális elemeinek számát jelöli. A képlet a blokktömbök száma alapján meghatározza, hogy az aktuális k darab bittel, maximálisan n darab blokkra van lehetőség.

$$n = 2^k - 1$$

Az NvM blokk konfigurációjában a „NvMNvBlockBaseNumber” paramétert úgy kell megadni, hogy a hozzá tartozó Fee blokkazonosító számát kapjuk meg, a következő egyenlet segítségével.

$$feeId = bNum \ll dselNum$$

Az egyenlettel az NvM és az FEE blokkok közötti egyértelmű kapcsolat határozható meg, amelyet minden esetben a két modul blokkjainak konfigurálásánál szem előtt kell tartani. Az NvM blokkhoz tartozó FEE blokk azonosítószámát (feeId) úgy kapjuk meg, hogy az NvM blokk bázisszámát (bNum) a blokk adatkiválasztó (dselNum) bitjeinek számával „shifteljük” balra.

4.1.2 Flash EEPROM Emulation

Az FEE [9] (Flash EEPROM Emulation) modul feladata, hogy a belső (on-chip) és külső (on-board) Flash memóriák ’kényelmetlen’ viselkedését elfedje és az NvM felé egy blokkokat kezelő absztrakciós felületet nyújtson a Memory Abstraction Interface-en keresztül, mely interfész segítségével az NvM bármilyen technológiájú nem felejtő memóriát adatblokkszintű hozzáféréssel kezelhet.

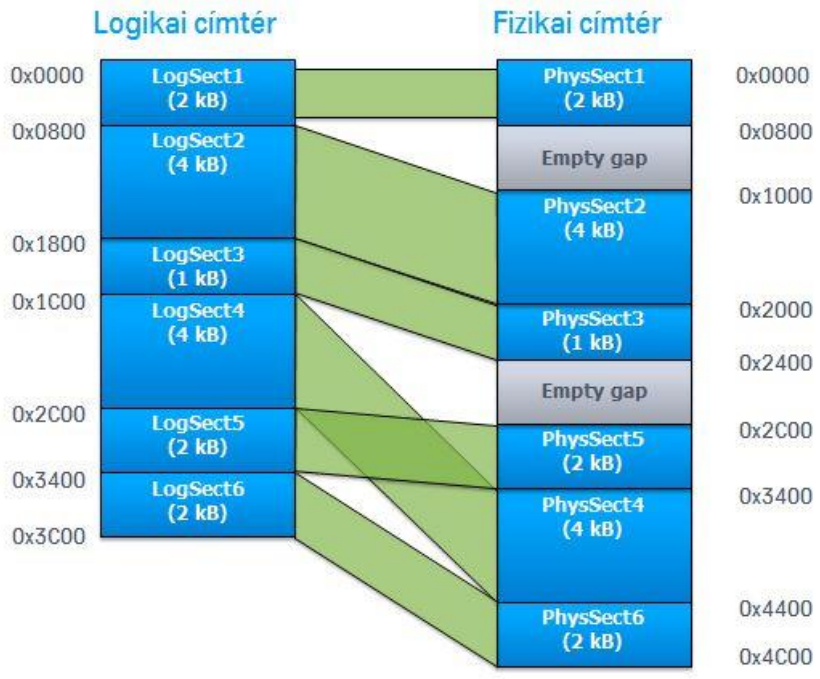
Mivel a Flash memóriák fizikailag eléggé korlátozott számban törölhetőek, ezért az FEE feladata a törlési ciklusok minimalizálásának biztosítása és a törlések számának egyenletes elosztása a Flash memória teljes területén.

Az NvM-hez hasonlóan az FEE is blokkszintű adattárolást valósít meg, viszont a **2.3.-**as fejezetben leírtaknak megfelelően blokk-menedzsmentinformációt kell egy FEE blokkhoz nyilvántartani és azzal együtt eltárolni. A blokk-menedzsmentinformáció tárolja (1) a blokk azonosítására szolgáló blokkszámot (FeeBlockNumber), (2) a blokk állapotot, amelynél a KORREKT, SÉRÜLT és a TÖRÖLT állapotokat különböztetjük meg. Abban az esetben, ha a blokk-menedzsmentinformáció és a blokk adat nem összefüggő területen van tárolva, akkor szükség van az adat kezdőcímre is. A blokkinformáció védelme érdekében, a blokk-menedzsmentinformációt védelmi mechanizmussokkal láthatjuk el.

Az FEE feladatainak megvalósítása jelentősen implementáció specifikus, amely a Thyssenkrupp Components Technology Hungary kft. érzékeny üzleti termékét képzi, ezért az FEE implementációját nem ismertetem részletesen.

4.1.3 Flash driver (FLS)

A Flash driver [10] faladata az, hogy egybefüggő virtuális címzést biztosítson egy báziscímtől kezdve a felsőbb rétegben található modulok, mint például az FEE számára és ezzel elfedje a mikrovezérlőbe integrált, és/vagy külső Flash memória címzési sajátosságait. Ez azért szükséges mert a különböző mikrovezérlőkben más-más címterületeken érhetjük el az egyes fizikai szektorokat, és ezek a szektorok nem feltétlen egybefüggő területen találhatóak a fizikai memóriában. A Flash Driver az előzőekben részletezett módon függetleníti az Fee modult a fizikai címtér bonyolult kezelésétől, mindösszesen flash terület méretéhez kell az Fee modulnak alkalmazkodnia. Ezen túl az FLS végzi a fizikai memória kezelését (törlés, írás, olvasás), regiszterszintű hozzáféréssel. A legtöbb esetben az olvasás közvetlenül a CPU memóriabuszának segítségével egy egyszerű olvasás művelettel is elvégezhető, de vannak esetek, amikor az olvasás csak regisztereken keresztül kérhető (a flash memória „lassúsága” miatt), majd a beolvasott adat egy dedikált RAM területről kiolvasható.



9. ábra - Logikai és fizikai címtér összerendelés

A 9. ábra láthatóak az FLS konfigurációjában meghatározott egybefüggő címterét alkotó logikai szektorok és a hozzájuk tartozó, az integrált Flash memóriában található fizikai szektorok közötti kapcsolatok. Rendszerint ezt a címösszerendelést egy egyszerű táblázat segítségével valósítja meg a modul.

Ezek alapján az FEE, úgynevezett lapjait ezekből a logikai szektorokból kell összeállítani a modulok konfigurációja során.

5 Felhasznált technológiák, eszközök

5.1 JAVA

A JAVA [11] általános célra használható, objektumorientált, semleges architektúrájú, egyszerű programozási nyelv. A C++-nál eggyel magasabb szintű és több megkötéssel rendelkezik, ezáltal kevesebb hibalehetőséggel rendelkezik. Erős szolgáltatáskönyvtár-támogatással bír, szinte bármilyen problémához található előre megírt könyvtár. Az alacsonyabb szintű nyelvekkel ellentétben pointerek helyett referenciákat használ, a veremben tárolt objektumpéldányok elérésére. Emellett lehetőséget ad primitív típusok használatára is, amelyek platformfüggetlen bájtmérettel rendelkeznek. Új típusok definiálása nem lehetséges. Az objektumok rendelkezhetnek tulajdonságokkal és az objektumon végezhető műveletekkel. Az tulajdonságok belső adatstruktúraként, a műveletek ezeken az adatokon operáló metódusokként realizálódnak az objektum típusát meghatározó osztály definíciójában. A JAVA teljes mértékben eleget tesz az objektumorientáltság fő ismérveinek.

Az osztályok definiálásánál lehetőség nyílik az információ egységbe zárására és rejtésére. Meghatározható, hogy az osztály változóit és metódusait más osztályok hogyan érhetik el. (public, private, protected)

Az objektumokat minden esetben objektumokból származtatjuk le, ezt nevezzük öröklődésnek. Öröklés esetén az a leszármazott osztály rendelkezik a szülő osztály minden tulajdonságával. Lehetséges egyszeres és többszörös öröklődés is, amely összefüggéseit hierarchikus öröklési fában lehet ábrázolni. Minden esetben az öröklési fa gyökérelemében az ősosztály helyezkedik el, ennek nincs szülő eleme.

Az öröklődés egyik mellékhatása, hogy egy objektum példánya minden olyan osztálynak is, amiből örökölt. Ezáltal a közös ősből származó minden objektum kezelhető a közös ős egy példányaként, ezt nevezzük polimorfizmusnak. Ennek a tulajdonságnak az a legnagyobb előnye, hogy különböző, de azonos ősosztályokból létrejövő objektumok egységesen listába szervezhetőek, vagy metódusoknak átadhatóak függetlenül a saját osztály definíciójuktól, az ősosztály példányainak tekintve őket. [12]

5.2 Xtend

Az Xtend [13] a JAVA egy rugalmas és kifejező dialektusa, amely fordítás után JAVA kódot eredményez. Létrehozásának célja, a JAVA modernizálása, kiterjesztése a hiányzó hasznos funkciókkal. Az Xtend-ben zökkenőmentesen használható minden korábban megírt külső JAVA könyvtár. Minden Xtend nyelvet implementáló forrásfájlból JAVA kód generálódik fordítási időben. Optimalizálása segítségével az Xtend által generált JAVA kód gyorsabb futási időt és szemmel jobban áttekinthető kódot eredményez. Az Xtend nyelvben a számtalan lehetőség közül megvalósítható például: a zárt típusok kiterjesztése új funkcionalitásokkal (Extension methods), operátorok túltöltése (Operator overloading) és sablon kifejezések (Template expressions). Ezek közül a feladatomhoz az olvasható formájú string-ek összefűzésre alkalmas sablonkifejezések funkciót ismertetem részletesebben.

Az Xtend Templates Expression-je lehetővé teszi az olvasható szövegek összefűzését, illetve előre elkészített szövegekből, paraméterezett szöveg előállítását. Ez a funkció nagyon népszerű a kódgenerálási feladatok elvégzéséhez, mert lehetőséget nyújt arra, hogy a kimeneti string-et formázásaival együtt lehet szerkeszteni. Az ilyen metódusokból készült, JAVA realizációk már korántsem tartalmaznak ennyire áttekinthető formát.

A sablon szövegek definiálásához három darab aposztróf (’’’) jellel nyitott és zárt blokkok szolgálnak. Lehetőség van a szövegben paraméterek vagy más szövegek elhelyezésére, a guillements karakterek (« ») által közrezárt blokkok segítségével. A sablon szövegek definiálását a 10. ábra szemlélteti egy példán keresztül.

```
1. def someHTML(List<Paragraph> paragraphs) '''
2.   <html>
3.     <body>
4.       «FOR p : paragraphs»
5.         «IF p.headLine != null»
6.           <h1>«p.headline»</h1>
7.         «ENDIF»
8.         <p>
9.           «p.text»
10.         </p>
11.       «ENDFOR»
12.     </body>
13.   </html>
14. '''
```

10. ábra – Sablon kifejezések példa. [14]

A dinamikus szövegek létrehozásához, lehetőség van feltételek és ciklusok definiálására. Ahogy az ábrán is látható, ciklusok definiálására a FOR és ENDFOR, feltételek meghatározásához az IF és ENDEIF nyelvi elemek alkalmazása szükséges.

Az elemző szoftver kimeneti eredményeinek ismertetésére szolgáló HTML dokumentumok előállítására használt Xtend osztályok a 7.2 fejezetben kerülnek részletezésre.

5.3 HTML

A HTML (HyperText Markup Language) [15] egy strukturált dokumentumleíró nyelv, amit eredetileg weblapok és webalkalmazások készítésére hoztak létre. A HTML nyelv utasításokon (tag) alapul, jellegzetessége, hogy legtöbb utasítása egy nyitó és egy záró elemből áll, az utasítás hatálya a kettő közötti részre terjed ki. Az utasításokat a böngészőprogram értelmezi és végrehajtja. Az egyes utasítások határozzák meg a tag-ek között található tartalmak (szövegek, linkek, képek, animációk, stb.) megjelenítendő formátumát. Az utasítások köre állandóan fejlődik és bővül, legfrissebb szabványosított verziója a HTML5.

Leggyakrabban CSS-el (Cascade Style Sheets) és JavaScript-el társítva készülnek a HTML dokumentumok.

A CSS [16] a HTML 4.0 verziója óta a szabvány része, ez a HTML dokumentumok vizuális megjelenését befolyásoló egyszerű nyelv. Ez azt jelenti, hogy a weblapok HTML elemeinek megjelenítését utasításokból, szabályokból felépített stíluslapokkal (style sheet) lehet definiálni. Az elnevezésében is szereplő egymásba ágyazhatóság (cascading) arra utal, hogy több stíluslap, meghatározás megadása is lehetséges egyszerre egy elemre, illetve egy stílus lehet több elemre is érvényes, ami akár egy másik stílussal felüldefiniálható. A CSS segít a szabványok betartásához a HTML és a CSS utasítások és szabályok elkülönítésére, melynek használata olyan weblapokat eredményez, amik forráskódja könnyen értelmezhető és jól néznek ki. A szabvány lehetővé teszi, hogy a CSS kódot külön fájlba szervezve alkalmazzuk. Ez a megoldás nagy segítséget nyújt a HTML kód tisztántartásában, egyszerűbb szerkeszthetőségben és karbantartásban. Egy külön CSS fájlba szervezett stíluslappal egyszerre akár több dokumentum megjelenítése is befolyásolható.

A JavaScript [17] egy HTML dokumentumokba ágyazható böngészőben futtatható parancs (script) nyelv, amelyet interaktív weblapok készítéséhez hoztak létre. A JavaScript prototípus alapú, objektum orientált, leginkább a böngészőben bekövetkezett események (például: kattintás, billentyűzet lenyomás, stb.) kezelésére alkalmazzák. A JAVA programozási nyelvvel ellentétben a JavaScript-ben nincsenek osztályok, kizárólag objektumok. Ezzel együtt az általános programozási nyelvekhez hasonlóan definiálhatók függvények és összetett adatstruktúrák. A prototípus alap úgy jelenik meg, hogy a függvények konstruktorként szolgálnak, az egyes objektumok létrehozására.

Dolgozatomban az imént ismertetett technológiákból felhasznált eszközöket és megoldásokat a 7.1 fejezetben ismertetem.

5.4 Eclipse

Az Eclipse [18] egy nyílt forráskódú fejlesztő platform ami különböző programozási nyelveken való fejlesztést támogatása mellett (pl. C, JAVA), használható különböző beágyazott fejlesztő eszközök, vagy más komplex összetett szoftverek megvalósítására is. Architektúrája úgy lett kialakítva, hogy az alapvető futtató környezet (ERP – Eclipse Runtime Platform) tartalmazza a hozzá kapcsolódó alkalmazások futtatásához szükséges információkat, frissítések kezelését, stb. Ehhez a platformhoz kapcsolódik a beépített fejlesztőkörnyezet (IDE – Integrated Developement Enviroment). Erre a fejlesztői környezetre lehet integrálni a különböző nyelvspecifikus fejlesztői környezeteket. (pl.: C/C++ és a JAVA). Az IDE-re épül a bővítmények készítésére szolgáló környezet (PDE – Plugin Developement Enviroment).

A bővítmények a rendszerhez hozzáadott alkalmazások kibővítésére, új funkcionalitás hozzáadására szolgálnak. Bővítmények lehetnek fejlesztéshez használt funkciók mint például a verziókezelést megvalósító SVN, vagy a statikus kódvizsgálatot végző SpotBugs, a számtalan további alkalmazás mellett. A bővítmények Extension-ök és Extension point-ok segítségével kapcsolódhatnak a futtató környezethez vagy akár egymáshoz is. Ha egy bővítmény funkcionalitását ki akarjuk terjeszteni, akkor egy Extension point-ot kell definiálnunk, amelyre kapcsolódva a bővítés Extensionnel kapcsolódik és adott interfészen keresztül együttműködik vele. Egyszerre több ilyen kapcsolódási pont is megengedett.

5.5 JAVA verifikációs technikák és eszközök

A szoftver funkcióit végző programkód verifikálása a fejlesztési folyamat egyik fontos feladata, ezért az idő folyamán temérdek eszközt, keretrendszert készítettek a feladat megoldásának segítésére.

A verifikálás első lépéseként érdemes a kódon statikus elemzést végezni, egy erre a feladatra készült segédprogram segítségével. Ilyen programok például a Parasoft Jtest [19], Coverity Scan [20], vagy kifejezetten JAVA kód analízishez készített SpotBugs [21].

A SpotBugs eszköz statikus kódanalízis útján, tipikus, helytelen viselkedést eredményező elemeket detektál JAVA programokban. A lehetséges hibákat négy csoportba rangsorolja: legveszélyesebb, veszélyes, kockázatos, aggodalomra okot adó. Ezek a csoportok szemléltetik az egyes hibák valószínűségét és helytelen viselkedés kiváltására való befolyást. A legnagyobb valószínűséggel, legnagyobb hibát okozó elemek a legveszélyesebb csoportba, a legkisebb valószínűség mellett legkisebb hatást jelentő elemek pedig az aggodalomra okot adó csoportba kerülnek. Problémát okozó eset lehet NULL objektum átadásának lehetősége. Tömbök esetén a túlindexelés, helytelen méret vagy ofszet használata is a keresett hibákat okozó minták közé tartozik a számtalan hibaforrás közül. A SpotBugs plugin lehetőséget ad, modellbe szervezett kivétel lista hozzá adására. Ezzel lehetséges az egyes hibatípusok elutasítása, vagyis a megjelölt mintákat figyelmen kívül hagyja a statikus kódanalízis közben.

A kódelemzés által felfedett hibák javítása után szükséges az egyes funkcióikat megvalósító, egymástól különválasztható részegységek, mint önálló modulok működésének ellenőrzése, tesztelése önállóan a rendszer többi részétől teljesen elkülönítve.

A modultesztek a tesztelő ismeretei alapján három alapvető csoportba -fehér, szürke és fekete doboz- sorolhatjuk.

- Fehér doboz teszt esetén a tesztelendő kód belseje ismert a tesztelés folyamán.
- Szürke doboz teszt esetén a tesztelendő kód egyes részei ismertek csak a tesztelés során pl. azért, hogy a tesztelendő kód runtime változói az egyes tesztesetekben alapértékre állíthatóak legyenek, ha a tesztelendő

kódnak nincs erre egy külön meghívható eljárása. A kód többi része nem látható és csak specifikáció alapján tesztelhető.

- Fekete doboz teszteknél a belső működés részletei egyáltalán nem ismertek, így csak specifikáció alapján tesztelhetőek az ilyen kódok.

Amikor alapvetően a be- és kimenetekre fókuszálunk és a teljes belső struktúra nem ismert, de vannak információk és beavatkozási pontok a normál ki- és bemeneteken kívül is, akkor szürke doboz tesztről beszélünk.

Az egyes modulok teljes körű tesztelése önmagában, egy komplex eszközkészlettel rendelkező, jól definiált keretrendszer nélkül magas emberi erőforrás-igényű feladat. Az erőforrások csökkentése érdekében szükséges egy olyan rendszer, amely segítségével egyszerűen létrehozhatók akár külön futtatható tesztek, amelyek eredményeit jól értelmezhető módon ismerteti. Egy jól használható keretrendszer segítséget nyújt az egyes tesztlépések tesztesetekbe foglalásához és ezek tesztkészletekbe való csoportosításához. Ez a módszer segít a funkcionális alegységek szeparálásában és a teszt strukturáltságának megteremtésében. Alapvető elvárás egy ilyen keretrendszerrel szemben, hogy a tesztlépések eredményeit informatív formában jelenítse meg, azaz valamilyen módszerrel jelezze a teszt sikerességét vagy hiba esetén minél pontosabban tájékoztasson a hiba okáról és helyéről. Ilyenek például Spock [22], Mockito [23], JUnit [24].

5.5.1 JUnit

A JUnit [24] az egyik, ha nem a legnépszerűbb és legtöbbet használt szabad forráskódú modultesztelő keretrendszer, ami JAVA programozási nyelven írt szoftveregységek automatikus teszteléséhez ad segítséget. Fontos tulajdonságai közé tartozik, hogy fejlesztőkörnyezetbe is integrálható mint például az Eclipse, ezáltal használata is leegyszerűsödik. Támogatja a hierarchikus tesztstruktúra kialakítását, a tesztesetek (test case) definiálásával, amelyek tesztkészletekbe (test unit) csoportosíthatók. Lehetőséget ad a tesztekhez úgynevezett előkészítő és lebontó metódusok definiálására, amiket a tesztek végrehajtása előtt illetve után futtat le. Ezen metódusok segítségével a teszthez használt globális változók kezdőértéke minden teszteset lefutása előtt inicializálható, ill. tesztesetek lefutása után alaphelyzetbe állítható.

A JUnit keretrendszer használatával javítható a tesztprogram minősége, ezáltal hibakeresési idő takarítható meg. Önmagában egyszerűbb modulok tesztelésére kiválóan használható, ahol összetettebb objektumok közti összefüggések lépnek fel helyettesítő objektumok kiterjesztésével ajánlott. (Lásd 5.5.2 fejezet)

A keretrendszerben implementált tesztek önállóan futó tesztprogramra fordulnak le. Az egyes funkciókat különböző annotációkkal jelölve különböztetjük meg. Ennek köszönhetően a fordító el tudja dönteni, hogy melyik metódust hogyan kell kezelnie. Ezeket az annotációkat a következő táblázatban gyűjtöttem össze.

@BeforeClass	Az első tesztmetódus előtt végrehajtandó metódus.
@Before	Minden egyes tesztmetódus előtt végrehajtandó metódus.
@Test	Tesztelésére szolgáló metódus.
@Ignore	Nem végrehajtandó tesztmetódus.
@After	Minden tesztmetódus után végrehajtandó metódus.
@AfterClass	Az utolsó tesztmetódus után végrehajtandó metódus.

1. táblázat – JUnit lényeges annotációk

Tesztesetek létrehozásához, először egy tesztosztály létrehozása szükséges. A tesztosztályokban definiálhatók a tesztlépéseket megvalósító metódusok, amelyeket „@Test” annotációval jelölünk. Az annotáció után zárójelben lehetőségünk van a metódus maximális futás idejének megadására. Ha a metódus a futása során túllépi a megadott időintervallumot, akkor kivétellel (exception) megszakításra kerül a teszt. Ezen kívül meghatározható a tesztmetódusok sorrendje, illetve megadhatók szöveg formájában a tesztlépések nevei, rövid leírásai az annotáció után ugyanúgy. A tesztmetódusok háromféle eredményt szolgáltathatnak:

- Sikeres végrehajtás: A teszthez tartozó összes ellenőrzési feltétel teljesül.
- Sikertelen végrehajtás: A teszt lefutott, de valamelyik ellenőrző feltétel nem teljesült.
- Hiba: Komolyabb hiba történt a teszt futtatása során, például, kivétellel megszakításra került a program futása.

A JUnit Assert osztálya segítséget nyújt, a teszt metódusokon belül az aktuális és elvárt eredmények összehasonlítására. Az Assert osztály a következő fontosabb metódusokat tartalmazza:

- assertTrue/False(boolean) metódus sikeres eredményt ad, ha a paraméterként megadott logikai kifejezés igaz vagy hamis.
- assertSame/NotSame(Object, Object) metódus sikeres eredményt ad, ha a paraméterként megadott két referencia ugyanarra az objektumra mutat.
- assertEquals(int, int) metódus sikeres eredményt ad, ha a két paraméter egyenlő. Ebből a metódusból minden primitív elemhez létezik megfelelő változat.
- assertEquals(Object, Object) metódus sikeres eredménnyel tér vissza, ha a paraméterként megadott két objektum egyenlő, az equals metódus használatával.

Az összes összehasonlító assert-tel kezdődő metódusból létezik olyan változat, amely első paramétereként hibaüzenetet lehet meghatározni szöveg formájában. Ezt a szöveget a tesztkörnyezet az ellenőrzés sikertelen kimenete esetén írja ki.

A tesztmetódusok implementációjának újrahasznosíthatósága érdekében a JUnit 4-es verziója tartalmaz egy tesztesetet megvalósító osztályhoz paraméter lista definiálását, ezt nevezik paraméterezett tesztnek. Ezzel a funkcióval az egyszer megírt tesztmetódusok futását különböző paraméterkombinációk mellett hajthatjuk végre egyszerű és átlátható formában. A paraméterezett teszt létrehozása több formában is megvalósítható, ezek közül a véleményem szerint legáttekinthetőbb megoldást mutatom be.

Elsőként az osztály futtatókörnyezetét meghatározó @RunWith annotációban kell megjelölni, hogy a Parametrized.class futtató osztállyal szeretnénk a teszt esetet definiáló osztályt végrehajtani. Következő lépésben létre kell hozni az osztály konstruktorát, amiben a teszt paramétereket és a kimenettől elvárt értékeket tároljuk el az osztályváltozókban. Végezetül a tesztparaméterek értékeit tartalmazó statikus metódus megírása szükséges, amely a paraméter kombinációkból előálló Cellection objektummal tér vissza. A paraméterlista előállításáért felelős metódust a @Parameters annotáció jelöli ki. A JUnit lehetőséget kínál az egyes paraméterkombinációk egyedi

megjelölésére is az annotáció után elhelyezett zárójelben. A teszteset szöveges azonosítójában a paraméterek aktuális értékei megjeleníthetők a paraméterindexek kapcsos zárójelben való hivatkozásával. A paraméterezett tesztosztály implementációjához szükséges kiegészítéseket a 11. ábra demonstrálja.

```
@RunWith(Parameterized.class)
public class NVIAnalyserTest_validateBlockTest2 {

    @Parameters(name = "{0}: BlockSzie:{1}, ArrayLength: {2}, Max block size:{3}, Expected result:{4}")
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
            /* TSname | blockSize | dataArray | expected result */

            {"Invalid data array", 0, -1, 0x3, false },
            {"Valid parameters",   0,  0, 0x3, true },
            {"Valid parameters",   1,  0, 0x3, false },
            {"Valid parameters",   0,  1, 0x3, false },
            {"Valid parameters",   2,  2, 0x3, true },
            {"Valid parameters",   3,  3, 0x3, true },
            {"Valid parameters",   4,  4, 0x3, false }
           });
    }

    /* Input parameters */
    private int actBlockSize;
    private byte[] actDataArr;
    private int actMaxBlockLength;
    /* Expected return value of the validateBlock method. */
    private boolean expResult;
    /* Actual result */
    private boolean actResult;

    public NVIAnalyserTest_validateBlockTest2( String testName,
            int paramBlockSize, int arrLength, int paramMaxBlockLength, boolean paramExpResult) {

        this.actBlockSize = paramBlockSize;
        this.actMaxBlockLength = paramMaxBlockLength;
        this.expResult = paramExpResult;
        if(arrLength < 0) {
            this.actDataArr = null;
        } else {
            this.actDataArr = new byte[arrLength];
        }
        Block testBlock = new Block("TestBlock", 0x01, 0x00000000, actBlockSize, actDataArr);
        this.actResult = testBlock.validateBlock(actMaxBlockLength);
    }

    @Test
    public void asserReturValues() {
        assertTrue(this.actResult == this.expResult);
    }
```

11. ábra – Paraméterezett JUnit teszt példa

Már egy összetett funkcionalitást megvalósító részegység esetében is, könnyen belátható, hogy az erőforrások korlátjai miatt az összes bemeneti paraméterkombináció nem tesztelhető. Ezért gyakran alkalmazott technika a paraméterértékek meghatározásánál a határértékanalízis (boundary value analyzis). Ez azt jelenti, hogy a funkcionális specifikációból kiindulva meghatározhatunk olyan tesztparaméter értékeket, melyek segítségével egy adott funkcionalitás a legkevesebb erőforrással letesztelhető. Ehhez szükséges az ekvivalencia partíciók meghatározása, a specifikáció alapján. Az ekvivalencia partíció egy olyan részhalmaza a paraméter értékeinek amelyekhez azonos kimeneti érték tartozik.

Több paraméter esetén a paraméterértékek összes kombinációjának részhalmazai alkotják az ekvivalencia osztályokat. Egy paraméter teljes értékkészletét a típusa határolja be. Például, egy 8 bites előjel nélküli paraméter értékkészlete 0-255 tartományba esik. Mivel a specifikáció alapján az egyes tartományokba eső értékekhez azonos kimenetelek tartoznak, ezért leszűkíthetők a tesztelendő értékek. Egy ilyen ekvivalenciafelosztás a 12. ábra látható.

12. ábra – Példa ekvivalenciaosztályok

A 12. ábra azt mutatja, hogy az egyes tartományokba eső értékekhez azonos belső utat járunk be végrehajtás szempontjából. Azt feltételezzük, hogy ha ezekből a csoportokból egy tesztértékre helytelenül vagy helyesen viselkedik a rendszer, akkor az összes többire ugyanúgy fog.

A hibák legtöbb esetben a partíciók határánál rejtőzhetnek. Leggyakrabban rossz döntési reláció, helytelen ciklus be- és kilépési feltételek vagy hibás adatstruktúra lehet a hiba forrása. Ezért érdemes belevenni a tesztadatokba a hátértékeket és mindkét oldalán elhelyezkedő értékeket is, ezt nevezzük határérték-ellenőrzésnek.

5.5.2 „Mockito” verifikáció

A unit tesztelés az egyes részegységek működését a többitől elkülönítve ellenőrzi. Ezért a külső hatásokat, amiket más funkciókért felelős osztályok vagy rendszerek okoznak a tesztelendő egységben eliminálnia kell a unit tesztnek, ha lehetséges. Ez megvalósítható a teszthez alkalmazott helyettesítésekkel az eredeti függőségek helyett.

A teszthelyettesítések a következők lehetnek:

- Egy bábobjektum átadásánál, amikor az objektum metódusaira nincs szükség, egy helyettesítő objektum használata alkalmas például a metódus paraméterlisták kitöltéséhez.
- Hamis objektumok, amelyek rendelkeznek implementációval, de jellemzően az eredetihez képest sokkal egyszerűbben vannak

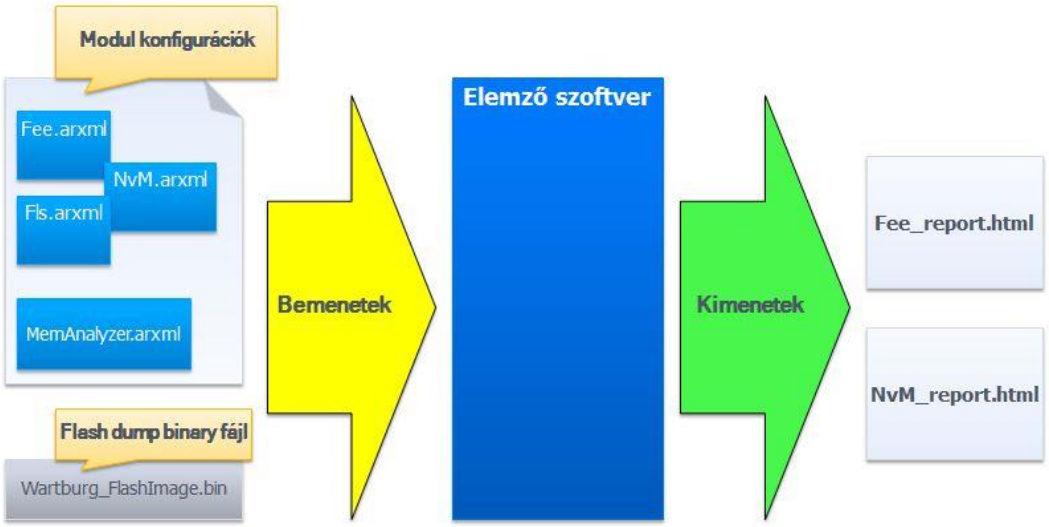
megvalósítva. Például egy objektum, amely eredetileg adatbázis adatokat használ, könnyen helyettesíthető statikusan megadott memóriában tárolt adatokat implementáló hamis objektummal.

- A csonk (stub) osztály, ami egy interfész vagy egy osztály részleges implementációját tartalmazza, hogy az eredeti interfész vagy osztály helyett használja a tesztelendő szoftver az eredetiek helyett. A csonkok általában semmilyen válasszal vagy befolyással nem lépnek interakcióba a tesztelt modullal. Ehelyett, a metódushívásokkal kapcsolatos paramétereket és ezeknek a sorrendjét rögzítik leggyakrabban.
- Az utánzó (mock) objektum egy bábimplementáció egy interfész vagy egy osztály megvalósítására, amely különböző visszatérési értékékkel tér vissza bemeneti paraméterek alapján. Az utánzó objektumok úgy vannak megírva, hogy a teszt folyamán különböző belső működést képesek legyenek megvalósítani az aktuális beállításaik alapján. Tipikusan, rögzítik a tesztelt modullal és teszttel való kölcsönhatás paramétereit, eredményeit és ezek az ellenőrzési fázisban a teszt rendelkezésére bocsátják.

Teszthelyettesítők átadhatók más tesztelendő objektumok számára. A teszt ellenőrizheti, hogy az osztály az elvárt viselkedéssel megegyezően lép kölcsönhatásba az átadott objektumokkal a teszt futása alatt. Például ellenőrizhető, hogy melyik metódusok lettek meghívva az utánzó objektumon. Ez segít megbizonyosodni arról, hogy csak a teszt befolyásolja az objektum állapotát és nincs semmilyen külső hatást gyakorló tényező. Az utánzó objektumok létrehozása történhet kézzel, vagyis keretrendszer szintaktikáját követő kód megírásával vagy az objektum szimulálásával.

6 ECU integrált Flash memóriaanalízis

Az elemző program lényegében egy Eclipse plugin, amely Java nyelven megírt és a cég által fejlesztett Autosar Architect névre keresztelt Eclipse alapú fejlesztőkörnyezet kódgenerátor osztályának segítségével a bemenetéül szolgáló bináris kiterjesztésű Flash memória részletből és a hozzá tartozó konfigurációkból (NvM, Fee, Fls) készít HTML riportokat. A plugin alkalmazás felépítése a 13 ábrán látható.



13. ábra - Elemző szoftver

Az elemző szoftver bemenetéül szolgálnak a korábban bemutatott mikrovezérlőbe integrált memória használatát támogató modulok konfigurációját leíró AUTOSAR .arxml modell fájlok. Ezekben a fájlokban szerepelnek a memória tartalmának értelmezéséhez szükséges paraméterek. Ezenkívül az elemzéshez szükséges bemeneti paramétereket a „MemAnalyzer.arxml” fájl tartalmazza. Ezek között a paraméterek között foglal helyet például a feldolgozandó bináris fájl elérési útvonala, ami a vezérlőegység memóriájából lett kimentve valamint, a memóriarészlet fizikai kezdőcíme, a mikrovezérlő típusa. Továbbá, lehetőség van meghatározni, a klasszifikált blokkok sorrendjét, vagy állapot szerinti csoportosítását is.

Az elemzés kimenetét képző HTML riportok ember számára jól értelmezhető, strukturált, informatív formában jelenítik meg az Fee és az NvM modulok szintjén klasszifikált lapokat, blokkokat. A modulok hierarchikus felépítéséből fakadóan az

NvM és az FEE összetartozó blokkjaihoz, az NvM riport referenciákat tartalmaz az FEE riportban lévő megfelelő blokkokra.

6.1 Elemző szoftver felépítése

6.1.1 Bemenetek kezelését végző osztályok

Az FLS modul [10] konfigurációs paraméterei közül a következő paraméterek eltárolására van szükség a memóriarészlet értelmezéséhez:

- Szektor lapméret (FlsPageSize): Legkisebb önállóan írható egység a flash memóriában (nem összetévesztendő az FEE-ben használt lappal),
- Szektorméret (FlsSectorSize): FLS szektor, mint legkisebb önállóan törölhető egység mérete,
- Szektor kezdőcíme (FlsSectorStartaddress): A szektor kezdőcíme az egybefüggő virtuális címtérben.

Az FEE [9] feladatainak ellátásához szükséges konfigurációs paraméterek a cég, (thyssnekrupp) által készített speciális implementáció alapján lettek meghatározva, amelyeket a cég érzékeny adatként kezel, az AUTOSAR FEE specifikációjából [11] a következő paraméterek szükségesek az elemzéshez:

- Virtuális lapméret (FeeVirtualPageSize): A blokkokat felépítő legkisebb egység bájtokban megadva. A logikai blokkok méretének kötelezően a virtuális lapméret többszörösének kell lennie.
- Blokkszám (FeeBlockNumber): Blokkazonosító,
- Blokkméret (FeeBlockSize): Logikai blokkméret

Az NvM modul [8] konfigurációjából a blokkokhoz tartozó konfigurációs paraméterek amelyek szükségesek a memóriarészletben szereplő blokkok értelmezéséhez a következőek:

- Blokk CRC típus (NvMBlockCrcType): A blokkhoz tartozó CRC kód méretét szabja meg (nem használt, 8, 16, 32 bites lehet).
- Blokkmenedzsmenttípus (NvMBlockManagementType) – Az NvM blokk típusát határozza meg, ami lehet natív, redundáns, dataset.

- Blokkbázisszám (NvMNvBlockBaseNumber): Az FEE és az NvM blokkok közti kapcsolat meghatározásához. A korábban ismertetett képlet segítségével meghatározható az FEE blokk száma.
- Dataset kiválasztó bitek (NvMDatasetSelectionBits): A „dataset” típusú blokkok maximális számát meghatározó szelektor bitek.
- Blokkhossz (NvMNvBlockLength): NvM blokk mérete bájtokban megadva.
- Blokkok száma (NvMNvBlockNum): Natív blokk esetén az értéke egy, redundáns blokk esetén kettő az értéke. Dataset típusú blokk esetén értéke meghatározza, hogy hány darab blokkból áll az adategyüttes.
- Blokkazonosító (NvMNvramBlockIdentifier): A blokk egyedi azonosítója, amely a blokk fejlécében szerepel opcionális esetben.
- Statikus blokk fejlécellenőrzés (NvMStaticBlockIDCheck): A blokk fejlécének ellenőrzését jelző konfigurációs paraméter, amelynek igaz értéke mellett szükséges a blokk fejlécének eltárolása a blokkadattal együtt és ellenőrzése a validációs folyamat részeként.

A konfigurációs paraméterek eltárolásán túl szükség van a bináris állomány megnyitására, és eltárolására könnyen kezelhető adattömb formájában. Ennek a feladatnak az elvégezéséhez készítettem egy osztályt, amely a konstruktora paraméterének a bináris fájl elérési útvonalát kapja. A megfelelő hibakezelési formákkal együtt a konstruktor megnyitja és bájttömb formájában eltárolja az elérési út által meghatározott fájl tartalmát.

6.1.2 Elemző adatstruktúrák és osztályok

Elsőként, a legalapvetőbb feladat az FLS és az FEE konfigurációs paraméterek alapján a memóriarészletben behatárolni, az FEE által eltárolt blokkokat és a memória-menedzsmentinformációkat kinyerni. Ehhez a feladathoz az FEE-ben konfigurált lapokhoz tartozó kezdőcímek meghatározása szükséges, a megfelelő FLS szektorok címeinek meghatározásával. Miután sikerült az egyes lapok fizikai elhelyezkedését meghatároznunk elkezdjük a memóriarészletben található blokkok speciális blokk-menedzsmentinformációinak és adatainak eltárolását egy, az erre a feladatra létrehozott

osztályba. A memóriarészletben talált blokk paraméterekből és adatokból létrehozott objektumokat egy listában szervezve tároljuk el a könnyű kezelhetőség érdekében.

A már előklasszifikált FEE blokkokból (blokkleíró és adat) álló listán a leíró hitelesítéséhez az implementációban használt adat konzisztenciájának védelmére szolgáló ellenőrzéseket kell elvégezni, hogy egy blokkot korrektnek vagy sérültnek tekinthessünk és megjelöljük ezek szerint. Ezután a korrekt és a sérült blokkok leíróján egyaránt elvégzett további ellenőrzések segítségével a blokkok állapotának meghatározása is szükséges az átfogó kép elérése érdekében. A blokkok állapota általánosan lehet érvényes, érvénytelenített és megsemmisített.

Összegezve, az FEE szerinti klasszifikációhoz készítettem a blokkleíró és a hozzá tartozó adat egységbe fogásához egy osztályt, amely konstruktora a blokkhoz tartozó menedzsmentinformációkat és adattömböt tartalmazza. Készítettem a memórialapokat leíró osztályokat is az adatok egységbezárása érdekében. Egy ilyen lapleíró osztályból létrehozott objektum a hozzá tartozó memóriarészből beolvasott blokkleíró objektumok listáját, a laphoz tartozó információkkal együtt tartalmazza.

Az NvM blokkok klasszifikációjának első alapvető lépése, hogy az egyes NvM blokkokhoz, a memóriarészletben talált FEE blokkokat társítsunk. Az NvM blokkok megfelelő kezelésének érdekében a korábban már bemutatott NvM blokktípusok paramétereinek tárolásra szolgáló osztályokat hoztam létre mind a natív, a redundáns és a dataset típusú blokkok részére.

Ezek az osztályok a megfelelő konfigurációs paramétereiken kívül tartalmaznak egy listát, amelyben az NvM blokkhoz tartozó FEE blokkok szerepelnek. A listát az NvM blokkokhoz tartozó osztályok konstruktorában adjuk át a többi paraméterével együtt. A teljes körű riport generálása érdekében az NvM blokkokat leíró objektumokat a típusuk alapján listába szervezve tároljuk, valamint a három blokktípuson kívül még a memóriarészletben nem tárolt NvM blokkok listáját is.

Az egyes NvM blokkok adat konzisztenciájának ellenőrzéséhez az opcionális fejléc és CRC ellenőrzéseket egységesen mindegyik típuson elvégezzük. Ezen kívül a redundáns blokkok esetében elvárt tovább a két letárolt adat blokk egyezésének ellenőrzése is.

7 Elemzés vizualizációja

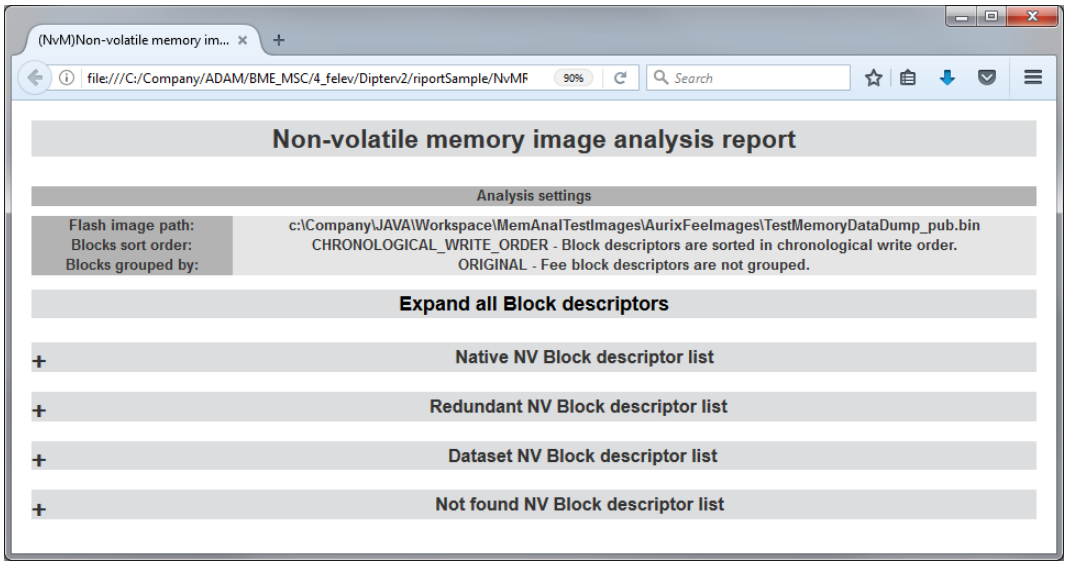
A vezérlőegység belső memóriatartalmának részletes elemzése semmit nem ér, az átláthatóságot nagyban segítő, továbbá interaktív vizualizáció nélkül. Fontos feladat, a legjobb technológia használata erre a problémára. Az elemzés eredményeit könnyedén lehetne táblázatos formában xls vagy csv formátumú fájlban tálalni a felhasználó számára. Továbbá az AUTOSAR modulok konfigurációjának tárolására szolgáló arxml vagy megannyi más xml formátum használata is megfelelő lehet a cél megvalósítására.

A feladatomhoz a riport generálásához HTML formátumot választottam, melynek eszközkészlete lehetővé teszi a feldolgozott memóriarészlet beszédes vizualizációját. Emellett, könnyen kezelhető és nincs különleges szoftverigénye, mivel könnyen megnyitható, még egy átlagos személyi számítógépen található böngésző segítségével is.

7.1 Interaktív HTML riport

Az elemzés legfontosabb feladata, hogy az emberi képességek számára nem triviális memóriarészletet, könnyen értelmezhető és kezelhető formában jelenítse meg a felhasználók számára. Az interaktivitást a megjelenő riport kirészletezhetősége adja, mely az erre vonatkozó vezérlő elemekkel (+, -) állítható.

A 5.3 fejezetben bemutatott CSS fájlba szervezett stíluselemek használata sok munkát takarított meg a riport fájl fejlesztése során, mivel egy megjelenítési szabállyal tudtam több HTML elem megjelenítési tulajdonságait definiálni. Emellett az egyes stílusok hangolásához elegendő egy helyen megváltoztatni a releváns paraméter értékét. A következő ábrán az elemzés eredményét ismertető riport interaktív HTML dokumentum kezdő felülete látható.



14. ábra – NvM riport kezdőfelület.

A riport első blokkjában az analízis beállításait láthatjuk, elsőként a feldolgozott memóriarészlet elérési útvonalát, ezután a blokkok rendezésével kapcsolatos beállítást és hozzájuk tartozó rövid leírást.

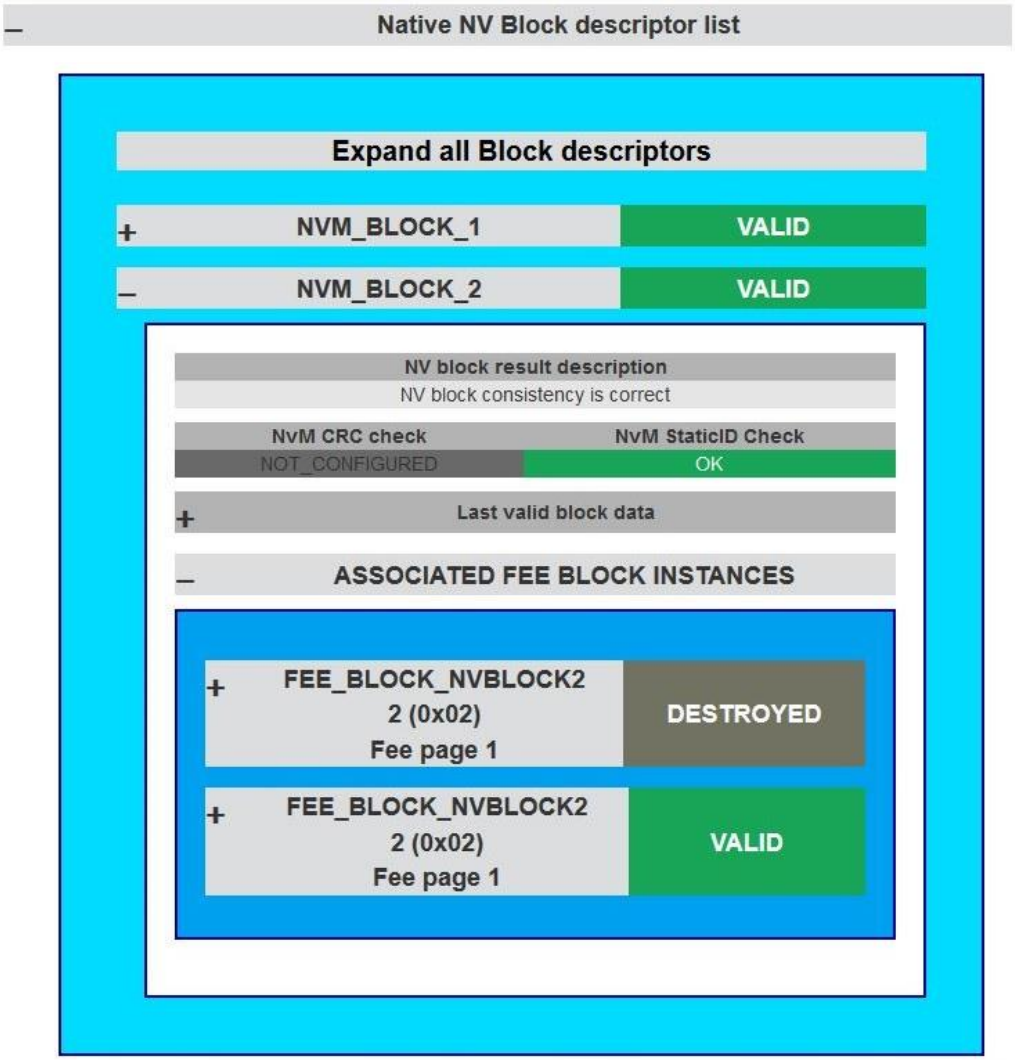
Ezek közül az első beállítás azt határozza meg, hogy a blokkokat milyen sorrendben jelenítse meg a riport egymás alatt. Az ábrán látható kronológiai írási sorrend (CHRONOLOGICAL_WRITE_ORDER) azt jelenti, hogy időben egymás után a memóriába írás sorrendjében jelenítendőek meg a blokkok.

Lehetőség van a blokkok fordított sorrendben, vagyis a legutolsó memóriába írt blokktól kezdődő kilistázására (LAST_WRITTEN_INSTANCE_FIRST), az elemző szoftver sorrendszervező konfigurációs paraméterének állításával. Ezzel a két lehetőséggel, megjeleníthetőek a memóriában szereplő blokkok írási sorrendjükben, ha a blokkok sorrendje a lényeges, illetve sorrendben fordítva, ha a legutoljára kiírt blokkok (legfrissebb eltárolt adatok) hordozzák a keresett információt.

További beállítási lehetőségünk van a blokkok csoportosítására. Ennél is két paraméterlehetőség közül lehet választani: eredeti (ORIGINAL) és érvényesség (VALIDITY). Alapértelmezetten a blokkok, abban a sorrendben jelennek meg egymás után, ahogy azok a memóriarészletben szerepeltek. Emellett, ha szükség van a blokkokat érvényességük, vagyis pontosabban klasszifikációjuk alapján csoportosítani, akkor a VALIDITY beállítást használjuk. Ezzel minden egyes lapon belül található blokkok közül elsőként az érvényesnek, ezután az érvénytelenítettnek, és végül a

sérültnek, inkonzisztensnek klasszifikált blokkok jelenítődnek meg, a felsorolt csoportokba rendezve.

Az elemzésbeállítások felsorolása után, egy vezérlőblokk található, amely segítségével az összes Nv blokk lista és a hozzájuk tartozó FEE blokkok megjelenítődnek hierarchikus formában, ahogy a 15. ábrán is látható, amely egy fiktív riport natív blokk listáját szemlélteti.

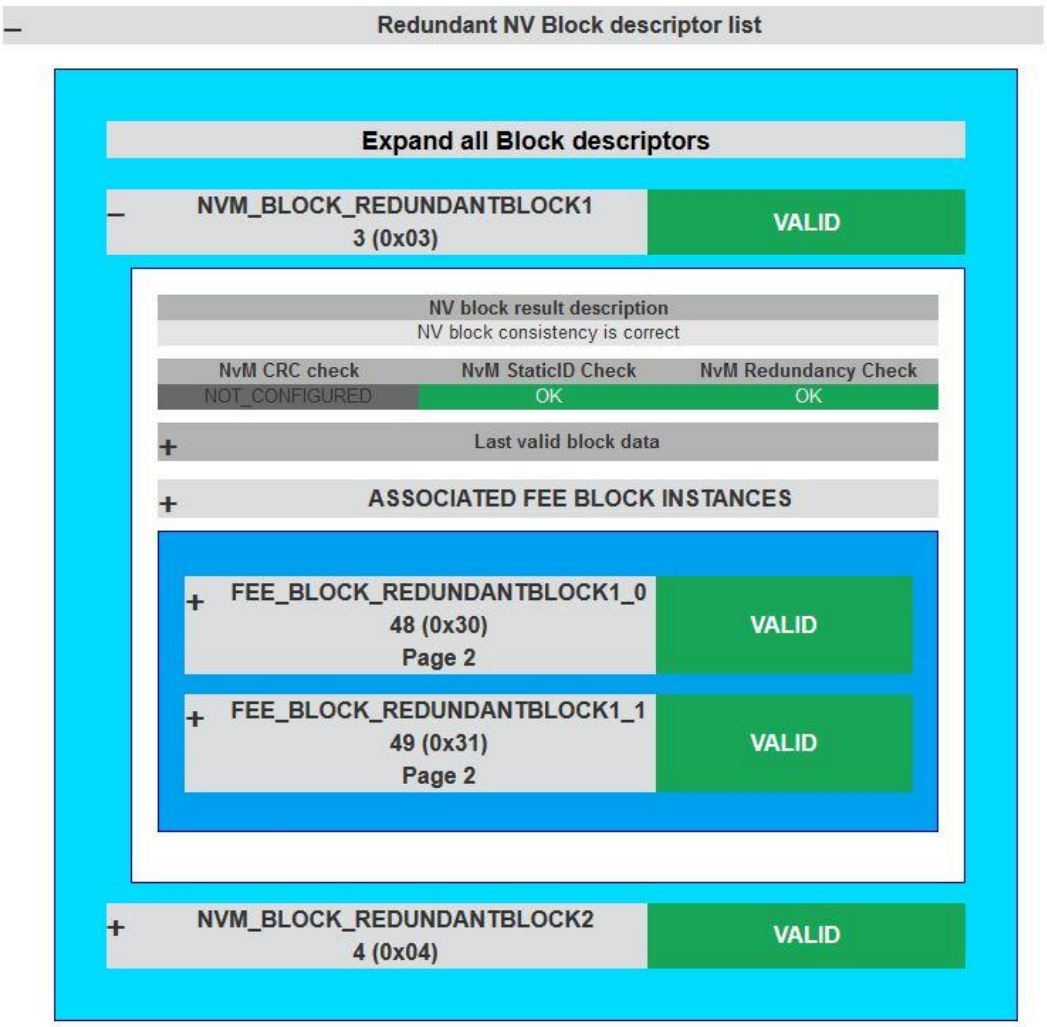


15. ábra – Natív NvM blokkok ábrázolása

A 15. ábra natív blokkokról közölt információk szemlélteti. Egy natív egyik legfontosabb eredménye a klasszifikálással meghatározott állapot, amely a képen látható két blokk esetén érvényes (VALID) értéket vesz fel. Ezen túl, a korábban említett érvénytelenített (INVALID) és sérült (CONSISTENCY FAIL) állapotok fordulhatnak elő. Előbbi azt jelenti, hogy a blokkból nem található egyetlen érvényes blokk sem az analízis beállításaival kijelölt memóriaterületen belül. A sérült állapot arra utal, hogy az

NvM blokk konzisztenciaellenőrző mechanizmusai az elvárttól különböző eredményt szolgáltattak, vagy a fizikai memóriában szereplő FEE blokkmenedzsment információinak integritásvédelme jelez hibás adatokat.

Az egyes NvM blokkokra kattintva megjelennek a blokk klasszifikációjának rövid leírásai, és a blokkhoz tartozó blokk és adat hitelesítésre szolgáló mechanizmusok: a CRC ellenőrzés, a statikus azonosító és kizárólag a redundáns blokkok esetén a redundanciaellenőrzés eredményei. A redundáns blokk leíróját, a 16. ábra szemlélteti. Ezen felül a legutolsó, érvényes adatokat tartalmazó FEE blokk adatbájtjait lehet elérni ez ellenőrzéseket tartalmazó táblázat alatt található lenyíló ablakban. Ezalatt a korábban ismertetett sorba és csoportokba rendezési beállításoknak megfelelően, a releváns FEE blokkok szerepelnek.



16. ábra - Redundáns NvM blokkok ábrázolása.

A Dataset és a „nem talált” típusú NvM blokkokkal kapcsolatos megjelenített információk teljesen megegyeznek a natív blokk leíróval.

A „nem talált” listába azok az NvM blokkok kerülnek, amelyek szerepelnek az NvM modul konfigurációjában, viszont a feldolgozott memóriaelemben nem szerepelt.

A dokumentum stílusának meghatározásához a HTML elemekhez osztályokat definiáltam, ezután az osztályokhoz különböző CSS stílusszabályokat definiáltam különböző stílusjegyek elérése érdekében. Külön osztályba tartoznak az NvM blokktípusok neveit tartalmazó elemek, a hozzájuk tartozó elemeket felsoroló elemek, az NvM blokkok neveit tartalmazó elemek, az NvM blokkokhoz tartozó klasszifikációs információkat és FEE blokkokat felsoroló elemek.

Ahhoz, hogy az egyes NvM blokkok listája interaktív módon megjeleníthető és eltüntethető legyen a lista nevét tartalmazó HTML elemet kattinthatóvá tettem. Ezekre az elemekre való kattintás hatására egy JavaScript függvény hívódik meg. A blokk listában szereplő NvM blokkelemek és az összes ilyen funkciót igénylő elemek estében is ugyanígy jártam el.

Minden egyes kattintható elem rendelkezik egyedi azonosítóval annak érdekében, hogy meg lehessen határozni a hozzá tartozó megjelenítendő vagy elrejtendő HTML elemet. Egy elem megjelenítéséért az osztály listájában szereplő osztályok és a hozzájuk tartozó stílusszabályok felelősek. Az aktuálisan megjelenített elemek listájában a „visible”, a rejtettekében pedig az „invisible” osztálynak kell szerepelnie. Előbbi esetén a stílusszabály display utasítása block értékkel, az utóbbi esetén none értékkel van definiálva. A JavaScript függvényben annyi a feladat, hogy ki kell cserélni az osztálylistában a két előzőleg ismertetett osztályt. Először meghatározza az elem azonosítóját, amire a kattintás történt. Ezután az azonosítóból meghatározza a hozzá tartozó megjeleníteni vagy elrejteni kívánt blokk azonosítóját. Az azonosító segítségével már lehetséges az elem attribútumainak változtatása, vagyis az osztálylista elemeinek cseréje. Az imént ismertetett funkcionalitás egy konkrét megvalósítását a 17. ábra demonstrálja.

```
        <h1 class="BlockButtonClass" id="Block000" onclick="changeVisibility()">
            Block 1
        </h1>

        <div class="DataBlockClass visible" id="Block000_data" >
            <p>Some block specific content.</p>
        </div>

        <script>
            function changeVisibility() {
                var buttId = this.id;
                var panelId = buttonId + "_data";

                var element = document.getElementById(panelId);

                if(element.classList.contains("visible")) {
                    element.classList.add("invisible");
                    element.classList.remove("visible");
                }else{
                    element.classList.add("visible");
                    element.classList.remove("invisible");
                }
            }
        </script>
    </body>
```

17. ábra – Példa HTML elemek megjelenítésére és elrejtésére.

7.2 Dinamikus riportgenerálás Xtend Template Expressions osztályokkal

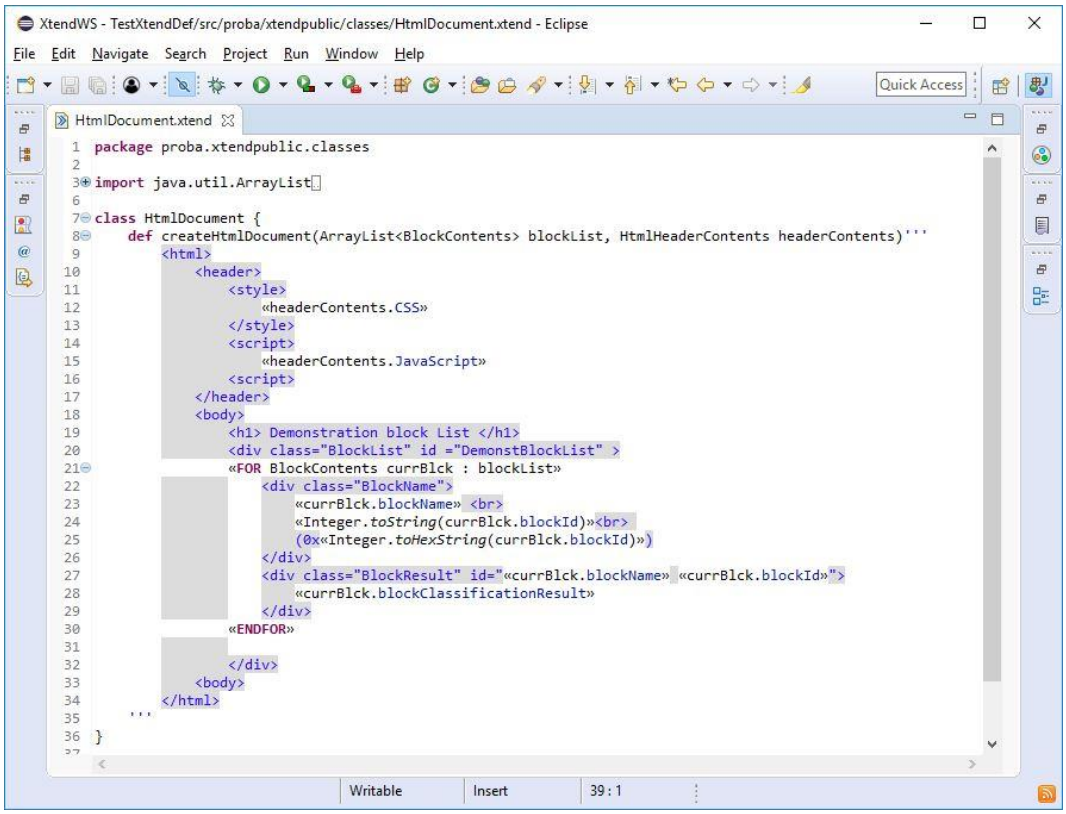
Az eredmények megjelenítésére szolgáló HTML elemeket alkotó szöveg létrehozása egy korántsem triviális feladat. Ennek oka, hogy a dokumentum legnagyobb részét statikus szöveg alkotja.

A riportfájlt alkotó szövegek két részre bonthatók: teljesen statikus és a változó paramétereket tartalmazó statikus elemek. A teljesen statikus elemek közé tartozik a CSS, JavaScript fájlok, szabványosított HTML elemek tartalma. A részben statikus elnevezéssel, azokra a blokk leírókra gondolok, amelyek elemei kizárólag egy-egy kulcsszóba, paraméterben, vagy stílusszabályban különböznek.

Ekkora mennyiségű szöveg kezelése, a JAVA „String” osztályának használatával megoldható, viszont erőforráspazarló, nehezen átlátható és nagy mennyiségű ismétlést tartalmazó forráskódot eredményez. Véleményem szerint a legjobb Eclipse általtal támogatott megoldás az Xtend sablon kifejezések funkciója (Template Expression).

A 18. ábrán egy Xtend Expression template példa kód látható, amely HTML szabványnak megfelelő szöveg előállítását valósítja meg.

A HTML dokumentum törzsébe a createHtmlDocument metódus blockList paraméter listaelemeinek értékeit helyezi el a blokkok megjelenítésére szolgáló HTML elemekben adaptív módon.



18. ábra - Xtend HTML template expression példa.

A korábban ismertetett elemzés, riport eredményeket vizualizáló HTML fájljainak tartalmát is hasonló megoldásokkal hoztam létre, némileg összetettebb módon. Ezzel a feladattal foglalkozó osztályhierarchia felépítését a következő fejezet taglalja.

7.2.1 HTML riportgeneráló Xtend osztályok

A riportszöveg generálását, három szinten valósítottam meg. A legfelső szinten a teljes HTML tartalom áll össze, a szabvány által meghatározott követelmények alapján a fejléc (header), törzs (body) paraméterekkel, amelyeket egy alacsonyabb szinten állítunk elő.

A fejléc tartalma teljesen statikus CSS stílus szabályokat és JavaScript kódot tartalmaz, viszont ezeknek a tartalmát a *NvmHtml5Css* és *NvmHtml5JavaScript* osztályokban implementáltam, a könnyed kezelhetőség elérésének érdekében.

A törzsben szereplő konfigurációs elemeket tartalmazó szöveget, a törzs szöveget létrehozó *Html5Body* osztályban implementáltam, ebbe az osztály által létrehozott szövegbe ágyazódnak be a blokktípusok szerint szétválogatott listák string formában.

Az elemzés eredményének információit hordozó listák szövegét, a *NvmHtmlBlocks* osztály hozza létre. A listákat alkotó blokk HTML elemeket a *NvmHtml5NativeBlock*, *NvmHtml5RedundantBlock*, *NvmHtml5DatasetBlock* és *NvmHtml5NotFoundBlock* osztályokban implementáltam az Xtend template expression funkciójának segítségével.

Az alábbi, ábrán a natív blokkok adatait reprezentáló HTML elemek előállításáért felelős Xtend osztály láthat, ami táblázat elemekből épül fel.

```
class NvmHtml5NativeBlock {
    def static nvmHtml5NativeBlock(NvMNativeBlockDescriptor nvMDesc, StringBuilder AssocFeeBlocks)'''

    <table CLASS="GroupHeadingTable">
        <tr class="buttSign collapsed" data-toggle="collapse" data-target="#«nvMDesc.nvmBlockName»" aria-expanded="true">
            <td width="60%">
                <big class="Heading3">«nvMDesc.nvmBlockName»<br> «nvMDesc.nvmBlockId» (0x«String.format("%02X", nvMDesc.nvmBlockId)»)</big>
            </td>
            <td class="«nvMDesc.lastValidNVBlock.nvmBlockReslult.htmlCellType»" width="40%">
                <big class="Heading3">«nvMDesc.lastValidNVBlock.nvmBlockReslult.name»</big>
            </td>
        </tr>
    </table>
    <div class="Blk collapse" id="«nvMDesc.nvmBlockName»" aria-multiselectable="true">
        <table class="ResultTable">
            <tr>
                <th class="TableHeadingCell">NV block result description</th>
            </tr>
            <tr>
                <td class="DefaultCell">«nvMDesc.lastValidNVBlock.nvmBlockReslult.desc»</td>
            </tr>
        </table>
        <table class="ResultTable">
            <tr>
                <th class="TableHeadingCell">NvM CRC check</th>
                <th class="TableHeadingCell">NvM StaticID Check</th>
            </tr>
            <tr>
                <td class="«nvMDesc.lastValidNVBlock.crcCheckRes.htmlCellType»">«nvMDesc.lastValidNVBlock.crcCheckRes.name»</td>
                <td class="«nvMDesc.lastValidNVBlock.staticIdCheckRes.htmlCellType»">«nvMDesc.lastValidNVBlock.staticIdCheckRes.name»</td>
            </tr>
        </table>
    '''
```

19. ábra – Natív Nv blokk információkat HTML elem generáló Xtend osztály

Elsőként a blokkazonosítókat tartalmazó csoport helyezkedik el, itt szerepel a blokk neve, azonosítója és a blokk-klasszifikáció eredménye, aminek a stílusát az eredmény értéke szabja meg. Inkonzisztens vagyis hibás blokkok piros, helyes érvényes blokkok zöld és olyan blokkok amikhez nem tartozik érvényes FEE blokk sötétszürke háttérszínnel jelenítik meg az eredményt, a megfelelő felirattal együtt. A korábban megismertetett megoldást alkalmazva, erre az elemre kattintva nyitható és zárható következő nagyobb elem (lásd 20. ábra), ami a blokk részletes információit tartalmazza.

```
    <table class="ResultTable">
      <tr class="TableHeadingCell buttSign collapsed" data-toggle="collapse" data-target="#«nvMDesc.nvmBlockName»_LastValidData" aria-expanded="true">
        <th class="TableHeadingCell" width="100%"> Last valid block data </th>
      </tr>
    </table>
    <div class="Blk_data collapse" id="«nvMDesc.nvmBlockName»_LastValidData">
      <table class="ResultTable">
        <tr>
          «IF nvMDesc.lastValidNVBlock.blockDescClassState == BlockDescClassState.CORRECT_BD_BDCLASS»
          <td class="DataBytesCell">«nvMDesc.getLastValidNVBlock().getDataToString("%02X ")»</td>
          «ELSE»
          <td class="DataBytesCell">«" Data have not been found. "»</td>
          «ENDIF»
        </tr>
      </table>
    </div>

    <table class="GroupHeadingTable" style="border-collapse:collapse;">
      <tr class="buttSign collapsed" data-toggle="collapse" data-target="#«nvMDesc.nvmBlockName»_allFee" aria-expanded="true">
        <td width:40%>
          <big class="Heading3">
            ASSOCIATED FEE BLOCK INSTANCES
          </big>
        </td>
      </tr>
    </table>
    <div class="panel-group Pa" id="«nvMDesc.nvmBlockName»_allFee" role="tablist" aria-multiselectable="true">
      <!-- Fee blocks will be below -->
      «AssocFeeBlocks»
    </div>
  </div>
  '''
}
```

20. ábra – Natív Nv blokkhoz tartozó FEE blokkok riportját előállító Xtend osztály részlet.

Ebben az elemben az első megjelenítendő információ a blokk-klasszifikálás eredményének rövid leírása. Ezt, a klasszifikációhoz szükséges CRC és StaticId ellenőrzés eredményei követik. Végül, az FEE blokkok felsorolása előtt ha létezik ilyen, a legfrissebb memóriában talált érvényes blokk adatai jelennek meg bájtonként csoportosítva, hexadecimális formában.

A natív és redundáns blokkok HTML reprezentációit előállító Xtend osztályok közötti egyetlen, különböző elemét a 21. ábra szemlélteti. Elsősorban, a blokk ellenőrzés eredményei között jelenik meg a redundanciaellenőrzés. Normális esetben ezek eredményeit kétszeresen ábrázoljuk. Elsőként az elsődleges (primary), alatta pedig a másodlagos (secondary) redundáns blokk eredményeit. Ezt az Xtend Expression template ciklussal valósítottam meg. A ciklus annyiszor fut le, amennyi FEE blokk tartozik a legutolsó érvényes NvM blokk elemhez.

```
  <table class="ResultTable">
    <caption class="TableCaption">Last valid Primary/Secondary NV block check results</caption>
  «FOR idx: 0..(nvMDesc.lastValidRedundantBlock.feeBlockDescDataArr.length - 1)»
    <tr>
      <th class="TableHeadingCell" width="33%">NvM CRC check</th>
      <th class="TableHeadingCell" width="33%">NvM StaticID Check</th>
      <th class="TableHeadingCell" width="33%">NvM Redundancy Check</th>
    </tr>
    <tr>
      <td class="«nvMDesc.lastValidRedundantBlock.crcCheckRes.get(idx).htmlCellType»">«nvMDesc.lastValidRedundantBlock.crcCheckRes.get(idx).name»</td>
      <td class="«nvMDesc.lastValidRedundantBlock.staticIdCheckRes.get(idx).htmlCellType»">«nvMDesc.lastValidRedundantBlock.staticIdCheckRes.get(idx).name»</td>
      <td class="«nvMDesc.lastValidRedundantBlock.getRedundancyCheckRes.htmlCellType»">«nvMDesc.lastValidRedundantBlock.getRedundancyCheckRes.name»</td>
    </tr>
  «ENDFOR»
  <table class="ResultTable">
    <tr class="TableHeadingCell buttSign collapsed" data-toggle="collapse" data-target="#«nvMDesc.nvmBlockName»_LastValidData" aria-expanded="true">
      <th class="TableHeadingCell" width="100%"> Last valid block data </th>
    </tr>
  </table>
```

21. ábra – Redundáns Nv blokk konzisztencia ellenőrzés eredmények

8 Verifikáció és validáció

A tesztelés meghatározott körülmények között végrehajtott szoftverrendszer vagy -komponens futás közbeni eredményeinek vizsgálatával értékeli több szempont szerint. Ezek lehetnek funkcionális aspektusok, mint funkcionális követelmények lefedésének vizsgálata vagy lehetnek nem funkcionálisak mint a szoftver megbízhatósága, terhelhetősége, teljesítménye. [25]

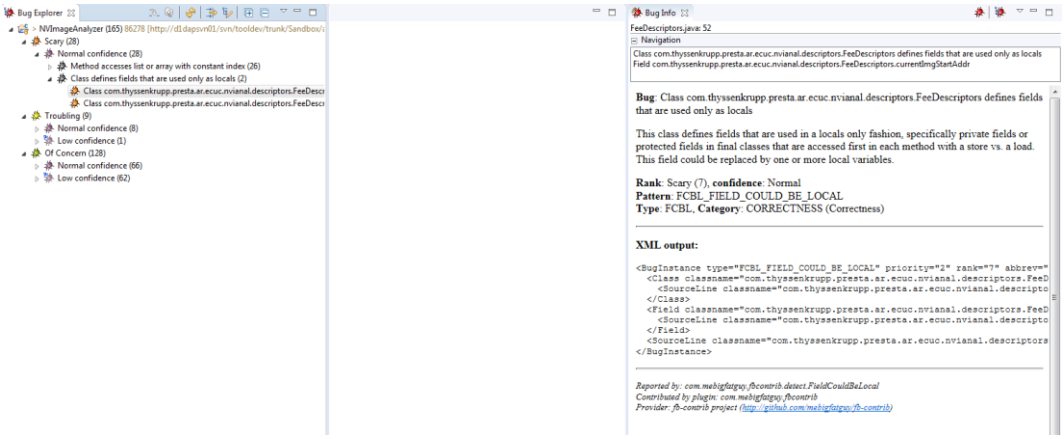
Fontosabb tesztelési alaptételek közé tartozik, hogy a tesztelés a hibák jelenlétét mutatja, kimerítő tesztelés nem lehetséges és a tesztelést minél korábban el kell kezdeni.

A hibátlan teszteredmény nem garantálhatja, hogy nem maradt hiba a rendszerben. Ezzel szemben csökkenti a nem észrevett hibák valószínűségét, de tesztelés folyamán a hibák hiánya, nem bizonyítja, hogy a kód hibamentes. Az, hogy egy teszt az összes lehetséges bemeneti kombinációt alkalmazza a teszt során nem lehetséges. A teljes paraméterérték halmaz helyett, az olyan kombinációkra kell fókuszálni, amely értékek előrevetítik a kihagyott részhalmazok helyességét is.

8.1 Elemző szoftver moduláris (unit) tesztelése

A kód tisztázásához és a rejtett hibalehetőségek feltárásához elengedhetetlen a statikus kód elemzése első lépésben. Az iparban számtalan szoftver keretrendszer létezik a statikus kódanalízis elvégzésére, szinte bármely célnyelvhez. A jelenlegi feladatomhoz a JAVA programnyelv elemzésére szolgáló Eclipse-be ágyazható eszközt, a SpotBugs programot használtam.

Lefuttattam az elemző szoftvert tartalmazó projekten a SpotBugs eszköz FindBugs parancsát amely a 5.5 fejezetben ismertetett csoportokban felsorolta az egyes lehetséges szemantikai hibákat. A hibák javítása során nagy segítséget nyújt, hogy a felsorolás elemei link-ként referálnak a kód releváns részére, ezzel könnyen elérhető a hiba helye. Az eszköz minden hibához készít egy részletesebb elemzést, ami tartalmaz egy leírást az esetleges okokról, előfordulási lehetőségekről, valamint ajánlást ad a hiba kiküszöbölésére.



22. ábra – SpotBugs találatok példa.

A hibaokokat és ajánlásokat értelmezve javítottam a találati listán szereplő hibákat. A legkomolyabb hibák forrása a NULL objektum átadásának lehetősége volt, amelyet a visszatérési érték ellenőrzésével oldottam meg az egyes metódusokban. Kisebb jelentőségű találatokat eredményeztek az inicializálatlan és használatlan tagváltozók, amelyek a fejlesztés közben maradtak az osztályok implementációjában.

Az elemző szoftver részegységeinek tesztelésénél fehér doboz tesztet alkalmaztam, a kód és belső struktúrák ismeretének birtokában.

Ahhoz, hogy megfelelő képet kapjak a bemeneti paraméterek megválasztásának módjától az egyes funkciókat megvalósító metódusokhoz meghatároztam az ekvivalencia osztályokat és a teszteseteket a partíciók határain a JUnit keretrendszer által támogatott paraméterezett tesztekkel hajtottam végre.

Az ekvivalenciapartíciók határértékein kívül, olyan érvénytelen paraméterértékeket is felvettem, amik minimális valószínűséggel fordulhatnak elő, mint például a null objektumok.

A moduláris tesztek végrehajtásának tervezésekor figyelembe vettem az egyes funkcionális egységek más osztályok objektumaitól való függőségeit és ezek komplexitását is. Ahogy a 5.5 fejezetben ismertettem az alapvető modultesztelési technikák egyszerű funkcionalitások tesztelésére alkalmasak leginkább. Emiatt a bonyolultabb feladatokat megvalósító osztályok esetében két megoldást alkalmaztam a függőségi probléma megoldására.

Az elemzés rész feladatait külön egységként megvalósító funkcionalitások metódusait különálló statikus osztályokba rendeztem. Ezzel megszüntettem a metódus, osztálytól való függőségét és könnyítettem a metódusok tesztelhetőségét. Például egy

olyan metódus teszteléséné, amely önmagában egyszerű feladatot valósít meg nem szükséges az összetett objektum létrehozása, elegendő csak statikusan végrehajtani a megfelelő paraméterekkel.

Számos teszteset készítése közben problémát okozott, hogy a tesztelendő osztály erős függőséget mutat más objektum értékeivel és viselkedéseivel. Ahhoz, hogy gyorsabban és hatékonyabban haladhassak az egyes funkcionális egységek tesztelésével az említett függőségek gyors zökkenőmentes feloldására a külső objektumok helyettesítését kellett alkalmaznom.

Másik megoldás, a komplex objektumok viselkedésének „hamisítása”, ami ugyanarra a problémára megoldás, miszerint egy-egy viselkedés függőségének kielégítése érdekében nem kötelező bonyolult osztályváltozókkal rendelkező objektum létrehozása. Objektum függőségek esetén a tesztelt modul által használt más moduloktól származó objektumokat a Mockito program segítségével hoztam létre. Ezzel a megoldással egyszerűen tudtam helyettesíteni a szükséges objektumokat. A lapokat klasszifikáló metódusok ellenőrzésénél a laphoz tartozó blokkokat, utánzó objektumokkal helyettesítettem. Ezzel elértem, hogy nem kellett példányosítanom a blokkokat és meghatározni a belső komplex adatstruktúrájukat, hanem csak a lap által aktuálisan használt metódus viselkedését írtam elő. A Mockito lehetőséget biztosít futásidőben a viselkedés meghatározására vagy megváltoztatására is a mock objektumoknál.

A modulteszteléssel kizárólag a belső adatstruktúrák értelmezését és elemzését megvalósító metódusokat vizsgáltam. A triviális működéssel rendelkező AUTOSAR szoftvermodulok konfigurációs adatainak kezelését megvalósító és a riport fájl tartalmát generáló osztályok ellenőrzését a teljes szoftverrendszer vizsgálatának fázisában ellenőriztem.

8.1.1 Tesztkészlet és tesztesetek bemutatása

A tesztesetek készítését a legkisebb egységek metódusaival kezdtem és fokozatosan haladtam az egyre nagyobb funkcionalitást megvalósító egységek felé. Azért így terveztem a tesztek implementálását, hogy az alapvető funkciókat megvalósító metódusokat hibamentesen tudjam használni, a nagyobb modulok esetén.

8.1.1.1 FEE címkalkuláló és elhelyezést meghatározó osztályok

Ehhez a tesztkészlethez három függvényt soroltam be.

Az első függvény paraméterként egy lapkonfigurációkból álló listát kap. A listában szereplő lapok fizikai kezdőcíme és mérete alapján, meghatározza, hogy a lapok között található-e a más alkalmazás által használt memóriarész. A tesztet egyszerűen implementáltam, a tesztparaméterek között szerepel a listában szereplő lapok mérete, és egy vezérlő paraméter, ami alapján a kezdőcímeket kiszámoljuk a teszt előtt lefutó, az előkészítő metódusban. Olyan esetekre határoztam meg a teszteket, amikor a lista négy elemből áll, vezérlő paraméter értékeket tartalmaz, előforduljon olyan eset, amikor nincs egyik lap előtt sem és minden lap előtt van memória hézag, valamint számos speciális kombinációban megjelenik a memóriahézag a lapok között változó hosszal.

A következő metódus feladata eldönteni, hogy a konfigurációból meghatározott lapok beleesnek-e az aktuálisan kiolvasott memóriarészletbe. Paraméterei közt szerepel egy lapleírókból álló lista és a memóriarészlet kezdőcíme és mérete. A lapok kezdőcímeiből és méreteiből, valamint a memóriarészlet paramétereiből eldönti, hogy van-e olyan lap a listában, amelyik nem esik a kiolvasott memóriarészlet fizikai határai közé. Ha létezik olyan lap, amelyik akár egy bájttal is kívül esik a megadott tartományból, a szoftver speciális log objektumba eltárolja a hiba részleteit és törli a lapot a listából. A meghatározott paraméterek alapján a teszt meghatározza a lapok méretét és címeit, és a memóriaelem beállításaihoz a paraméterként kapott értékeket használja. Itt úgy határoztam meg a tesztparamétereket, hogy a kiolvasott memóriarészlet kezdőcíme és végcíme körül is legyen hibás lap, ami kilép a területből.

Ebben a készletben az utolsó metódus, amit teszteltem azért felelős, hogy konfigurált fizikai lapok egyáltalán benne vannak-e a kiolvasott memóriatartományban. Ellenőrzi, hogy nem nagyobb-e a memóriatartomány kezdőcíme a legutolsó lap utolsó bájtjának a címénél, illetve, hogy az első lap kezdőcíme nem nagyobb-e a memóriatartalom utolsó bájtjának címénél.

8.1.1.2 Blokk és lap adatokat adattömbből kiolvasó metódusok készlete

Az adatok tárolását megvalósító és támogató metódusok közül, elsőként két olyan metódust teszteltem, amelyek feladata az, hogy egy blokk- és lap-menedzsmentinformációt tartalmazó bájt tömbökből a leírók megfelelő értékeit

kiolvassa. Talán az okozhat hibát ezeknél a metódusoknál, hogy a leíró adatok más-más méretűek lehetnek, ezért könnyen rossz változóba kerülhetnek a különböző tömbelemek. Itt a teszt lényegében egy diverz implementációt takar, mivel lényegében a teszt a tesztelendő metódus műveleteit hajtja végre. Ez a tesztmódszer arra alkalmas, hogy megvizsgálja a tesztelendő metódusban alkalmazott algoritmus helyes működését, egy diverzív módon megvalósított (de ugyanazt az eredményt adó) implementáció és az eredeti megavalósítás által produkált eredmények egyezősége alapján.

8.1.1.3 FEE blokk és lapklasszifikáló metódusok

Az klasszifikációs állapotok megállapításának módját és a tesztparaméterek meghatározását érintőlegesen mutatom be, mert beszédesek az FEE algoritmusaival kapcsolatban. Az algoritmusok publikálásával, a thyssenkrupp Components Technology Hungary kft. elvesztené technológiai előnyét a versenytársakkal szemben.

Az FEE blokkok klasszifikálását két szinten, két külön metódusban implementáltam.

Az első lépcsőben a blokk-menedzsmentinformációi alapján történik döntés, hogy érdemes-e a továbbiakban foglalkozni a blokk-kal vagy elkülöníthető a megfelelő hibacsoportba. Ennek a metódusnak a tesztelése során a blokk-menedzsment információkat úgy határoztam meg, hogy a klasszifikáció eredmények közül az összes előálljon, legalább egyszer.

A második, magasabb szintű klasszifikálás a lap állapotának meghatározása. Ezt a feladatot végző metódus az alapszinten vizsgált blokkok klasszifikációs eredményei alapján meghatározza a blokkokat tartalmazó lap klasszifikációját. Ennél a metódusnál a teszt paramétereket úgy határoztam meg, hogy a blokklista-paraméter elemei között az érvényes mellett előforduljon az összes lehetséges hibatípussal rendelkező blokk, legyen kizárólag hibás és kizárólag érvényes blokkokat tartalmazó lista.

A következő metódusok a klasszifikálás támogatására szolgálnak, ezek tesztjeit a következőképpen készítettem el.

A következő metódus, a paraméterként kapott FEE blokkok listájában megvizsgálja, hogy található-e a listában érvényes állapotú blokk, amit a metódus bool típusú visszatérési értéke jelez. A metódus ellenőrzéséhez, a teszteset paramétere egy blokk aktuális állapotait tartalmazó lista, amiből a metódus paraméterlistája állítódik

elő, valamint a keresés eredményére elvárt érték, amit az aktuális kimenettel hasonlítok össze a tesztelésnél.

Ebbe a csoportba tartozik az a metódus, ami azt a célt látja el, hogy a blokkleíró lista paraméterelemei közül visszaadja a legutolsó érvényes állapotú listaelem indexét. Ehhez a teszthez használható teljes mértékben az előző bekezdésben szereplő teszt paraméterkészlete, csak itt az elvárt értéknek a legutolsó helyes blokk leíró indexét kell megadnunk a listában.

Az elő-klasszifikálás után és némi ellenőrző metódus eredményeinek elemzésével előfordulhat, hogy törölt memóriatartományt is blokkleírónak és adatnak tekintettünk a kijelölt tartományt reprezentáló bájttömbben. Ezeknek a blokkoknak a törlését végző metódus tesztelését az előzőekhez hasonlóan végeztem, csak itt azt vizsgáltam, hogy a rossz eredményt tartalmazó blokkokat eldobja-e a listából a metódus, a tesztparaméterek megfelelő hangolásával.

8.1.1.4 NvM blokkok konzisztencia ellenőrző metódusok

A konzisztenciavizsgáló metódusok feladata, a 4.1.1 fejezetben ismertetett CRC, StaticId és redundáns blokkok esetén redundancia vizsgálatát takarja. Feltéve, ha a blokk konfigurációjában engedélyezve lett az első két elemző szempont.

Elsőként a CRC ellenőrző metódust teszteltem, itt mint a korábbi teszteseteknél diverz implementációt alkalmaztam, vagyis a tesztparaméterek alapján előállított FEE blokk felhasználói adataiból és nélkülözhetetlen menedzsmentinformációkból állítottam elő az elvárt CRC értékét 8, 16 és 32 bájtos formában. Ugyanezekkel a blokk értékekkel hívtam meg az eredeti tesztelendő metódust. A tesztlépésben összehasonlítottam az elvárt és aktuális eredményeket hasonlítottam össze.

A StaticId ellenőrzésnél a megfelelő tesztparaméterekkel, a felhasználói adat elejére fűztem a statikus blokkazonosítót és ezzel a frissített blokkobjektummal hívtam meg az ellenőrző metódust, a statikus azonosító vizsgálatának jelölése mellett.

A redundancia elemzését implementáló metódus teszteléséhez a tesztben definiáltam két új paramétert, amivel beállítható, hogy a két vizsgált blokk adatai meggyezzenek-e vagy különbözzenek és ezek alapján állítottam elő a vizsgálandó objektumokat. A vizsgált felhasználói adatok méretét, tartalmát, többféleképpen változtatva teszteltem a metódust.

8.1.1.5 NvM blokk klasszifikáló metódusok

A blokkok klasszifikációjához elsőként a konzisztenciaellenőrzéseknek kell elvégződnie és ezután ezekből az eredményekből határozható meg az NvM blokk állapota. A natív és dataset blokkok teszteléséhez, tesztparamétereknek a szükséges konzisztenciavizsgálat eredményeket definiáltam, hogy az ezekkel ellátott NvM blokkal futassam az állapotmeghatározó metódust. A teszt paraméterei között szerepel az elvárt blokkállapot is, amit kézzel adtam meg az egyes paraméter összeállításokhoz.

Mivel csak három bemeneti paraméter határozza (CRC, StaticID, FeeBlockStatus) meg az állapotot, ezért az összes lehetséges bemeneti kombinációra hajtottam végre a tesztet.

A redundáns blokkok állapotmeghatározó metódus tesztjének paraméterlistáját kiterjesztettem a redundanciaellenőrzés eredményével a három alapvető vizsgálaton kívül. Ezzel megdupláztam a tesztvégrehajtások számát, viszont így is kézben tartható mennyiségű tesztparaméter-kombinációt használtam a teszt végrehajtásánál.

8.1.1.6 Moduláris tesztek összegzése

A tesztek nagyon hasznosnak bizonyultak a rejtett hibák felfedezésében és javításában. Emellett, a helytelen paraméter és visszatérési értékek ellenőrzésének szükségességére is rávilágítottak. A tesztek alapján javítottam az összes metódust ahol hibát okozhat egy hibásan átadott null tartalmú referencia.

Minden egyes teszteset tervezésénél és implementálásánál előkerültek olyan eshetőségek, amikre korábban nem számítottam, ezáltal nem kezeltem le.

Sikerült elérnem, hogy az összes tesztelt metódushoz készített teszt, alapvető, összetett és szélsőséges értékeket tartalmazó paraméterkészletekkel is helyes eredménnyel futott le.

8.2 Teljes szoftver validációja

A tesztelés verifikációs részében azt vizsgáljuk, hogy a rendszer tervezése, implementációja helyes-e, az egyes fejlesztési lépésekben használt specifikációknak megfelelően. Ezzel szemben a validáció már rendszerszinten vizsgálja a produktumot és azt igyekszik bizonyítani, hogy a jó rendszert sikerült-e megvalósítani, a felhasználói elvárások alapján.

Az elemző szoftver futtatásának elengedhetetlen feltétele, a bemeneteinek biztosítása. Ehhez szükség van a memóriavermet alkotó BSW modulok (FLS, FEE, NvM) konfigurációs modelljeit tartalmazó fájlokra. Szükséges egy valós hardverből kinyert memória adathalmaz, amit az adott konfigurációval rendelkező memóriamenedzsment szoftvermodulok kezeltek.

A cégnél rendelkezésemre állt egy AUTOSAR memória stack-et tesztelő integrációs tesztrendszer, melynek segítségével előállíthatóak az elemző plug-in bemenetei. Az integrációs teszt a cég által fejlesztett hardveren fut, így a teszt által, a flash memóriában közvetlenül előállíthatóak a saját tesztjeimhez szükséges memória image-ek. Az integrációs tesztrendszer segítségével változatos adatokkal töltöttem fel a releváns memóriaterületet és a Lauterbach debugger és Trace32 PC szoftver segítségével kinyernem az adatokat. Ezek között szerepel számos olyan adat, amelyek helyes működés, illetve helytelen viselkedés eredményeként kerülhetnek a memóriába. Helytelen viselkedés például a blokkírás vagy lapváltás közbeni tápfeszültségelvesztés, vagy az adatot tartalmazó memóriarész meghibásodása, amelyet detektál és korrigál memóriaverem, vagy esetleg olyan meghibásodás, amelyet nem tud detektálni.

A validáció folyamán fekete doboz tesztelési módszer alkalmaztam, ami azt jelenti, hogy az imént ismertetett bemeneti adatokhoz a specifikáció alapján meghatároztam az elvárt kimeneteket, majd összevetettem a szoftver kimeneteivel.

Az integrációs tesztelés folyamata közben, is voltak olyan esetek, ahol szemantikai hibákat kellett korrigálnom. Végezetül, a Flash memória image-eknek és a hozzájuk tartozó konfigurációknak megfelelő elemzési eredményeket kaptam, és sikeresnek könyveltem el az általam implementált elemző szoftvert valós környezetben.

Továbbfejlesztési szempontból érdemes lehetne egy automatizált teszt létrehozása. Feladat, hogy az integrációs tesztesetek lépései alapján felépítsük az elemző szoftver elvárt kimenetét, azaz meghatározzuk a lapok és blokkok tartalmát. Végezetül az elvárt és az elemző szoftver által szolgáltatott adatstruktúra összehasonlításával ellenőrizhető az elemző szoftver helyes működése. A feladat komplexitásának okán ez a feladat nem része, dolgozatomnak.

9 Eredmények értékelése

A feladat elvégzéséhez részletesen megismerkedtem az AUTOSAR Memória stack-et alkotó, vezérlőegységbe integrált memória eléréséért felelős modulokkal (NvM, FEE, FLS).

Elkészítettem az egyes modulokhoz tartozó konfigurációbeolvasó osztályokat, amelyek az AUTOSAR .arxml kiterjesztésű modellfájlok szükséges adatait tárolják el az erre a célra készített Java osztályokba.

Elkészítettem a bináris memóriarészletet tartalmazó fájlt kezelő osztályt.

Megterveztem és elkészítettem a bináris fájlból kiolvasott menedzsmentinformációkat és adatokat elemző osztályokat. Mind az FEE és mind az NvM szintjén egyaránt részletesen kidolgoztam az adat-konzisztencia ellenőrzésére (CRC, statikus ID, …) és az állapotok meghatározására szolgáló metódusokat.

Megterveztem, és létrehoztam egy alapvető HTML riport prototípust, ami elfogadható az elemzés eredményeinek vizualizálására. A prototípusból kiindulva, elkészítettem, ugyanezt a dokumentumot dinamikusan előállítani képes riportgenerátor feladatot megvalósító részegységet, az Xtend nyelv Template Expressions funkcióját implementáló metódusok segítségével.

Többféle eszköz segítségével (statikus kódelemzés, moduláris tesztek) ellenőriztem az elkészült szoftver működését, esetleges hibák meglétét. Fekete doboz teszt módszerrel, iparban is előfordulható bemeneti paramétereket használva megvizsgáltam a teljes szoftver viselkedését a kimeneti riport értelmezésével. Minden verifikációra és validációra használt módszer és technológia alkalmazása mellett is sikerült helyes teszteredményeket elérnem, amik az autóipari vezérlők memóriasérüléseinek felderítésére és vizualizációjára készített szoftver minőségét jellemzik és elfogadhatónak nyilvánítják.

Irodalomjegyzék

[1] Az EEPROM memória álltalános bemutatása:
https://www.pcmag.com/encyclopedia/term/42403/eeprom
(2018. December)

[2] Az Flash memória álltalános bemutatása::
https://www.pcmag.com/encyclopedia/term/43272/flash-memory
(2018. December)

[3] Difference between cikkje az EEPROM és a Flash memóriák különbségeiről:
http://www.differencebetween.net/technology/hardware-technology/difference-between-eeprom-and-flash/
(2018. December)

[4] AUTOSAR hivatalos honlapja:
http://www.autosar.org
(2018. December)

[5] Dr. Pintér Gergely: AUTOSAR alapú autóipari szoftverrendszerek tantárgy: Szoftverkomponensek előadás.
(2017. Február)

[6] AUTOSAR konzorcium: AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf
https://www.autosar.org/fileadmin/Releases_TEMP/Classic_Platform_4.3.1/General.zip
(2018. December)

[7] Faragó Dániel: AUTOSAR alapú autóipari szoftverrendszerek tantárgy: Basic Software - Kommunikáció előadás.
(2017. Február)

[8] Specification of NVRAM Manager AUTOSAR CP Release V3.2.0
Document Identification No.: 033
(2018. December)

[9] Specification of Flash EEPROM Emulation AUTOSAR CP Release V2.0.0
Document Identification No.: 286
(2018. December)

[10] Specification of Flash Driver AUTOSAR CP Release V3.2.0
Document Identification No.: 025
(2018. December)

[11] The Java language specification
https://docs.oracle.com/javase/specs/jls/se7/html/index.html
(2018. December)

[12] Kovácsházy Tamás: Beágyazott rendszerek szoftvertechnológiája tantárgy: JAVA bevezető előadás.
(2018. December)

[13] Hivatalos Xtend dokumentáció
https://www.eclipse.org/xtend/documentation/index.html
(2018. December)

[14] Template Expression leírása az Xtend hivatalos weblapján.
https://www.eclipse.org/xtend/documentation/203_xtend_expressions.html
(2018. December)

[15] HTML 5.3 specifikáció
https://w3c.github.io/html/introduction.html#introduction
(2018. December)

[16] Legfirsebb CSS verzió specifikációja .
https://www.w3.org/TR/2018/NOTE-css-2018-20181115/#intro
(2018. December)

[17] A World Wide Web konzorcium JavaScript oktatásért felelő weplapja.
https://www.w3schools.com/js/
(2018. December)

[18] Legfrissebb Eclipse kiadás dokumentáviója.
https://help.eclipse.org/2018-09/index.jsp
(2018. December)

[19] Parasoft Jtest dokumentációja.
https://alm.parasoft.com/hubfs/69806/New_Pages/Jtest%20Datasheet.pdf
(2018. December)

[20] Coverity Scan hivatalos weblapja.
https://scan.coverity.com/
(2018. December)

[21] SpotBugs leírás.
https://spotbugs.readthedocs.io/en/stable/
(2018. December)

[22] Spock teszt keretrendszer dokumentációja
http://spockframework.org/spock/docs/1.2/index.html
(2018. December)

[23] Mockito teszt keretrendszer dokumentációja.
https://static.javadoc.io/org.mockito/mockito-core/2.23.4/org/mockito/Mockito.html
(2018. December)

[24] JUnit teszt keretrendszer dokumentáció
https://junit.org/junit5/docs/current/api/overview-summary.html
(2018. December)

[25] ISO/IEC/IEEE International Standard for Software and systems engineering--Software testing--Part 4: Test techniques. (doi:10.1109/IEEESTD.2015.7346375.) (2015. December 8.)