MŰEGYETEM 1782

DIPLOMATERVEZÉSI FELADAT

Schmidt Zoltán
szigorló villamosmérnök hallgató részére

Mérőrendszer-fejlesztés mechatronikai termékek rendszer-tesztelési feladatainak ellátására

Az autóiparban mindig is kiemelten fontos szempont volt a megbízható működés és a minőség biztosítása. Ezen célok minél hatékonyabb teljesülése végett mára a fejlesztési folyamatok szerves részévé vált a komplex tesztelés. A különböző teszteljárások nem csak a minőséget igyekeznek biztosítani, de a fejlesztés során keletkező hibák minél előbbi kiszűrésére is rendkívül nagy hangsúly kerül. Bár a tesztek kidolgozása és elvégzése jelentős költségteher a gyártók számára, a hibák minél korábbi fázisban történő felfedezésével jelentős mértékben csökkenthetőek a fejlesztési költségek és a termékek piacra-kerülésének ideje, ami fontos szempont.
A Robert Bosch Kft. AE/EEC osztályán egy BLDC motor meghajtású szivattyú elektronikájának fejlesztése folyik, melyhez jelenleg korlátozottan állnak rendelkezésre tesztelési megoldások. A hallgató feladata, hogy megismerje az említett terméket, majd ez alapján végezze el egy új tesztrendszer fejlesztését és implementálását, mely képes kiszolgálni a szükséges tesztelési igényeket.

A hallgató feladatának a következőkre kell kiterjednie:

- Ismerje meg a tesztelendő termék funkcióit és specifikációit!
- Ismerje meg a termék korábbi generációjánál alkalmazott tesztelési megoldásokat!
- A fejlesztési csoport által meghatározott tesztelési stratégia alapján határozza meg a mérőrendszerrel szembeni követelményeket!
- Mutassa be a tesztelési folyamatokat szabályzó szabványokat!
- A konzulenssel egyeztetve tervezze meg a szükséges mérőrendszert, építse meg, majd ellenőrizze a működését!
- Tervezze meg, majd implementálja a szükséges teszteket, és bizonyosodjon meg a helyességükről!

Tanszéki konzulens: Krébesz Tamás István, tanársegéd
Külső konzulens: Erdei András (Robert Bosch Kft.)

Budapest, 2017. …….

……..
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
Automatizálási és Alkalmazott Informatikai Tanszék

Schmidt Zoltán

MÉRŐRENDSZER-FEJLESZTÉS MECHATRONIKAI TERMÉKEK RENDSZER-TESZTELÉSI FELADATAINAK ELLÁTÁSÁRA

TANSZÉKI KONZULENS
Krébesz Tamás István

KÜLSŐ KONZULENS
Erdei András

Budapest, 2017

Tartalomjegyzék

DIPLOMATERVEZÉSI FELADAT **1**
Összefoglaló **6**
Abstract **7**
1 Bevezetés **8**
1.1 Feladat ismertetése 9
2 Irodalomkutatás **11**
2.1 Tesztelés 11
2.2 ASPICE 14
2.2.1 Történeti háttér és motiváció 14
2.2.2 Folyamat megközelítés 14
2.2.3 V-modell alapú fejlesztés 17
2.2.4 Verifikáció és validáció 18
2.3 TPI Automotive 20
3 Az üzemanyagszivattyú **21**
3.1 Mechatronikai rendszerek 21
3.2 Mechanikai felépítés 23
3.3 Elektronikus részek 27
4 Tesztkörnyezet definiálása **28**
4.1 Követelményanalízis 28
4.1.1 Platform projekt 28
4.1.2 Követelmények 29
4.2 Vector Informatik tesztmegoldások 31
4.2.1 vTESTstudio 31
4.2.2 CANoe 32
4.2.3 VT-System 35
5 Tesztfejlesztés **37**
5.1 Teszt designok elkészítése 37
5.1.1 Újrafelhasználhatóság támogatása 37
5.2 Testek struktúrája 42
5.2.1 Grafikus test design 42
5.2.2 CAPL 43
5.3 Elkészült tesztek 45

5.4 Alkalmazás életciklus-menedzsment eszközök 46
5.5 Továbbfejlesztési lehetőségek 47
6 A tesztrendszer továbbfejlesztése 48
6.1 Kiindulási alapok 48
6.2 CANoe mélyebb működése 50
6.3 A LabVIEW integrálása 53
6.3.1 LabVIEW 53
6.3.2 Általános motiváció 53
6.3.3 Megoldási lehetőségek vizsgálata 54
6.4 Választott megoldás 60
7 Összegzés és értékelés 62
7.1 Kitekintés 62
Köszönetnyilvánítás 63
Irodalomjegyzék 64
Szakmai rövidítések jegyzéke 65
Ábrajegyzék 66
Függelék 68

HALLGATÓI NYILATKOZAT

Alulírott **Schmidt Zoltán**, 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, 2017. 12. 14.

...…….

Schmidt Zoltán

Összefoglaló

A diplomamunkám témája egy olyan tesztrendszer megépítése volt, melynek elsődleges feladata egy BLDC hajtású üzemanyagpumpa rendszertesztelésének végrehajtása. A munka során széleskörű ismereteket szereztem mind a tesztelendő eszközről, a tesztelés elméleti hátteréről, a használt szoftverekről, valamint a fejlesztési folyamatot meghatározó szabványokról.

A bevezetésben kitérek az automatizált tesztelés szerepére, majd bővebben ismertetem a feladatomat ezen a témán belül, illetve felvázolom a diplomamunkám felépítését.

Az első néhány fejezetben az irodalomkutatás alatt szerzett ismereteimet foglaltam össze röviden, először a tesztelés elméletéről, majd az ASPICE fejlesztési folyamatot előíró szabványról.

Az elméleti alapok megszerzése után igyekeztem minél alaposabban megismerni a tesztelendő üzemanyagszivattyú felépítését és működését, majd az összegyűjtött követelmények alapján egy alkalmas tesztrendszert találni hozzá.

A tesztrendszer kiválasztása után a rendelkezésre álló rendszerkövetelmények alapján megvalósítottam a szükséges teszteket a Vector Informatik szoftvereinek segítségével. Emellett a tesztrendszer továbbfejlesztésével foglalkoztam, hogy alkalmassá tegyem azt külső mérőeszközök integrálására, hogy fejlesztéstámogató teszteket is végrehajthassunk a rendszeren.

A dolgozat végén összefoglaltam az elért eredményeket és az esetleges további fejlesztési lehetőségeket.

Abstract

The topic of my thesis is the design and development of a test system that is primarily set up to manage all the system-test activities of a BLDC controlled fuel pump. During my work I gathered comprehensive knowledge about the device under test, the concept of testing and the used softwares, as well as the standards regulating the development process.

In the introductory chapter I discuss the role of automated testing, introduce the task itself, and present the structure of the MSc thesis.

In the first few chapters I summarize the knowledge gathered during the review of the corresponding literature, starting with the background of testing and then followed by the ASPICE standard that regulates the development process.

After obtaining the necessary theoretical basis I tried to gain a deeper understanding of the structure and the functionality of the fuel pump that I tested, to find then a suitable test system based on the collected requirements.

After defining the test system, I created the desired tests based on the available system requirements applying the software tools of Vector Informatik company. Besides that I worked on further improvements on the test system, to make it capable of integration with external measurement devices in order to perform tests for development support.

The thesis ends up with a summary of the results achieved and the potential improvements for the future.

1 Bevezetés

A napjainkban gyártásra kerülő autók szinte minden elemükben tartalmaznak már valamilyen elektronikus alkotórészt, melyek térhódítása láthatóan megállíthatatlan tendenciává vált mára. Egy valóságtól nem túl elrugaszkodott megközelítést alkalmazva tekinthetünk úgy az autókra, mint nagyméretű, különösen komplex beágyazott rendszerekre. Ezek a rendszerek már olyan komplexitási szintet értek el, hogy célszerűbb az egyes részegységeiket önmagukban is külön rendszerként kezelni. Ez az oka annak, hogy az egyes funkciókat megvalósító részegységek fejlesztése külön zajlik, sőt szinte minden esetben különböző beszállítóktól érkező komponensekből állnak végül össze egy-egy autógyár végleges termékei.

A diplomatervemet a Robert Bosch Kft. budapesti fejlesztőközpontjának egy osztályán készítettem, ahol elsősorban az előzőekben említett, különféle elektromechanikus alkatrészek vezérlőegységeinek, szakmai terminológiával élve ECU moduloknak (Electronic Control Unit) a fejlesztése zajlik a különböző autógyárak megrendeléseire.

Ahogy a legtöbb iparág esetén észrevehető napjainkban, így az autóiparra is jellemző, hogy nagymértékben lerövidülnek a fejlesztés ciklusok, ezáltal egyre kevesebb idő áll rendelkezésre magára a fejlesztésre és ezzel együtt a tesztelésre. Ugyanakkor megállapítható, hogy az egyre bonyolultabbá váló termékek esetén egyre nagyobb kihívást jelent a hibák maradéktalan megtalálása. Nem nehéz belátni, hogy az autógyártás esetén egy erősen biztonságkritikus iparágról beszélünk, elég csak belegondolni, hogy egy meghibásodás mekkora anyagi kárral vagy emberélet veszélyeztetésével járhat, éppen ezért a hibák megtalálása és a minőség biztosítása nem lehet alku tárgya. Ez a szemlélet már igen régre nyúlik vissza, szinte egészen az autóipar kezdeteihez, amit a következő, Robert Boschtól származó idézet is megerősít:

„*I would rather lose money than trust*”

\- *Robert Bosch*

A fejlesztési idő szűkössége és a minőség biztosítása között fennálló ellentmondást többek között erőteljes automatizálással igyekeznek feloldani a piaci szereplők. Egy termék fejlesztését már korai fázistól kezdve számtalan tesztelés kíséri nyomon, amihez a megfelelő tesztrendszerek elkészítése és az automatizálás megvalósítása rettentően költséges feladat, de hosszútávon mindenképpen kifizetődő.

Az automatizált tesztelés előnyei, hogy csökkenti az emberi tényező okozta hibalehetőségeket, és biztosítja a tesztek megismételhetőségét. A szabványos automatikus riportgenerálásnak köszönhetően a visszakövethetőség és a feldolgozhatóság mellett fontos szempont az esetleges vevői panaszok kezelésében és adott esetben iránymutatást ad a felelősséget viselni köteles (természetes vagy jogi) személy kilétét illetően. Továbbá nem elvetendő szempont, hogy rengeteg időt és emberi erőforrást spórolhatunk meg az automatizálással, ami hosszú távon a költségek oldalán is megtérülést hoz.

1.1 Feladat ismertetése

A diplomatervem keretében egy elektromos vezérlésű üzemanyagszivattyú fejlesztési folyamatába kapcsolódtam be, ahol az egység különféle teszteléseivel, ezek közt első sorban rendszertesztéléssel és az ahhoz szükséges tesztrendszer összeállításával foglalkoztam. Az eszköz maga két fő részből tevődik össze, ezek:

- a vezérlőelektronika
- és az üzemanyagot továbbító szivattyú

Az én feladatom főként a vezérlőelektronikára terjedt ki, de természetesen nem minden esetben választható szét teljesen a kettő.

A feladatom kiinduló pontja a tesztelés elméletének, a fejlesztési folyamatoknak, és hozzájuk tartozó szabványoknak a megismerése. Első lépésként az elméleti háttérben igyekeztem elmélyedni, kezdve a tesztelés elméletével, amelyhez az ISTQB nemzetközi szoftvertesztéléssel foglalkozó szervezet irodalmai adtak jó kiinduló pontot [1]. A fejlesztési folyamat egy ASPICE nevű szabvány alapján zajlott [2], ezért annak a megismerésével, illetve kifejezetten a tesztelés gyakorlatiasabb megértését segítő TPI nevű szabvánnyal [3] folytattam az irodalomkutatást.

Ezt követően közelebbről megismerkedtem az üzemanyagpumpa felépítésével és az elvárt működésével. Itt különös figyelmet fordítottam a mechanikai működés megismerésére, mert ez nagymértékben befolyásolja a tesztek kiinduló pontját, a követelményeket.

A megfelelő háttértudás birtokában a követelmények összegyűjtése és vizsgálata következett, ami alapján kiválasztásra kerültek a tesztrendszer hardveres és szoftveres alkotóelemei. Itt a Vector Informatik által kínált tesztmegoldás mellett született a döntés,

ami egészen pontosan a vTESTstudio testdesign programból , a tesztek futtatásához használt CANoe nevű programból, valamint a VT-System nevű moduláris teszthardverből állt [4]. A tesztmegoldás definiálása után végül a rendszer összeállítása és tesztek elkészítése következett. A tesztek elkészítése mellett a rendszer továbbfejlesztésével foglalkoztam, hogy alkalmas legyen különböző fejlesztéstámogató tesztek végrehajtására.

2 Irodalomkutatás

2.1 Tesztelés

Mindenekelőtt fogalmi szinten szeretném tisztázni a tesztelést, főként azért, mert egy igen sok mindent magába tömörítő fogalomról beszélünk, ezért célszerű tisztában lenni a mögöttes tartalmaival.

Gyakran előforduló szemlélet a tesztelésben kevésbé jártas emberek között, hogy úgy tekintenek a tesztelésre, mintha az nem állna másból, mint tesztek futtatásából, amelyek adnak egy végeredményt a vizsgált eszközről, hogy az éppen megfelelt-e vagy sem. Természetesen ez is része a folyamatnak, ám csak egy keskeny szelet az elvégzendő munkák közül.

A tesztelés, mint folyamat már jóval a tesztek futtatása előtt megkezdődik, sőt még utána is folytatódik, kezdetben a követelmények feldolgozásával, tesztkörnyezet definiálásával, tesztfeltételek kitalálásával és tesztek elkészítésével, az eredmények birtokában pedig az adatok feldolgozásával és a kapott eredmények értékelésével.

Általánosságban elmondható a tesztelésről, hogy a következő okokból kifolyólag végezzük el:

- Rejtőző hibák megtalálása
- Bizonyosságszerzés a minőség szintjéről
- Információszerzés döntéshozáshoz
- Meghibásodások megelőzése

A tesztelés különböző válfajai általában más-más pontra igyekeznek fókuszálni az előbb felsoroltak közül. A fejlesztés-támogató tesztek (modul-, integrációs-, és rendszerteszt) elsődleges célja például, hogy a fejlesztés minél korábbi fázisában a lehető legtöbb hibát felfedezzék, és így azok minél kisebb költség mellett kerülhessenek javításra. Ezzel szemben vannak a validációs tesztek, melyek során azt szeretnénk megmutatni a vevő számára, hogy a készített termék a várakozásoknak megfelelően működik és teljesíti a kívánt követelményeket. Az előzőektől megint csak eltérő eljárás a regressziós tesztelés, mely során azt szeretnénk biztosítani, hogy a fejlesztés során végzett változtatások nem okoztak új hibákat a rendszerben.

Az évtizedek során összegyűlt teszteléssel kapcsolatos tapasztalatok alapján a következő hét alaptételt fogalmazták meg [5], melyek bármilyen tesztelés esetén helytállóak:

1. A tesztelés csak hibák jelenlétét mutatja meg:
    - A tesztelés képes megmutatni hibák jelenlétét, de nem tudja biztosítani a hibamentességet, csupán csökkenteni a fel nem fedezett hibák valószínűségét.
2. Kimerítő tesztelés legtöbbször nem lehetséges:
    - A triviális esetektől eltekintve a minden bemeneti kombinációra való tesztelés lehetetlen, éppen ezért a legtöbbször kockázatok és prioritások alapján osztják el az tesztelési erőforrásokat.
3. A tesztelést minél korábban kell kezdeni:
    - A hibák mielőbbi felfedezésének érdekében a tesztelést a fejlesztési ciklus lehető legkorábbi fázisában el kell kezdeni.
4. Hibák csoportosulása:
    - Sok esetben a hibák oka a rendszer csak egy kisebb, de komplex részéből ered.
5. Rovarirtó paradoxon:
    - Ha mindig ugyanazokat a teszteket alkalmazzuk, akkor a rendszer egy idő után „immunissá” válik velük szemben, és nem fogunk velük új hibákat találni. Ennek elkerülésére a teszteseteket időről időre felül kell vizsgálni.
6. A tesztelés kontextus függő:
    - Az eltérő környezetek más tesztelést kívánnak meg. Mások az elvárások például egy biztonságkritikus alkalmazás tesztelése esetén, szemben mondjuk egy mobilalkalmazás tesztelésével.
7. A hibamentesség önmagában nem elegendő:
    - A hibák megtalálása és kijavítása önmagában még nem eredményezi, hogy a rendszerünk teljesíteni fogja a felhasználó által támasztott követelményeket.

A teszteket az alkalmazott módszerek alapján 3 nagy csoportba sorolhajuk, ezek alapján megkülönböztethetünk white-box, black-box és gray-box teszteket.

White-box tesztek: tipikusan struktúra alapú tesztek, melyek elsősorban a komponens tesztek szintjén elterjedtek, fő jellemzőjük, hogy a teszt folyamán ismert számunkra a futó kód belseje.

Black-box tesztek: tipikusan specifikáció alapú tesztek, a legtöbb tesztelési fázisban előfordul, de leginkább a későbbi fázisokban jellemző. A teszt alatt a belső működés ismeretlen számunkra, kizárólag a meglévő ki- és bemenetekre koncentrál a teszt.

Gray-box tesztek: az előző két típus keverékeként tekinthetünk rá, elsősorban itt is a ki- és bemenetekre koncentrálunk, a teljes belső stuktúra nem ismert, de rendelkezünk extra információkkal és beavatkozási pontokkal a normál ki- és bemeneteken kívül. Klasszikus példa rá beágyazott rendszerek esetén az integrációs tesztelés.

A felsorolt három típus közül a második kettővel foglalkozom a diplomaterv keretében.

2.2 ASPICE

2.2.1 Történeti háttér és motiváció

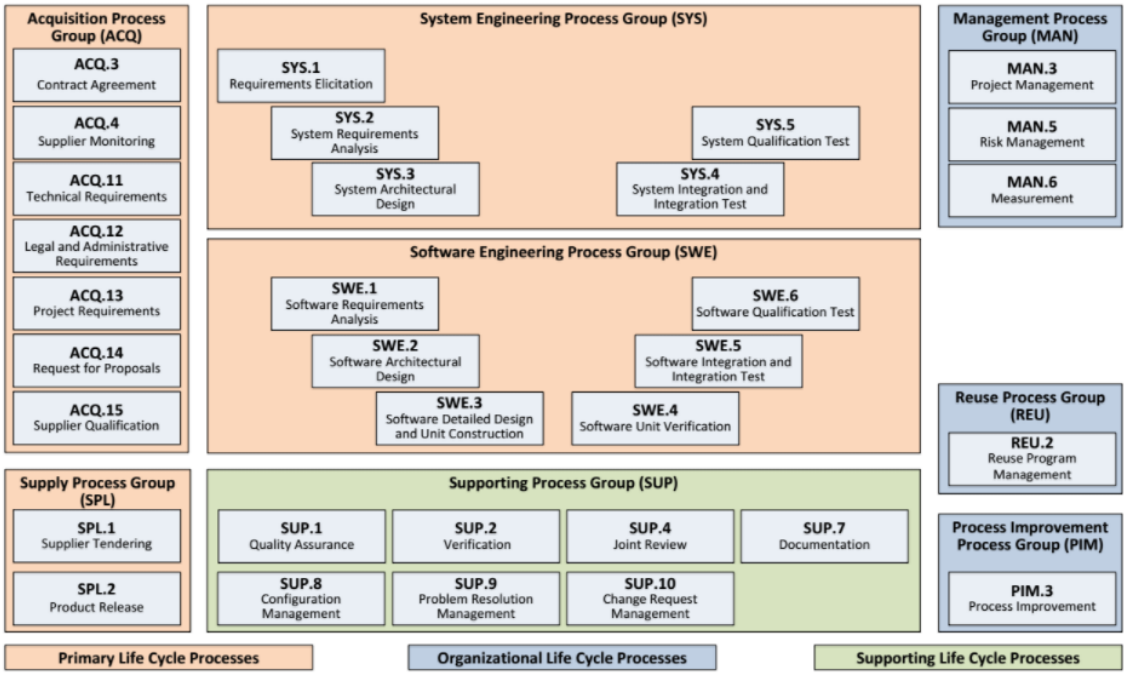
Az autógyártók közt uralkodó versenyben napjainkra egyre nagyobb kihívássá válik, hogy a gyártók fenntartsák az általuk már elért termékminőséget úgy, hogy közben csökkenteni tudják a költségeiket amellett, hogy rövidül a termék piacra kerülésének az ideje. Mindezek mellett nagy nyomás nehezedik rájuk, hogy folyamatos újításokat is alkossanak. Ahhoz, hogy ilyen körülmények között is megállják a helyüket, a gyártók kénytelenek a folyamataikat a lehetőségeknek megfelelően legjobban optimalizálni.

Az alapgondolat az, hogy a fejlesztés végén létrejött termékek minőségét nagymértékben meghatározza az elkészítés folyamatának minősége. A ’90-es évek folyamán erre a gondolatra építve jöttek létre többek közt a CMM és ISO/IEC 15504 (SPICE) szabványok [6], melyek elsődleges célja a szoftverfejlesztési folyamatok szabványosítása és ezáltal minél professzionálisabb szintre emelése. Ezen szabványok eredményességének folyományaként jött létre 2005-ben az Automotive SPICE, röviden ASPICE [7], melyet a nagy európai autógyártókat magába tömörítő Special Interest Group készített, kifejezetten az autóipari fejlesztéseket megcélozva. Mivel a diplomaterv tárgyát képező fejlesztés is ASPICE alapokon nyugszik, azért fontos tisztában lenni a támasztott követelményeivel.

2.2.2 Folyamat megközelítés

Az ASPICE fejlesztési folyamatok értékelési és referencia modelljeként szolgáló szabvány, mely alapján egy 0-5-ig terjedő skálán értékelik a beszállítók fejlesztési folyamatait és útmutatást nyújtanak ezen folyamatok javítására. A legkézenfekvőbb alkalmazása az értékelési szinteknek, hogy egy bizonyos ASPICE szint alatt nem kötnek üzletet beszállítókkal.

Az 1. ábra jól szemlélteti, hogy a szabvány alapján milyen részekre tagolódik egy fejlesztés. Megfigyelhetjük, hogy a szabvány nem csak a szigorúan vett mérnöki fejlesztésre korlátozódik, hanem az egyéb kisérő folyamatokra is kiterjed, például beszállítókra, konfigurációmenedzsmentre vagy projektmenedzsmentre.



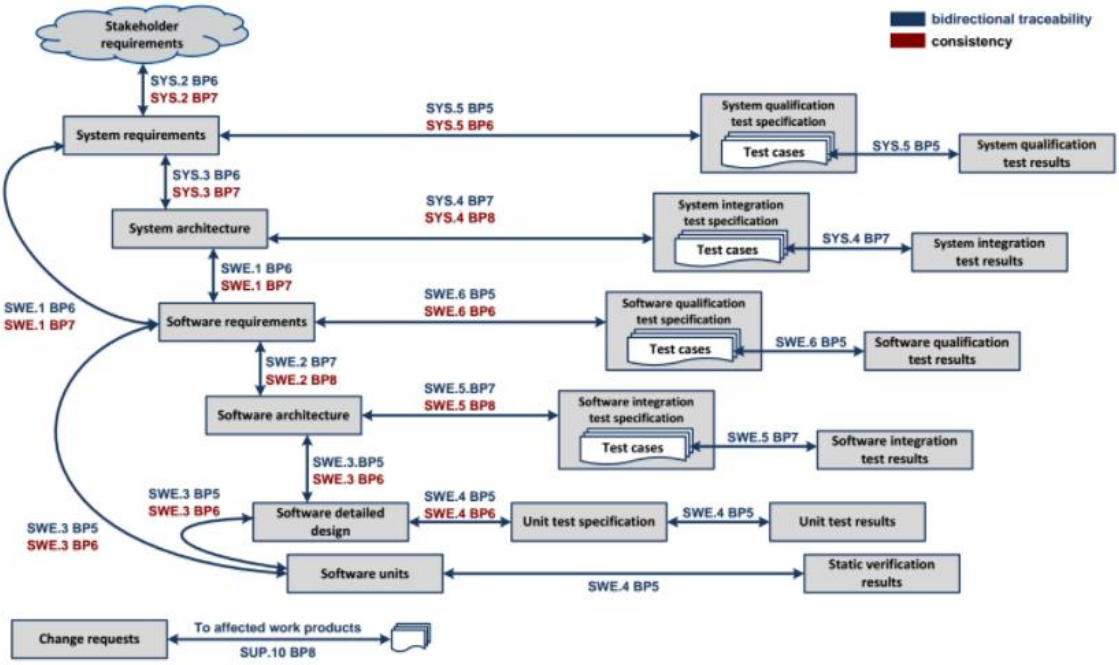
1. ábra Fejlesztési folyamatok csoportosítása (Process Map)

Fontos szempont, hogy a fejlesztés részeit képező tevékenységek megfelelően legyenek definiálva, és ezáltal minden résztvevő pontosan tudja, mi az, ami az ő felelősségi körébe és feladatai közé tartozik. A globalizáció hatására manapság ez a gondolat egyre nagyobb létjogosultságot nyer, hiszen a nagy fejlesztések jellemzően multinacionális cégeknél zajlanak szerte a világban elszórva, mely során meghatározott pontokon felszabdalják a fejlesztést. Jellegzetes példa erre, mikor egyes termékek fejlesztése és tesztelése eltérő csoportok által, gyakran eltérő helyen is történik. Ahhoz, hogy ez megfelelően működjön, tényleg elengedhetetlen a folyamatok „jól-definiáltsága”, ami gyakorlatban megvalósítva általában nem egy egyszerű feladat.

Process reference model	**Process ID**	The individual processes are described in terms of process name, process purpose, and process outcomes to define the Automotive SPICE process reference model. Additionally a process identifier is provided.
	Process name	
	Process purpose	
	Process outcomes	
Process performance indicators	**Base practices**	A set of base practices for the process providing a definition of the tasks and activities needed to accomplish the process purpose and fulfill the process outcomes
	Output work products	A number of output work products associated with each process *NOTE: Refer to Annex B for the characteristics associated with each work product.*

2. ábra Részfolyamatok leírásának formája

Az ASPICE az egyes részfolyamatokat a 2. ábraán látható módon igyekszik standardizálva leírni. Számunkra elsődlegesen fontosként a két teljesítménymutatót emelném ki, melyek a „Base practices” (BP) és „Output work products” (WP) cellák a táblázatban. Ezek közül a BP szekció tulajdonképpen nem más, mint egy pontokba szedett útmutató, melyben összefoglalják, hogy miket kell teljesíteni egy adott részfolyamaton belül, hogy elérjük a kitűzött célját. Ezek az ajánlások azonban meglehetősen általános érvényüknek megfelelően gyakran elég absztrakt előírások, például: „Legyen kidolgozott tesztstratégia!”, amit mindig az adott projekt keretén belül kell megfelelően megvalósítani. A WP felsorolásban találjuk összegyűjtve a részfolyamathoz tartozó dokumentumokat, melyek kézzel fogható bizonyítékul szolgálnak a BP-ban talált javaslatok elvégzéséről. Tulajdonképpen ezek a dokumentumok szolgáltatják az egész rendszer működésének alapját, és kapcsolják össze az egyes részfolyamatokat egy egésszé. A 3. ábra látható, hogy hogyan kapcsolódnak egymáshoz a részfolyamatok a kétirányú követhetőség (bidirectional traceability) és konzisztencia (consistency) jegyében. Ennek a két elvnek a megvalósulása az ASPICE konformancia alapvető feltétele.



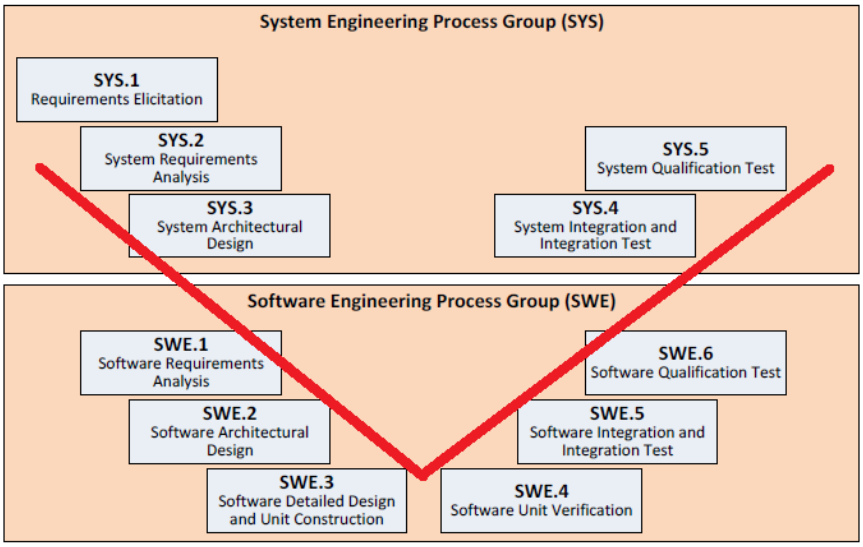
3. ábra Részfolyamatok közti kapcsolatok

A szabványban felvázolt elképzeléssel szemben a leggyakoribb felmerülő kritika, ami tulajdonképpen fő oka annak, hogy még nem világszerte elterjedve alkalmazzák ezt a módszert, annak az elképzelésnek a gyakorlatba való átültetése, hogy minden fejlesztési

folyamat az előírtaknak megfelelően legyen dokumentálva. Ez rettentően sok időt, erőforrást, leegyszerűsítve költséget jelentene a beszállítók számára, ugyanakkor az alkalmazása hosszútávon jövedelmező lenne.

2.2.3 V-modell alapú fejlesztés

Az ASPICE a fejlesztések szigorúan vett mérnöki részeit a gyakorta használt V-modell alapú megközelítés mentén tárgyalja, ezért erről is ejtenék néhány szót.



4. ábra Mérnöki fejlesztés a V-modell alapján

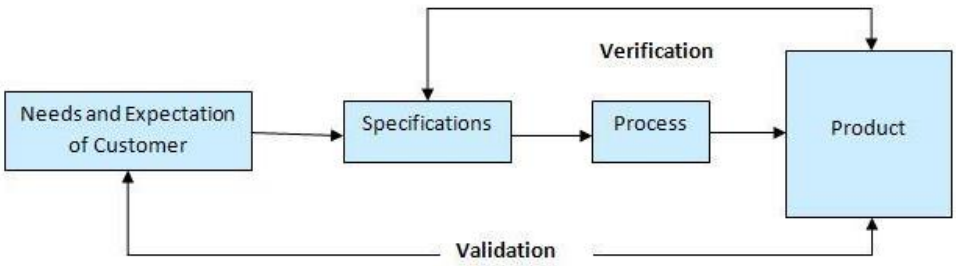
A V-modell egy rendszerfejlesztésnél elterjedt életciklus modell, mely az elvének grafikus ábrázolásáról kapta az elnevezését. Ez az elrendezés látható a 4. ábra, amely a Process Map vonatkozó részének nagyított változata. Az elgondolás alapja az, hogy a „V” betű két szára megfeleltethető a fejlesztés két nagy fázisának, a bal oldal jelképezi a fejlesztést, míg a jobb oldal a tesztelést. Minden egyes bal oldalon található fejlesztési fázishoz tartozik egy tesztelési szint a jobb oldalon, mely azt hivatott biztosítani, hogy az adott fejlesztési szinten végzett munka az elérhető legnagyobb valószínűséggel hibamentes legyen. A modell másik nagy előnye, hogy a különböző munkafolyamatok nem feltétlen szekvenciálisan folynak. Ez úgy nyilvánul meg, hogy a megfelelő követelmények birtokában a szoftvertesztelő már azelőtt elkezdheti a tesztek írását, hogy a tesztelendő funkciók elkészülnének, ezáltal párhuzamosan tud dolgozni a fejlesztővel. Ez a megközelítés nagymértékben le tudja rövidíteni a fejlesztés idejét, ami stratégiai szempontból fontos szempont a piacon, de a valóságban ezáltal természetesen jóval

körülményesebbé válik a fejlesztés, mert több iterációban alakulnak ki a végleges megoldások, és folyamatos kommunikációt igényel a résztvevő felektől.

2.2.4 Verifikáció és validáció

A *4. ábra* könnyen szembe tűnhet, hogy a „V” jobb oldali szára felül csonka. Ahhoz, hogy megértsük, hogy ott milyen lépés foglalhat helyet, körbe kell járjuk a verifikáció és validáció, azaz V&V fogalmak jelentését.

A legtöbb fejlesztés rendszerint azzal kezdődik, hogy a beérkező vevői követelményeket a fejlesztők esetleges belső követelményeivel összevetve lefordítják rendszerkövetelményekre. Ez egy igen sarkalatos pontja az egész folyamatnak, ugyanis a vevői követelmények gyakran pontatlanok, egymásnak ellentmondók lehetnek, márpedig fontos, hogy amit fejlesztünk és az elképzelés, amit a vevő szeretne, a munka végeztével megegyezzen. A fejlesztés alatt végrehajtott teszteket az előzőek ismeretében két részre oszthatjuk annak alapján, hogy milyen követelmények teljesülését igyekeznek bizonyítani, amelynek egyszerűsített folyamatábráját láthatjuk az 5. ábra. A validációs tesztek igyekeznek választ adni arra a kérdésre, hogy „Jó terméket fejlesztek-e?”, azaz megfelel-e a végtermék a vevő elvárásainak. A verifikáció az előzővel szemben arra ad választ, hogy „Jól fejlesztem-e az adott terméket?”, tehát megfelel-e az általunk kitűzött specifikációknak.



5. ábra Verifikáció és validáció

Az előbb említett verifikáció körébe tartoznak a fejlesztést támogató tesztek. Ezek általános jellemzője, hogy viszonylag kevés mintán hajták őket végre, az elsődleges céljuk a fejlesztés helyességének igazolása magunk felé. Ez olyannyira igaz, hogy az itt elvégzett tesztekről a vevőknek nem is feltétlen szükséges értesülniük, nem cél velük a vevők megnyugtatása vagy tájékoztatása. Gyakoriak a különböző határhelyzetek vizsgálata, ahol például szélsőséges hőmérséklet vagy terhelés mellett próbálják felderíteni az eszköz tűréshatárát. Komplexitásuk szempontjából csoportosítva a V alján

helyezkednek el a komponens tesztek, melyek csak egy-egy részre koncentrálnak, ezt követik az integrációs tesztek, melyek több komponens együttes működését ellenőrzik, végül a rendszertesztelés, mely az egész rendszert, tehát az összes komponens együttes működését vizsgálja. A diplomatervem keretében elsősorban az integrációs- és rendszertesztelés szinteket fogom érinteni.

A fennmaradó tesztek, amelyek képzeletben a V jobb felső részén helyezkednek el, a már emlegetett validációs tesztek. Itt a tényleges cél az, hogy bebizonyítjuk, hogy a termékünk teljesíti a vevőtől származó követelményeket. Ezek folyamán a vevő folyamatosan értesül az eredményekről, sőt gyakran ő szolgáltatja az elvégzendő teszteket, esetleg független harmadik fél által ellenőrzi a megfelelő működést. A verifikációval szemben itt általában több mintán végzik el a teszteket, hosszú ideig futtatva, hogy az eredményeket statisztikai módszerekkel is lehessen elemezni.

2.3 TPI Automotive

Miután már jobban körbejártam a tesztelés fogalmát, látható, hogy egy meglehetősen összetett dologról van szó, amiről sokan gondolják azt, hogy átláthatatlan és túl sok időt, illetve pénzt emészt fel, valamint nem lehet vele megfelelően tervezni. Sok vállalat jött már rá arra, hogy ennek legfőbb oka a kiforratlan tesztmegoldások alkalmazása, melyek javítása magával hozza az említett problémák megoldását is. Ezt az összefüggést felismerve hozták létre a TPI (Test Process Improvement) szabványt, mely gyakorlati tudás és tapasztalat által igyekszik megközelíteni a tesztelést. Ennek a szabványnak a kifejezetten autóiparra szabott verziója a TPI Automotive [3].

A szabvány lényegében nézőpontot ad a tesztelés kapcsán alkalmazott eljárások fejlettségi szintjéről, és a fejlődést kézzel fogható, gyakorlati tanácsok formájában segíti elő. A gyakorlatban ez úgy valósul meg, hogy egy nagy, sok jelentést magába tömörítő fogalmat, mint a tesztelés, 21 darab egyszerűen körülhatárolható kulcsterületre bont, amelyek a 6. ábra láthatóak. A feldarabolásnak köszönhetően már sokkal kézzelfoghatóbb dolgokról lehet beszélni. Minden egyes kulcsterülethez megszabott fejlettségi szintek találhatóak, melyek eléréséhez adott feltételeket, checkpoint-okat kell teljesíteni, azok teljesítéséhez pedig gyakorlatból eredő tanácsokat kapunk.

Key Area	**Key Area**
Test strategy	Test functions and training
Life-cycle model	Scope of methodology
Moment of involvement	Communication
Estimation and planning	Reporting
Test design techniques	Defect management
Static test techniques	Testware management
Metrics	Test process management
Test automation	Evaluation
Test environment	Low-level testing
Office and laboratory environment	Integration testing
Commitment and motivation	

6. ábra A tesztelés Automotive TPI által meghatározott kulcsterületei

A TPI Automotive elsősorban azért került megemlítésre, mert segítséget nyújtott az ASPICE által megkövetelt tesztelésre vonatkozó előírások értelmezésénél és gyakorlatba ültetésénél. Az elsődleges célunk nem egy már meglévő folyamat továbbfejlesztése volt, hanem egy új tesztelési folyamat megfelelő kialakítása.

3 Az üzemanyagszivattyú

Korábban már említésre került a tesztelés tárgyát képező üzemanyagszivattyú, melynek szintén minél mélyrehatóbban igyekeztem megismerni a működését. A szivattyú és az általános elhelyezkedése egy üzemanyagtankban a 7. ábra látható. Az üzemanyaggal való kapcsolat folytán valószínűleg még a laikus emberekben is intuitíven adódik, hogy ez egy veszélyes terület. Ez az előfeltevés természetesen egyáltalán nem alaptalan, a termék magas veszélyességi kategóriába tartozik, ezért általában véve szigorú követelmények között végzett teszteket igényel.

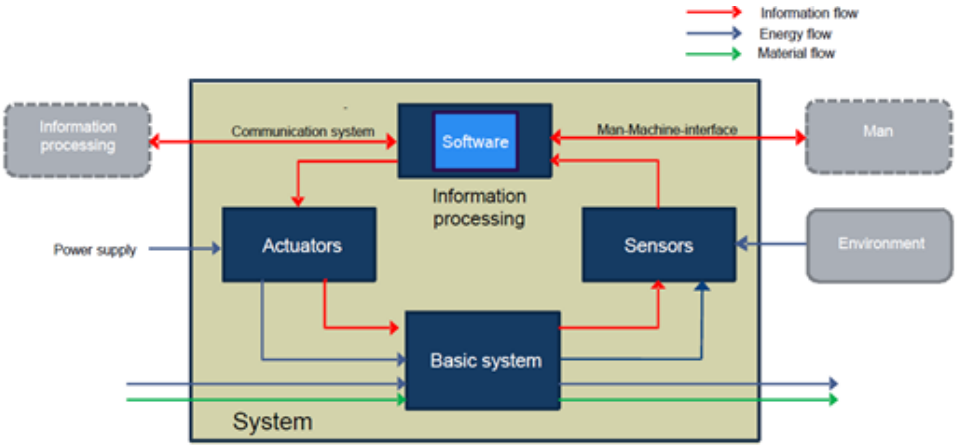
7. ábra Az üzemanyagpumpa elhelyezkedése a tankban

3.1 Mechatronikai rendszerek

Az autókban található részegységeket végignézve a legtöbb esetben valamilyen mechatronikus rendszerrel találkozunk, ezért célszerű tisztában lenni az ilyen jellegű rendszerek általános felépítésével.

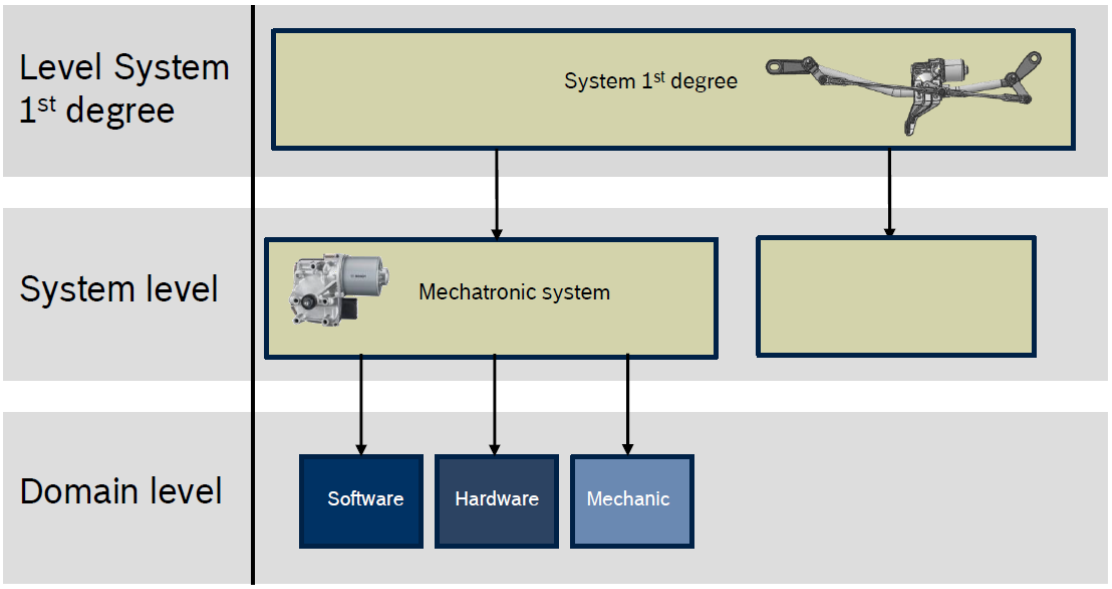
A 8. ábra egy általános mechatronikus rendszert leíró blokkvázlatot láthatunk. A struktúra alapját minden esetben valamilyen energia-, vagy anyagáramlással kapcsolatban lévő kiinduló rendszer alkotja, ehhez csatlakoznak különböző érzékelők és beavatkozók, illetve a hozzájuk kapcsolódó információk feldolgozásához egy központi vezérlőegység. Az alapot képező kiindulási rendszer leggyakrabban elektro-mechanikai

komponensek formájában jelenik meg, például egy motor formájában, mely más mechanikai, hidraulikus vagy pneumatikus részekkel áll kapcsolatban.

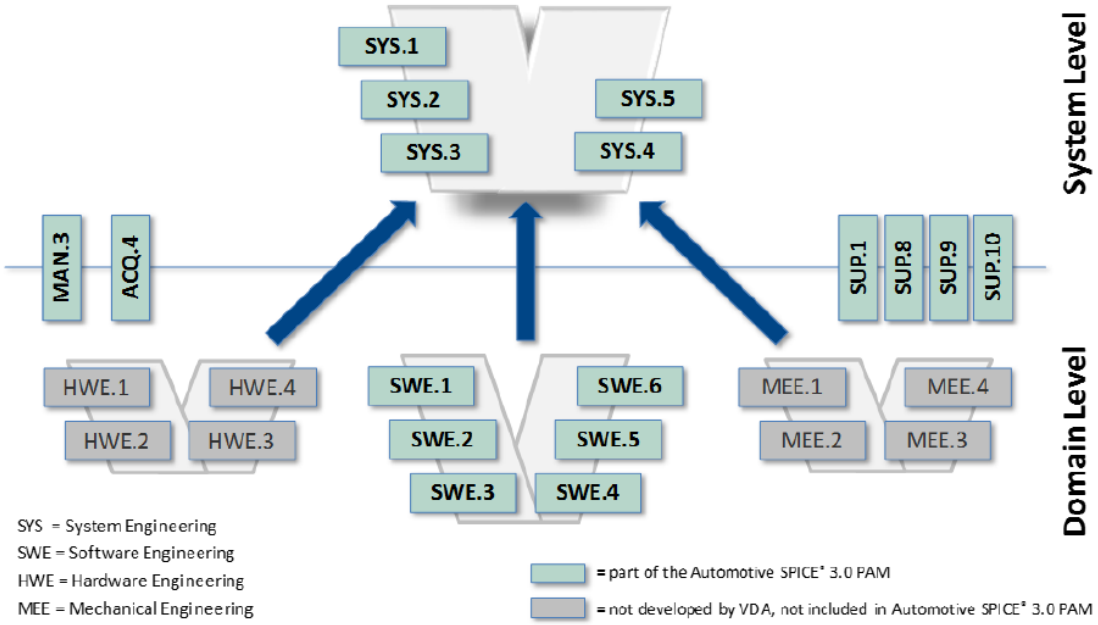


8. ábra Egy mechatronikai rendszer általános felépítése

A mechatronikai rendszerek fejlesztése a komplex felépítésüknek és különböző technológiák alkalmazásának köszönhetően meglehetősen bonyolult lehet, az egyes fejlesztési területek a 9. ábra látható módon válnak szét szoftver, hardver és mechanikai fejlesztésekre a rendszer szinttől lefelé haladva. Ez a V-modell szemléletben a 10. ábra felvázolt módon jelenik meg, amely az úgynevezett „Plug-In” fejlesztési koncepció, hiszen a három fejlesztési terület csupán a V forma aljában tér el, olyan, mintha csak azokat cserélgetnénk. Itt meg kell említeni, hogy ASPICE szempontból kizárólag a szoftverfejlesztés van definiálva.



9. ábra A termékek fejlesztési területekre bontása



10. ábra Plug-In fejlesztési koncepció

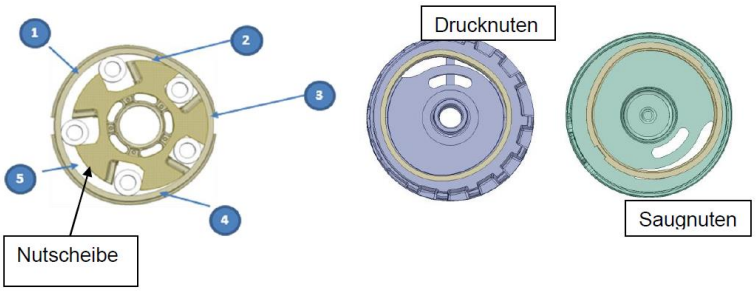
3.2 Mechanikai felépítés

Mivel a fizikai konstrukció és a követelmények szorosan összefüggenek, ezért a megfelelő teszteléshez a mechanikai felépítés beható ismerete nélkülözhetetlennek bizonyult a feladat során. Ami ezt a legjobban alátámasztja, hogy alapvetően kétféle termékről beszélünk az üzemanyagszivattyú kapcsán, méghozzá benzin- és dízelüzemű típusokról. A két termékcsalád alapvetően eltér az alkalmazott szivattyúzási elv alapján, melynek köszönhetően teljesen más követelményeket kell teljesítenünk a különböző ECU-k esetén.

A *11. ábra* láthatjuk a két szivattyú típus robbantott ábráját, a dízeles (felül) majd a benzines (alul) verziót. A benzines szivattyú a hagyományos és közismert turbina elven működik, annak egyszerűsége miatt nem is ejtenék róla több szót, a dízel ezzel szemben egység-térfogat szállítás elvet alkalmaz, melynek megértése fontos.

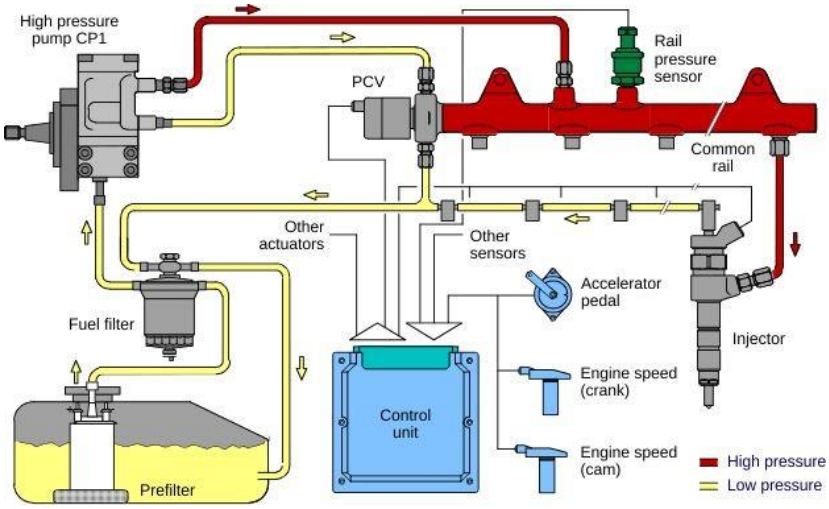
11. ábra Felül dízel-, alul benzinpumpa robbantott ábrája

A dízel szivattyú szívási elvét megvalósító elemei a 12. ábra láthatóak, mely alapvetően a csúszólapátos szivattyúk továbbfejlesztett verziója. A szivattyúk hengeres kialakítású házában egy excentrikus elhelyezésű tengely körül forog a forgórész, melynek radiális hornyaiba sugár irányba elmozdulni képes fém gyűrűk helyezkednek el. Működés közben a centrifugális erő a gyűrűket a szelence széléhez nyomja, a horony térfogata pedig forgás közben a szívónyílástól kezdve fokozatosan nő, majd a nyomó nyílás felé csökken, így a horony melletti bemaráson keresztül beáramló folyadékot el képes szállítani. A szimmetrikus kialakításnak köszönhetően könnyen belátható fizikai megfontolások alapján, hogy minden gyűrűhöz tartozó körcikk egységnyi anyagáramot szállít, ezért az anyagáramlás egyenes arányban áll a motor fordulatszámával. Ennek a mérésre vonatkozó következménye, hogy nem szükséges közvetlenül anyagáram mérést terveznünk a tesztrendszer kialakításánál, hiszen elég a fordulatszámmal vett kapcsolatát ismerni. Azaz egy egyszeri karakterisztika felvétele után elég a fordulatszám ismerete. A zárt motorház miatt a fordulatszám mérése sem lenne feltétlen egyszerű, de itt szintén élhetünk egy egyszerűsítéssel, mivel az alkalmazott szivattyúban BLDC motor van. A BLDC motor esetén igaz az, hogy az áram arányos a nyomatékkal, míg a feszültség a fordulatszámmal, tehát ezen gondolatmenet alapján az anyagáramlás mérését feszültségméréssé tudtuk redukálni, persze ennek gyakorlati megvalósítása pontos, jól átgondolt elrendezést és műszereket igényel.



12. ábra A dízelpumpa főbb alkatrészei

A dízeles rendszer nem csak a szívási elvben tér el, hanem az egész üzemanyagszállítási rendszert tekintve. A fejlesztett rendszer egy Common Rail, azaz közös nyomócsöves befecskendezési rendszer részét fogja képezni, ennek a koncepcióját mutatja be a 13. ábra.



13. ábra Common Rail üzemanyag-rendszer vázlata

Az üzemanyag útját végigkövetve először az általunk fejlesztett szivattyú juttatja el a dízelt egy esetleges szűrőn keresztül egy nagy nyomású pumpához, majd az 1000 bar fölötti nyomást előállítva juttatja el a megfelelő üzemanyagmennyiséget a közös nyomócsőbe. Számunkra fontos szempont egyrészt, hogy a nagynyomású szivattyúban a buborékképződés meggátolása miatt 6 bar körüli nyomást kell biztosítanunk a bemenő oldalon, azaz az üzemanyagszivattyúnk nyomó oldalán, ami számos követelményt von magával, erre egy szemléltető példa a manapság elterjedt Start-Stop funkció alatti működés. Megállás esetén a szivattyú is leáll, és egy szelep biztosítja, hogy megmaradjon a nyomóágban előállított nyomás, azonban tesztelés szempontjából fontos lesz az a határhelyzet, amikor az újraindítás esetén felpörgő motor nem lassan építi fel a nyomást, hanem egyből 6 bar nyomással szembesül a nyomó oldalon. Ez a példa is jól szemlélteti,

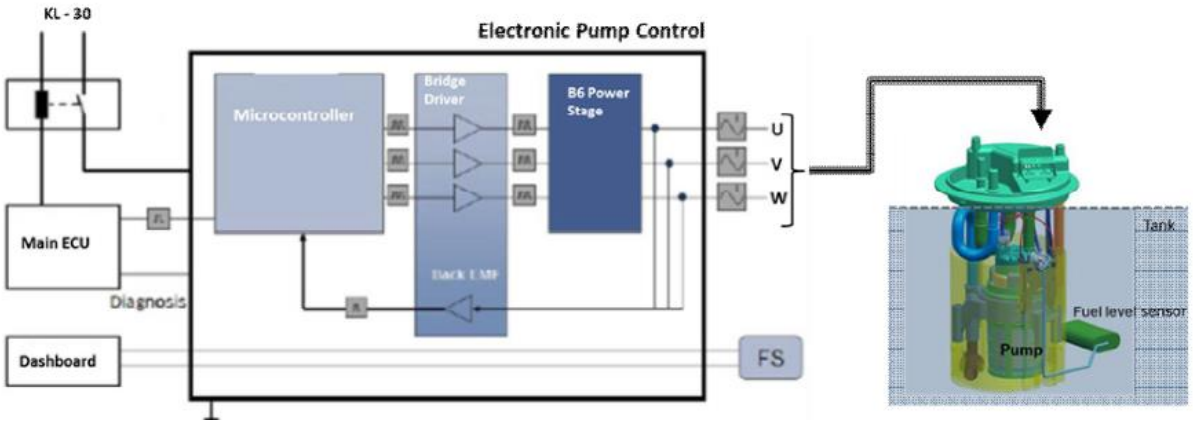
hogy a teszteléshez elengedhetetlenek a rendszerszintű ismeretek, sőt, akár a nem feltétlen szorosan kapcsolódó funkciók ismerete is.

A megfelelő nyomás biztosításán kívül egy elsőre meglepően nagy mennyiségű anyagáramlást is biztosítania kell a szivattyúnak, ugyanis ahogy a 13. ábra is látjuk, a nagynyomású szivattyú, a közös nyomócső és a befecskendezők hűtése is az üzemanyag segítségével történik, ehhez pedig nagyságrendileg néhány tíz liter/perc mennyiségű üzemanyagot kell áramoltatnunk.

3.3 Elektronikus részek

A számomra legfontosabb rész az elektronikus vezérlő, melyhez elsősorban egy előző generációs termék dokumentációi álltak rendelkezésemre, ennek egyszerű vázlata látható a 14. ábra. Ezen koncepció megismerése jó alapot nyújtott az új eszközhöz készült tesztrendszerhez, hiszen egyes jellemzők nem változtak jelentős mértékben, illetve a működés néhány implementációs különbségtől eltekintve nagyjából megegyezett a két generációnál.

Láthatjuk, hogy a vezérlő fő elemei egy mikrokontroller, egy MOSFET-ekből álló B6 teljesítmény fokozat, amivel előállítjuk a motor fázisait, illetve egy bridge driver a teljesítményfokozat meghajtásához. Az ECU relén keresztül kapcsolt tápellátással rendelkezik, és keresztül van vezetve rajta az üzemanyagszint érzékelő jele. A mikrokontroller PWM vonalon keresztül kapja meg a kívánt fordulatszámot, és egy analóg diagnosztikai vonal segítségével képes az egyes hibák jelzésére.



14. ábra Az elektromos vezérlő blokkvázlata

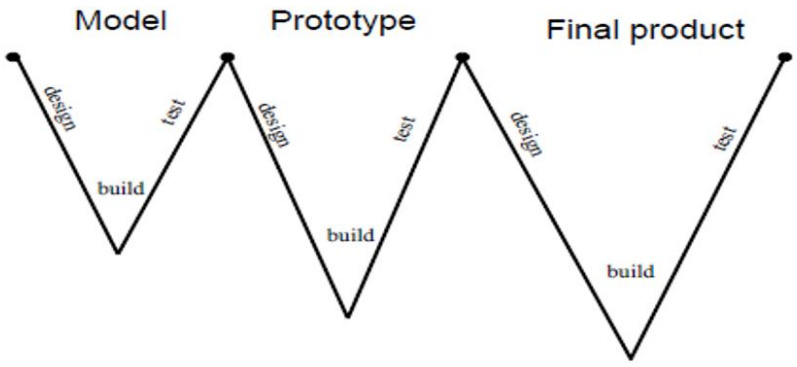
4 Tesztkörnyezet definiálása

4.1 Követelményanalízis

A követelményanalízis jelen esetben nem takar mást, mint hogy összegyűjtöttem az elkészítendő tesztrendszerrel szemben támasztott követelményeket, amihez alapvetően két helyről érkeztek megkötések. Az egyik ilyen megkötés az ASPICE konformancia előírásainak teljesíthetősége. A másik forrás maga a termékből eredő követelmények összessége, azaz, hogy milyen mennyiségeket, milyen módszerekkel és milyen pontossággal kell mérnünk, illetve szimulálnunk. Ehhez fontos ismerni az osztályunkon alkalmazott fejlesztési stratégiát, amit a következőben ismertetnék is.

4.1.1 Platform projekt

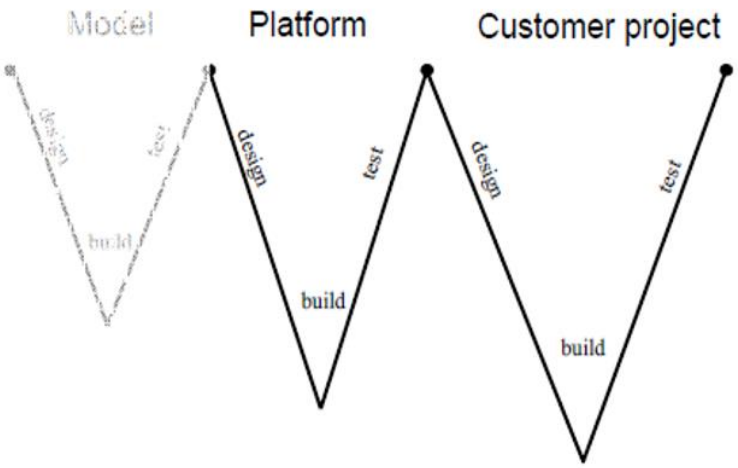
Az üzemanyagszivattyú fejlesztése a cégnél platformprojekt formájában zajlik. Ez gyakorlatban azt jelenti, hogy az eszköz fejlesztését nem egy adott vevő számára kezdik el, hanem egy fennálló, lehetséges piaci igényre, jelen esetben arra alapozva, hogy szükség lehet modernebb üzemanyagszivattyúkra a piacon. Ezt természetesen bonyolult piackutatások és tárgyalások alapozzák meg, de egyszerűsítve kijelenthető, hogy nincs szerződött vevő a fejlesztés kezdetekor. Könnyen gondolhatnánk, hogy ez a projektmenedzsment dolga és a tesztelést nem érinti, de ez nem igaz.



15. ábra Háromfázisú fejlesztési modell

A 15. ábra látható egy megszokott, normális fejlesztési stratégia lefutása, ahol a vevő elképzelései alapján először építenek egy modellt, ha az megfelelő, készítenek egy prototípust, végül pedig a kész terméket. Ezzel szemben a 16. ábra a platform projekt elvét láthatjuk, ahol a modellkészítést jellemzően kihagyják, vagy rendkívül felszínesen

végzik . Helyette egyből egy univerzális platformot készítenek, majd végül azt szabják át az adott vevő kívánságának megfelelően. A hangsúly leginkább az univerzálison van, mert ez azt jelenti, hogy ha több érdeklődő jelentkezett a platform készítésének fázisában, akkor az ő összes követelményüknek az unióját kell megvalósítani, és ami számunkra lényeges, tesztelni. Erre egy triviális példa, amikor az egyik vevő csak PWM alapú vezérlést igényel, míg egy másik valamilyen kommunikációs busz használatát (CAN, LIN, FlexRay...).



16. ábra Platform fejlesztés modellje

4.1.2 Követelmények

Első körben igyekeztem az eszközből származó követelmények összegyűjtésére, ezt az előbb leírtak alapján természetesen platformra vonatkozóan, majd ezekhez rendre mérési- és tesztmegoldásokat rendeltem. A 17. ábra látható módon három kategóriát különböztettem meg:

17. ábra A megvalósítandó feladatok három csoportja

Az első csoportba kerültek azok a jelek, melyekre az eszköznek szüksége van a normális működéshez, mint például a tápellátás. Bizonyos esetekben egyértelmű volt a feladat, pont mint a tápellátás esetén, hogy szükség lesz egy tápra, de a fordulatszám beállításához már PWM jel generálását és kommunikációs buszok kezelését is lehetővé kellett tenni az eltérő vevői igények miatt.

A második oszlopban találhatóak azok a mérendő mennyiségek, melyeket tesztelés szempontjából mérnünk kell. A feszültség/áram szintén triviálisan adódott, a nyomás, hőmérséklet és anyagáramlás mérési problémáját pedig igyekeztem minél egyszerűbb megoldássá redukálni, lásd a dízelszivattyú működési elve alapján történő egyszerűsítést az anyagáramlás esetén.

A harmadik oszlop a működési környezet szimulációja, az első oszlopban megadott elektromos jeleken kívül. Az itt felmerülő problémák általában már különlegesebb tesztmegoldást igényeltek, például a nyomás szabályozásának a szükségessége, vagy annak a kérdése, hogy a tesztüzemanyag használata milyen mértékben szükséges és mit von maga után.

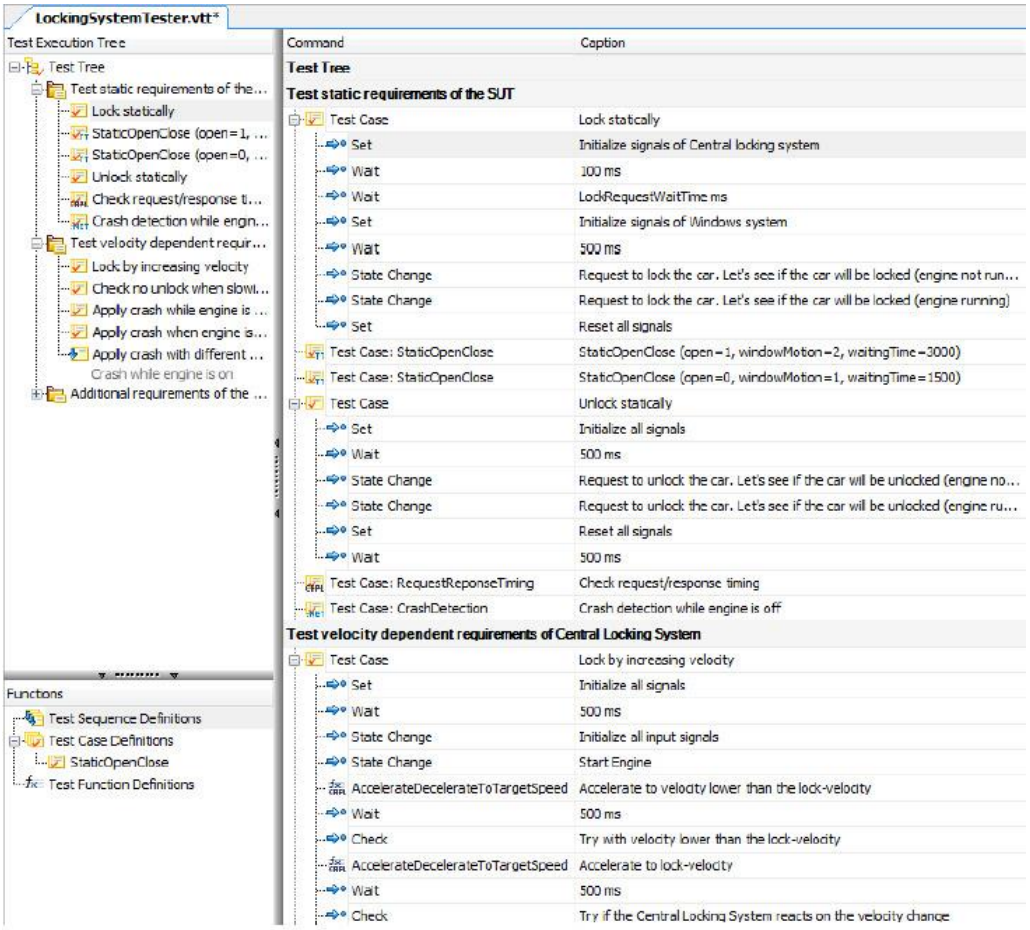
A termékkel szemben támasztott követelményeken kívül fontos szempont volt, hogy az ASPICE által megkövetelt alap elvárásokat minél jobban teljesítse a kiválasztott tesztrendszer.

4.2 Vector Informatik tesztmegoldások

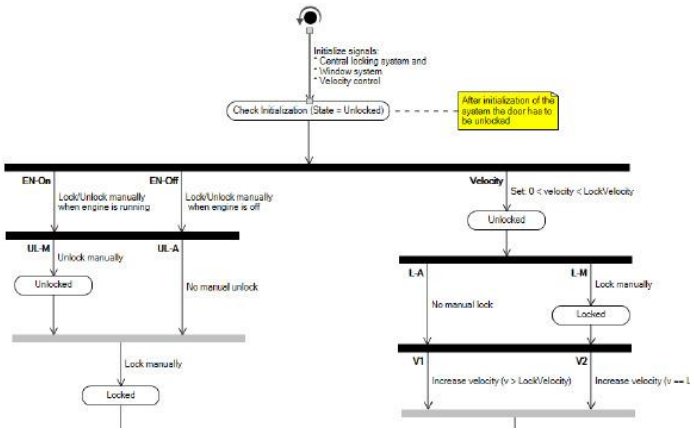
A végső döntés a Vector Informatik által nyújtott megoldások választására esett, itt a vTESTstudio és CANoe programokkal ismerkedtem meg közelebbről, illetve a VT-System teszthardverrel [4].

4.2.1 vTESTstudio

A vTESTstudio egy kifejezetten beágyazott rendszerek tesztfejlesztéséhez létrehozott integrált, sokrétű környezet. A programban különböző testdesign nyelveken írhatunk teszteket, rendelkezik CAPL és C# editorral, szöveges alapú táblás editorral, valamint elérhető hozzá egy folyamatábra alapú grafikus editor opció is, amelyeket a 18. ábra, és a 19. ábra mutat. Más szoftverekhez, például a National Instruments TestStand szoftveréhez képest itt csak tesztszekvenciákat készíthetünk, azonban azok futtatását már a CANoe programkörnyezetben tudjuk végrehajtani, a különböző módokon készített tesztekből egy CANoe programban futtatható TestUnit-ot állít elő a program.

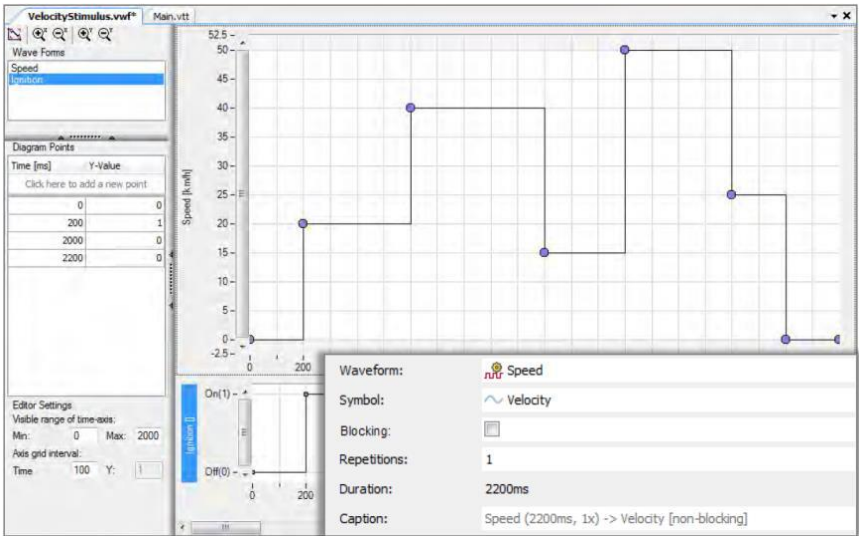


18. ábra Szöveges alapú táblás tesztszerkesztő



19. ábra Grafikus szerkesztőfelület

A program rendelkezik egy beépített jelalak készítő modullal, ezzel jó támogatást nyújtva az rendszertesztelés részét képező elektromos tesztek implementációjához, amelyet a 20. ábra mutat. Az editorban 1 ms-os felbontással készíthetünk hullámformákat, melyeket jelként elmentve tetszőlegesen használhatunk máshol, a környezeten belül éppúgy, mint CANoe-ban.



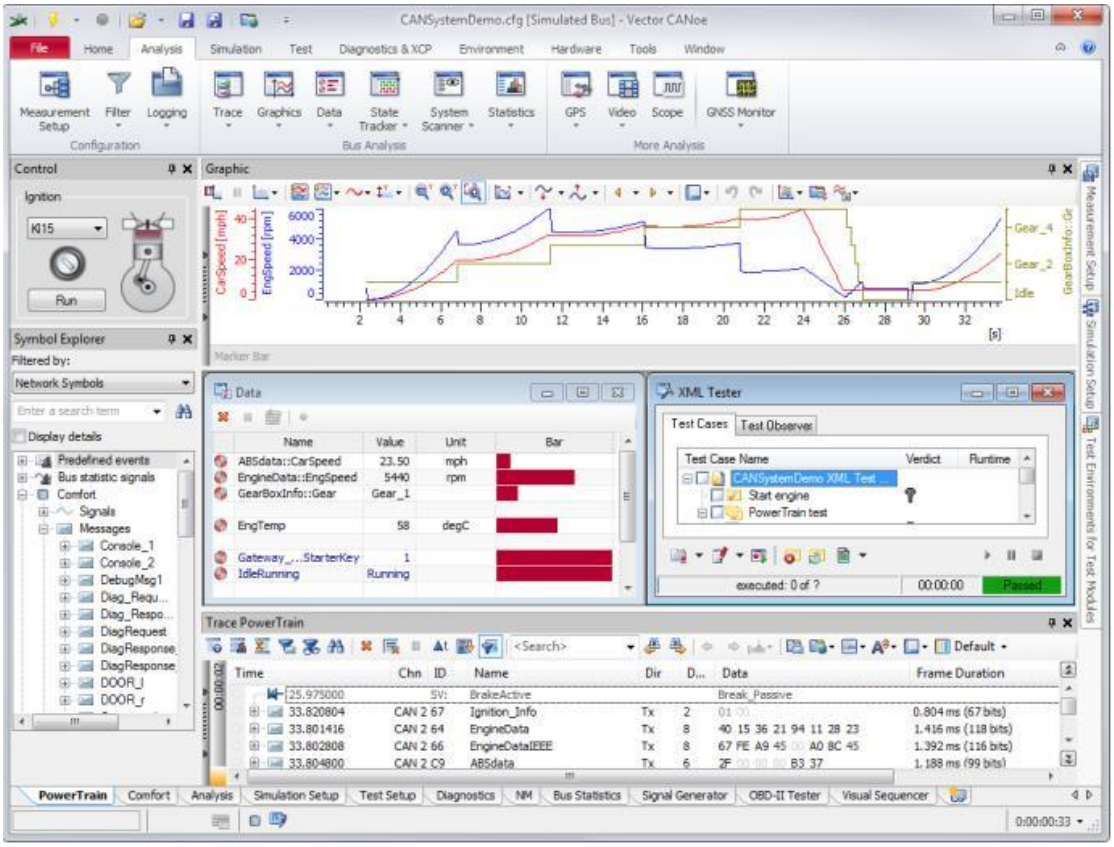
20. ábra Beépített hullámforma-szerkesztő

4.2.2 CANoe

A CANoe a Vector által fejlesztett sokoldalú szoftver, mely segítséget nyújt az autóelektronikai eszközök fejlesztéséhez, vizsgálatához és elemzéséhez. Lehetőségünk van benne teljes autós hálózatok modellezésére, diagnosztikai műveletek végrehajtására, kommunikáció monitorozására és tesztprogramok futtatására is, továbbá lehetőséget

biztosít a teszteredmények html fájlba való mentésére, a buszforgalom naplózására és még számos egyéb hasznos funkcióra.

Az automatizált tesztek megvalósításához erőteljes támogatást nyújt tehát a CANoe, az ehhez tartozó funkcionalitásokat pedig az úgynevezett TFS-ben (Test Feature Set) fogja össze. A TFS eszköztárában helyet kapnak a tesztprogramok létrehozására alkalmas CAPL (CAN Access Programming Language) és XML nyelvek, és hozzá tartozó funkciók, de a tesztek eredményeiről készült jelentések generálása is a TFS feladatkörébe tartozik.

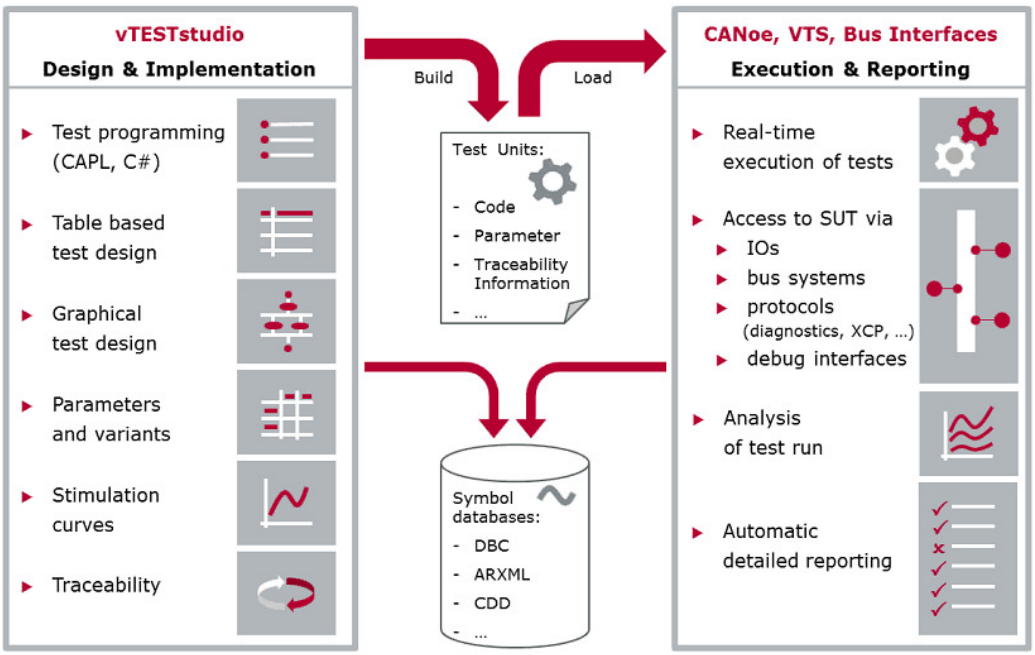


21. ábra A CANoe felhasználói felülete

4.2.2.1 vTESTstudio + CANoe integrálódása

Miután láttuk, hogy a vTESTstudio kizárólag a tesztek implementálására szolgál, és a végrehajtás ettől szeparáltan történik, ebből az következik, hogy a két program ilyen módon kényszerűen összefonódva használható tesztelésre. Egy teszt előkészítésénél először a CANoe környezetben kell létrehoznunk egy új konfigurációt, majd azt elmentve a vTESTstudioba is tudjuk importálni. Az importálás után a tesztkészítésnél közvetlenül elérjük a CANoe-ban definiált változókat, szimbólumokat, és a hozzájuk tartozó

funkciókat. A rendszert további funkcionalitásokkal bővíthetjük a projektfájlhoz hozzáadott Shared Library-k megadásával, amiket a fordító a Test Unit-ba fordít majd.



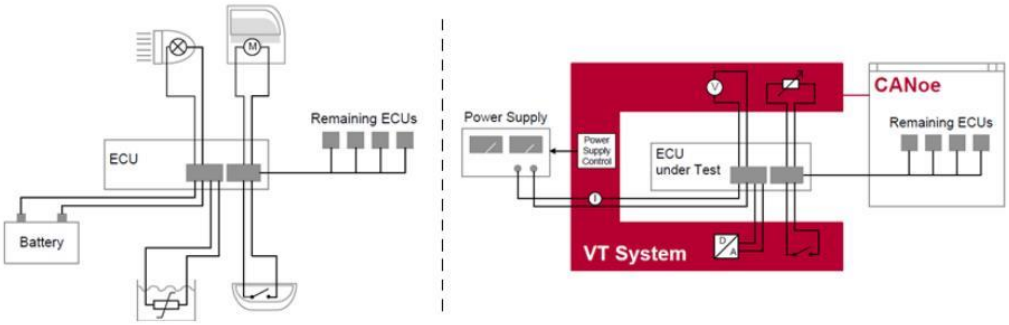
22. ábra CANoe és vTESTstudio közti kapcsolat

A két program együttese jól támogatja a test requirement management-et is, például az egyik legelterjedtebb követelménykezelő szoftverrel, az IBM DOORS nevű programjával [8] együtt működve. A programhoz készített pluginnal az előírt tesztkövetelmények közvetlenül importálhatók a vTESTstudioba, ahol hozzárendelhetjük az egyes elkészített teszteket, majd a CANoe-ban történő végrehajtásuk után az eredményeket exportálhatjuk onnan DOORS-ba, így sokkal egyszerűbb a tesztek teljesítésének nyomon követése. Ez a tulajdonság nagymértékben támogatja az ASPICE kétirány követhetőségének elvárásait, ami az egyik döntő szempont volt a választásnál.

4.2.3 VT-System

23. ábra VT-System moduláris mérőrendszer

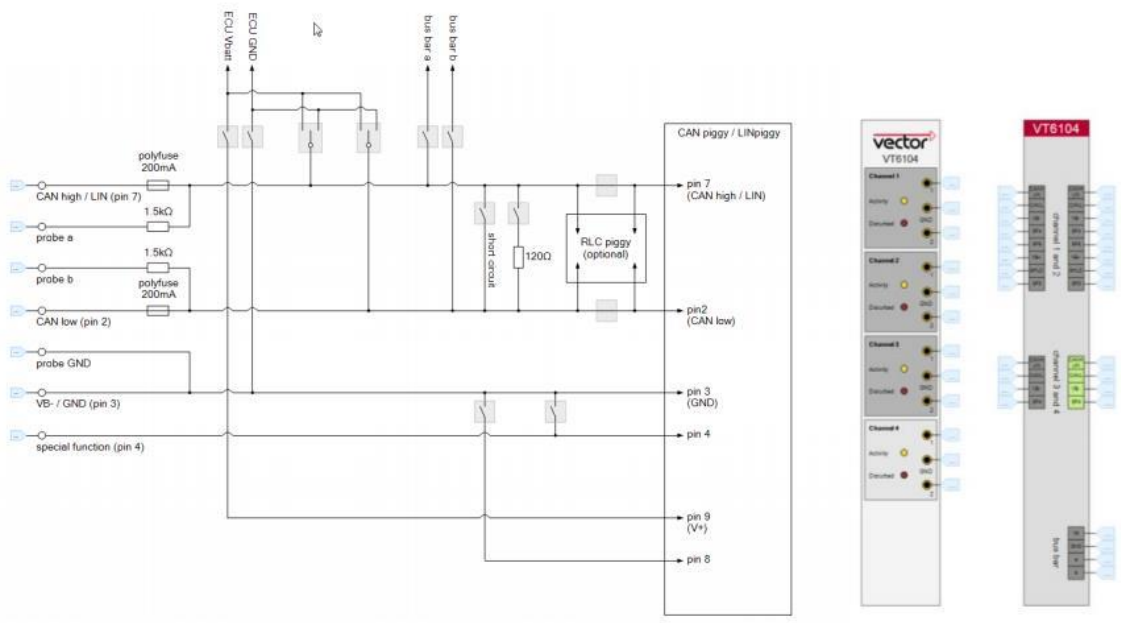
A Hardware-in-the-loop tesztelés támogatására rendelkezésre áll a Vector cég VT-System nevű moduláris teszthardver rendszere. A hardware-keret tetszőlegesen ellátható különféle bővítőkártyákkal az aktuális mérési elrendezésnek megfelelően, a relékártyáktól kezdve különböző mérést végző, stimulus generáló, vagy éppen kommunikációs csatornát biztosító kártyákkal.



24. ábra HIL elrendezés

4.2.3.1 VT-System és a CANoe

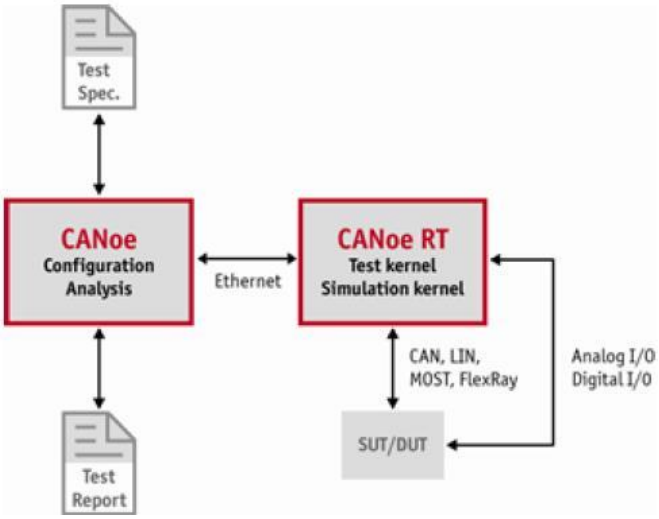
A VT-System tökéletesen integrálódik a CANoe környezetbe, csatlakoztatás után a tesztkörnyezet automatikusan felismeri az elérhető VT modulokat, illetve azok csatornáit és belső reléit, amiket aztán rendszerváltozók formájában tesz elérhetővé a fejlesztők számára. A vTESTstudio-ba történő integrálás után ott is szabadon használhatjuk ezeket a jeleket a tesztkészítés során. CANoe-n belül rendelkezésre áll egy manuális kezelőfelület is a modulokhoz, az egyes kártyákhoz az itt bejelölt kapcsolókat és kimeneteket állíthatjuk, amelyre a 25. ábra láthatunk példát.



25. ábra A CANoe manuális kezelőfelülete a VT-System moduljaihoz

4.2.3.2 Valós idejű működés

Ha a követelmények megkövetelik a valós idejű tesztelést, a megfelelő modul birtokában lehetőségünk van a VT-System real-time működtetésére. Ebben az esetben a PC Etherneten keresztül csatlakozik a hardverhez, itt csak konfigurációt és az adatok feldolgozását végezzük, míg a CANoe RT magja a VT-Systemen fog futni, végrehajtva a szimulációkat és tesztvezérlést. Ezt az elrendezést láthatjuk a 26. ábra.



26. ábra A Valós idejű működés koncepciója

5 Tesztfejlesztés

5.1 Teszt design-ok elkészítése

Hosszú és új ismeretekkel teletűzdelt út vezetett odáig, hogy az első teszt elkészítéséhez végül hozzáfoghattam, keresztül a tesztelés és fejlesztési folyamat elméletén, a mechatronikai rendszerek működésén és az üzemanyagpumpa jellemzőin át. A felkészülés utolsó pontja az volt, hogy bővebben megismerjem a választott szoftverek használatát és a rendelkezésre álló eszköztárukat.

5.1.1 Újrafelhasználhatóság támogatása

Az újrafelhasználhatóság egyrészt magától értetődően azért fontos, mert rengeteg időt és pénzt spórolhatunk meg vele, aminek köszönhetően jó piaci pozíciót biztosíthat a cégünknek. Ha egy bevált megoldást képesek vagyunk magas szintre fejleszteni és azt több helyen használni, akkor ebből hosszú távon a legtöbbször még akkor is profitálunk, ha ez az elején nagyobb többletköltséggel vagy fejlesztési idővel is járt.

Egy másik érv emellett szintén a hatékonyság szempontjából közelíti a témát, és érinti a munkamegosztás szemléletét is. Az emberekről általánosságban kijelenthető, hogy szeretnek rendszerben gondolkodni, ismerős sémákat ráhúzni a dolgokra. Ennek a ténynek a tudatában sokkal hatékonyabbá tehetjük a közös munkát, ha a rendszereinket, jelen esetben például a tesztjeinket egy saját, jól megalapozott forma alapján készítjük el. Ha ezt sikerül megvalósítani és meggyökereztetni a használatát, sokkal könnyebb lehet egy esetleges harmadik fél, például külföldi Test Center bevonása a fejlesztésbe, vagy az újonnan érkező munkatársak megismertetése a projektekkel, hiszen egy projekten keresztül megismerhetik az összes többi projekt alapfelépítését is.

Szeretnék visszautalni a mechatronikus eszközök koncepciójára, ugyanis kínálja magát a lehetőség, hogy a hasonló jellegű termékek fejlesztése esetén a közös jellemzőkre próbáljunk univerzálisan használható megoldásokat készíteni. A mechatronikai rendszerek megismerése során szerzett átfogó ismeretek jól jöttek az általános és eszköz-specifikus szintek elválasztásánál, ezeknek a megkülönböztetésére hoznék fel két példát:

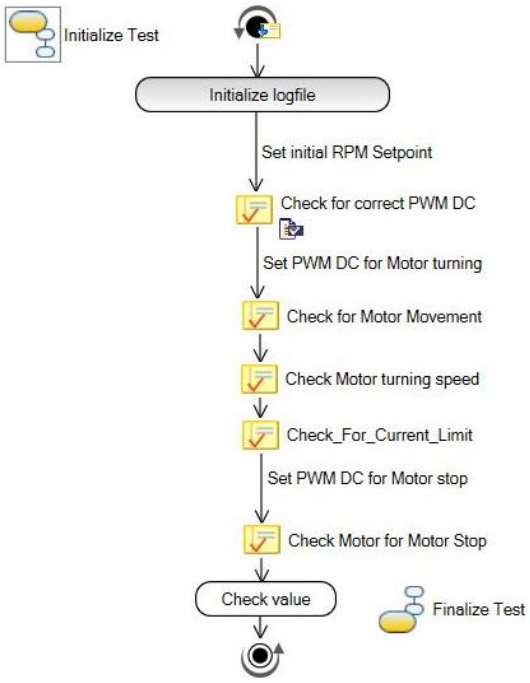
- Minden tesztelendő eszköznél kivétel nélkül szükség van a tápfeszültség vezérlésére, valamint áram- és feszültség mérésére, melyek szinte teljeséggel meg fognak egyezni minden esetben

- Valószínű, hogy szükség lesz valamilyen kommunikációs interfész használatára (CAN, LIN, Flexray,..), vagy valamilyen fizikai anyagáramlás közvetett vagy közvetlen mérésére (folyadékáram, nyomás, fordulatszám, ...), ám ezek már merőben eltérhetnek az egyes esetekben.

Természetesen tudomásul kell venni azt a tényt, hogy nem lehet minden tesztet univerzálisra, és újra felhasználhatóra alakítani, mert mindig lesznek lényegi eltérések, különleges esetek. Mindezek ellenére az egy hasznos cél, ha azt tűzzük ki magunk elé, hogy ezt minél jobban közelítsük, mert hosszú távon mindenképpen megéri. A következőkben be is mutatnám, hogy a használt szoftverek hogyan támogatják ezt a szemléletet.

5.1.1.1 A vTESTstudio eszközei

A program mellett szóló egyik legnagyobb érv, hogy meglehetősen sok lehetőséggel igyekszik támogatni a „Reuse concept” alapján gondolkodó fejlesztőket. Elsősorban ennek a célnak az érdekében történt az, hogy a fejlesztők különválasztották a tesztek modelljét a bennük használt paraméterektől. Ennek a koncepciónak köszönhetően a tesztmodellek függetlenek egy-egy termékre jellemző értékektől, ezáltal más projektek esetén is újra felhasználhatóak, csak az új projekthez szükséges paraméterlistát kell elkészíteni. Erre láthatunk példát a 27. ábra**,** ahol egy „Smoke Test”-et mutatok be, mely általában minden terméknél hasonlóan néz ki, és az eszközök normál működését ellenőrizzük vele.



27. ábra A legegyszerűbb futtatott teszt: Smoke Test

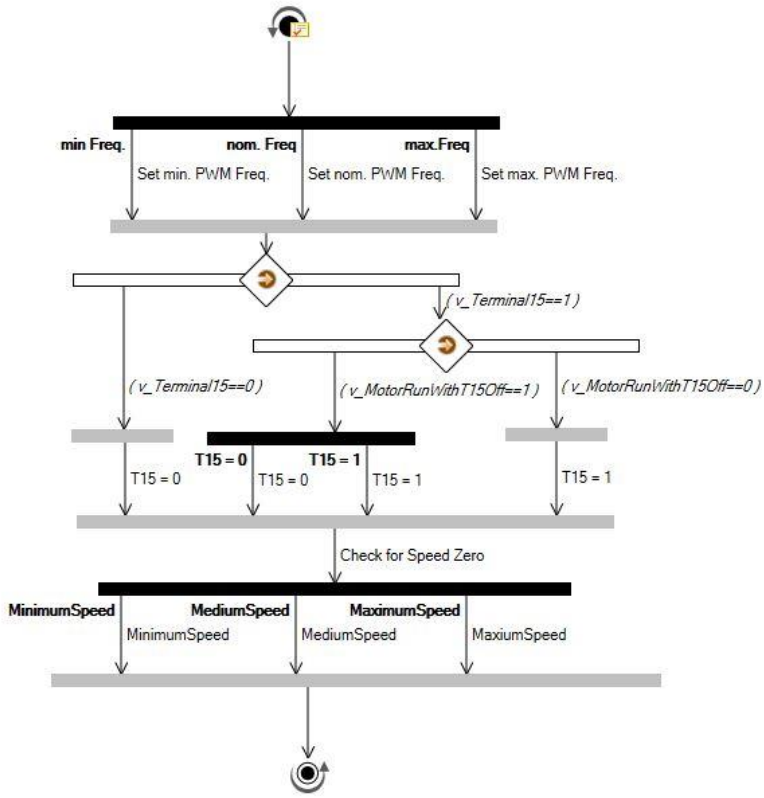
Azon túl, hogy a paramétereket külön forrásból töltjük be, köztük még egy további hierarchia is helyet kap. A programban lehetőségünk van a variánsok kezelésére is, amik az egyes termékek eltérő piacra, vagy felhasználásra szánt verzióit jelenti. Ilyen esetekben az eltérő értékeket külön vihetjük be egy variáns nevéhez rendelve. A 28. ábra láthatjuk két variáns paramétereit, melyek közül a megfelelő verziót futtatáskor egyetlen kattintással kiválaszthatjuk.

	Selector	
Name	EPC_Var1	EPC_Var2
1.10 Voltage_Max_Switch_Off	21	22
1.10 Voltage_Max_Switch_On	18	18
1.10 Voltage_Min_Switch_Off	6	6,2
1.10 Voltage_Min_Switch_On	Voltage_Min_Switch_Off	7,5
±10 PWM_Diag_Available	1	1
±10 vp_PWM_DC_Speed_Min [%]	15	15
±10 vp_PWM_DC_Speed_Max [%]	50	55
±10 vp_PWM_DC_Speed_Med [%]	85	90

28. ábra Variánsok kezelése a paraméterek szintjén

További eszközök a programban, melyek explicit támogatják a kód újrafelhasználást, a Test Sequence, Test Case és Test Function definíciók, melyek

kifejezetten az adott tesztszinten újrafelhasználható kódok készítésére lettek kitalálva. A tesztek struktúrájának kidolgozása során ezek megfelelő használata volt a legfőbb feladat és a legnagyobb kihívás, hogy hatékony és jól felépített architektúrát kapjak. A 29 egy tipikus ECU módokat kombináló, újrafelhasználható szub-szekvenciadiagramot láthatunk.



29. ábra A paraméterek variálása újrafelhasználható subdiagramban

5.1.1.2 A CANoe adta lehetőségek

A CANoe konfigurációs fájlok tartalmazzák a futtatandó teszteket, a kommunikációs beállításokat, valamint a VT-System által gyűjtött jelek feldolgozását, tehát nagyon sok feladat sűrűsödik benne. A Vector Informatik megkönnyíti a munkánkat azzal, hogy OEM-ekhez (Original Equipment Manufacturer) tartozó specifikus konfigurációs csomagokat bocsát a rendelkezésünkre, amivel még könnyebb a CANoe tesztkörnyezet előállítása. Ezt a célt segíti a Canoe-ban a Model Generation Wizard, melyben pár kattintással elkészíthetjük az adott vevő követelményeinek megfelelő összeállítást.

A konfigurációk hatékony újrafelhasználását többek közt a HIL rendszer által gyűjtött jelek okos felhasználásával is elérhetjük. A VT-System csatornáin beolvasott értékeket CANoe-ban a rendszer többi jeléhez hasonlóan egyszerű változókként érjük el. Ezeket a csatornákat szabadon elnevezhetjük a felhasználásuknak megfelelően, ám ettől létezik számunkra optimálisabb megoldás. Az általam készített rendszerben a bejövő változókat közvetlenül nem használom fel, hanem helyettük új, egy-egy felhasználásra specifikus változókat hozok létre, majd összerendelem őket. Ez az úgynevezett mapping.

Ez a módszer egyrészt azért előnyös, mert az így kapott virtuálisan származtatott, és nem hardverbemenetből eredő jeleimet szabadon feldolgozhatom és manipulálhatom (interpolálás, ablakozás, egyéb szűrések). Másik előny az, hogy ennek a felső változórétegnek a használatával hardverfüggetlenné tehetem a konfigurációt, hiszen más hardver esetén, például eltérő mérőkártyáknál, csak a mappelést kell megváltoztatni, cserébe a teljes konfigurációnk hordozhatóvá válik. A virtuális jelek definiálásánál jó kiindulási alapot jelentettek a más termékeknél és teszteknél megismert követelmények, amelyeknél látott hasonlóságok alapján igyekeztem a lehető legbővebb használandó változóhalmazt előállítani. A 30. ábra láthatunk egy példát erre az összerendelésre, aminek részeként lehetőségünk van elsőrendű polinomiális kapcsolatot is meghatározni a két jel közt, erre példa az első sorban a táp esetén látható, ahol az eszköz kalibrálása alatt feljegyzett mérési hibája alapján kompenzáltuk a beolvasott értéket.

	Variable / Signal		Offset	Factor		Variable / Signal	Mapping
☑	IN_V_PowerSupply	⇐	0.04...	1.00...		M1_Ch1::Avg	OnChange
☑	OUT_Dig_SenseSup		0	1	⇒	M2_Ch1::DigitalOutput	OnChange
☑	OUT_Dig_LIN		0	1	⇒	M2_Ch2::DigitalOutput	OnChange
☑	OUT_Dig_LeanCOM		0	1	⇒	M2_Ch3::DigitalOutput	OnChange
☑	IN_RPM_n	⇐	0	60		M2_Ch8::PWMFreq	OnChange
☑	IN_DegC_ECU	⇐	-100	30		M2_Ch12::Avg	OnChange
☑	IN_DegC_Ambiant_Temp	⇐	-100	30		M2_Ch13::Avg	OnChange
☑	IN_A_PowerSupply_Out1	⇐	0	1		M3_Out1::AvgCurrent	OnChange
☑	IN_A_PowerSupply_Out2	⇐	0	1		M3_Out2::AvgCurrent	OnChange

30. ábra Jelek összerendelése CANoe-n belül

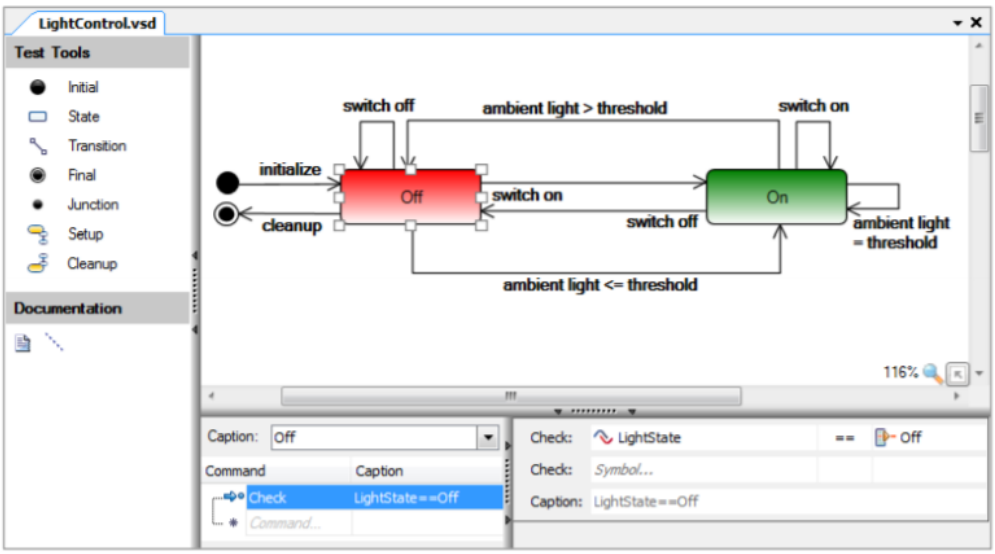
5.2 Testek struktúrája

A tesztek elkészítésére a rendelkezésre álló lehetőségek közül a következő architektúrát dolgoztam ki: a tesztek lépéseinek modelljét grafikus test design formában készítettem el, ami mögött táblás editorban készített test case-ek találhatóak, valamint C#-ban írt funkciók. Néhol szükséges volt a jelek utófeldolgozása, amit CANoe-ban végeztem el CAPL scriptek segítségével, illetve felhasználói felületként szintén CANoe-ban készítettem kijelző panelt.

5.2.1 Grafikus test design

A tesztszekvencia-diagram szerkesztő segítségével lehetőségünk nyílik grafikus formában is megvalósítani a tesztjeinket. Ennek a struktúrának az az előnye, hogy tömör és jól követhető formában képes leírni a teszteket, amin a tesztlefedettség is könnyen ellenőrizhető. Erőssége abban rejlik, hogy grafikus elemekkel elfedi a mögöttes, aprólékos kódrészleteket, és csak a lényegi lépéseket ábrázolja, tulajdonképpen elválasztja egymástól a control flow-t és a tesztek implementációját. Fordítás során a diagram elemeiből (elágazások, ciklusok, stb.) mind különálló tesztesetek generálódnak, így lényegében ez a módszer tényleg csak az implementációt tömöríti, a végkimenetel változatlan a többi módszerhez képest. Ez egyrészt könnyű betanítást és tudásmegosztást tesz lehetővé új csatlakozó kollegák esetében, másrészt a tesztelés technikai részleteiben esetleg kevésbé jártas menedzserek számára is megemészthető formát jelent, ezzel is megkönnyítve a kommunikációt és az egész projektmunkát.

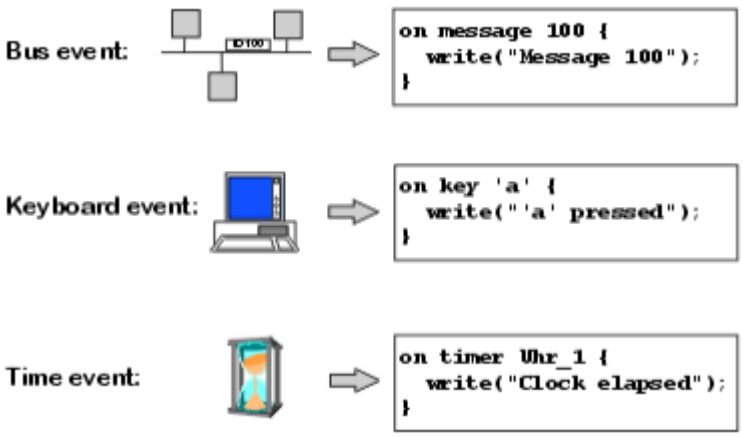
A program legfrissebb verziójában lehetőségünk van a 31. ábra látható állapotgép-szerű leírások elkészítésére is, ami még tömörebb és követhetőbb leírást tesz lehetővé, miközben nagy tesztlefedettséget érhetünk el vele. A fejlesztők azt a tényt használták ki, hogy a tesztek közt nagyon gyakori, hogy különböző állapotokat és a köztük zajló átmeneteket vizsgáljuk, hogy azok vajon megfelelően mennek-e végbe. Ugyan a tesztek elkészítése során még nem állt rendelkezésre ez a szoftverfunkció, de szabad erőforrások esetén a fejlesztési terveink közt szerepel ennek integrálása az újonnan készítendő és már meglévő tesztekbe is.



31. ábra Állapotgép alapú tesztleírás

5.2.2 CAPL

A CAPL egy C-hez hasonló szintaktikájú procedurális programozási nyelv [9], melyet a Vector fejlesztett ki. A programkódokat egy külön erre dedikált szerkesztőablakban fejleszthetjük, futásuk események által vezérelt, mely lehet buszon történő, billentyűzeten bevitt, vagy időzített esemény, ahogy a 32. ábra láthatjuk. A nyelv keretein belülről szabadon elérjük a rendszerben definiált változókat, jeleket, üzeneteket, sőt előre elkészített függvények is támogatják a tesztelést és a szimulációt.



32. ábra A CAPL programokat vezérlő események

A CAPL használata a fejlesztés összes szintjén elérhető, akár a tesztjeinket is írhatnánk ilyen nyelven, de a nyelv alacsony szintje miatt ezt a használatot elvetettem. Az

általam elkészített rendszerben az adatok utófeldolgozásánál kapott szerepet ez a scriptnyelv, például a bejövő adatok szűrésénél.

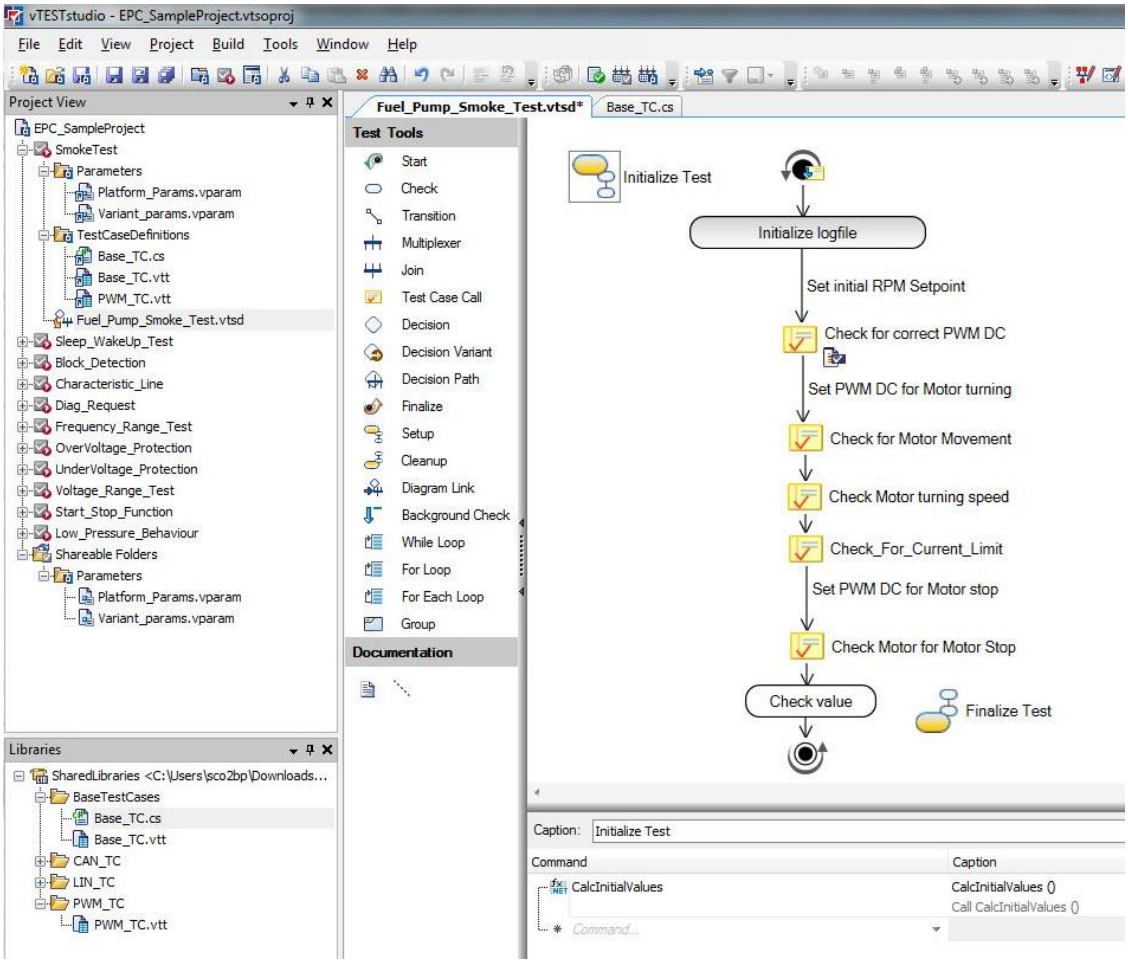
Az alábbi kód két árammérő csatorna által számolt átlagértéket összegezi. Ezt a megoldást alkalmazzuk olyankor, mikor egy mérőcsatorna limitjét meghaladó áramértékeket kell mérnünk. Megfigyelhető a kódban, hogy bármelyik csatorna értékében bekövetkező változás trigger-eseményként szolgál a kódnak, amit a két blokk első sora definiál.

```
on sysvar sysvar::VTS::M6_Out1::AvgCurrent
{
  @sysvar::IO::VTS::IN_A_PowerSupply_2 = @sysvar::VTS::M6_Out1::AvgCurrent
+ @sysvar::VTS::M6_Out2::AvgCurrent;
}

on sysvar sysvar::VTS::M6_Out2::AvgCurrent
{
  @sysvar::IO::VTS::IN_A_PowerSupply_2 = @sysvar::VTS::M6_Out1::AvgCurrent
+ @sysvar::VTS::M6_Out2::AvgCurrent;
}
```

5.3 Elkészült tesztek

A V-modell alapú fejlesztési folyamatnak megfelelően a tesztek fejlesztése a termék fejlesztésével párhuzamosan zajlott és zajlik még most is folyamatosán a fejlesztés előre haladtával. Az aktuális fejlesztési mintafázisnak megfelelő szoftverfunkciókhoz elkészültek a tesztek, melyek listáját a 33. ábra láthatjuk. Néhány, a listában található teszthez még nem készült el az adott követelményeket teljesítő szoftverfunkció, ám a már meglévőket a jelenlegi hardver- és szoftver verzióval kipróbáltam.



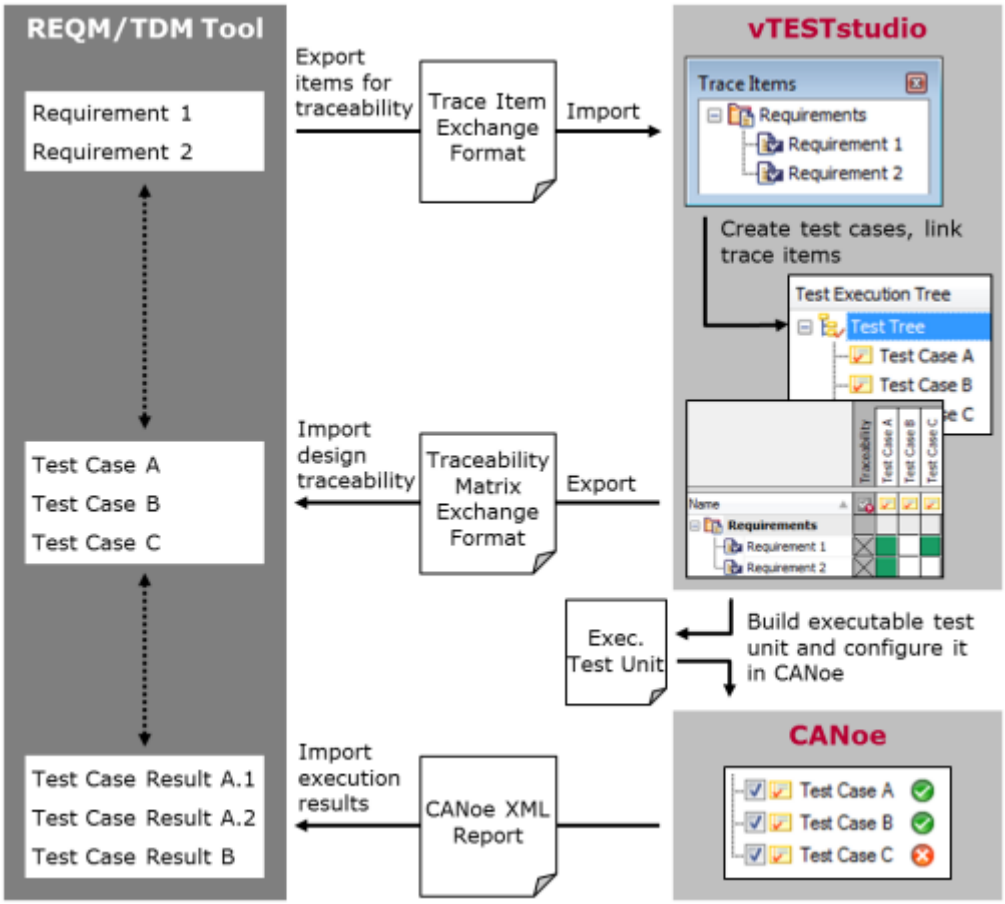
33. ábra Az elkészült tesztek vTESTstudioban

5.4 Alkalmazás életciklus-menedzsment

Az alkalmazás életciklus-menedzsment (Application Lifecycle Management - ALM) olyan előre meghatározott folyamatok összessége, melyek az üzleti ötlet beérkezésétől kezdve végig kísérik az adott alkalmazás fejlesztésének fázisait. Ezek az előírások mind az alkalmazás-fejlesztés átláthatóságát igyekeznek biztosítani, és lehetőséget adnak arra, hogy a projekt résztvevői jobb, rendszerezettebb betekintést kapjanak a fejlesztési folyamatba. Ahogy már korábban az irodalomkutatás résznél is említettem, a fejlesztésünkre vonatkozó életciklus-menedzsment-ben teljesítenünk kell az ASPICE szabványban előírtakat. Az alkalmazás életciklus-menedzsment nagyban megkönnyíti a globálisan elosztott fejlesztést, mert az ALM rendszerek általában egy központi tárolóhelyen fogják össze az egyes elszórt fejlesztő csapatok munkáit, így egy sokkal centralizáltabb projektszervezés valósulhat meg, továbbá a szabott keretek közt történő információáramlás is növeli az együttműködés hatékonyságát. Ennek gyakorlati megvalósítására számos cég kínál különböző megoldásokat, az általunk használt keretrendszer az IBM egyik terméke.

Az említett ALM keretrendszer a „The Rational solution for Collaborative Lifecycle Management” nevet viseli, mely kombinálja az IBM Rational Team Concert (RTC), az IBM Rational DOORS Next Generation (DNG) és az IBM Rational Quality Manager (RQM) különálló szoftvereket [8]. Ezekkel az integrált szoftvereszközökkel a keretrendszer teljes egészében lefedi a fejlesztési folyamatot, köztük a követelmény-, és minőségmenedzsmentet, a verzió- és konfigurációkezelést, valamint projektek tervezési és nyomon követési feladatait.

Számomra elsődlegesen a DNG és RTC szoftverek voltak fontosak, ugyanis rajtuk keresztül kapcsolódnak be a tesztek a folyamatba, a követelmények importálásával és a tesztek eredményének regisztrálásával. A rendszerek közti kapcsolatot a 34. ábra láthatjuk, vTESTstudioba egy plugin segítségével tölthetjük be a követelményeket, míg a kész eredményeket a CANoe XML formátumú riportján keresztül tudjuk felvinni az ALM rendszerbe.



34. ábra Az ALM tool és a Vector szoftverek integrálódása

5.5 Továbbfejlesztési lehetőségek

A fejlesztés jelenlegi fázisának megfelelően elkészültek a 33. ábra látható tesztek, melyek a jövőben érkező újabb szoftverfunkciókkal folyamatosan bővülni fognak. A munka során a tesztek fejlesztése mellett egy másik, a tesztrendszer felé érkező extra követelmény megvalósításával is foglalkoztam, mégpedig a rendszer felhasználásának kibővítése külső eszközök integrálásával, amivel a termék fejlesztőinek munkáját támogathatjuk.

6 A tesztrendszer továbbfejlesztése

Először is megválaszolnám azt a kérdést, hogy miért is merült fel a tesztrendszer továbbfejlesztésének ötlete? Az elkészített összeállítás elsősorban rendszertesztelési feladatokra lett szánva, ezáltal az én munkám is erre a szintre terjedt ki leginkább, ám a termék fejlesztése során kiderült, hogy nagy segítséget nyújthatok a fejlesztőknek tesztelőként azzal, hogy különböző fejlesztéstámogató teszteket futtatok le számukra az egyes fejlesztési mintafázisokban.

Az említett fejlesztéstámogató tesztekre példaként szolgálhat a különböző tápfeszültség tranziensekkel szembeni tűrőképesség vizsgálata, ami nagy segítséget nyújthat például a hardvertervezők számára abban, hogy milyen alkatrészeket válasszanak, vagy hogyan helyezzék el őket a nyákon. Az egyik ilyen célra elkészített teszt azt veszi górcső alá, hogy hogyan viselkedik az eszköz, ha a tápfeszültségére különböző frekvenciájú szinuszos zavarjelek szuperpozícionálódnak. A gond az volt, hogy az ilyen tesztek egy része túlmutatott a VT-System hardveres képességein, méghozzá leggyakrabban azért, mert az eszköz nem volt képes a megfelelő frekvenciájú zavarjelek előállítására vagy mintavételezésére.

A hatékonyság jegyében természetesen nem vetettük el a rendszer használatát, hiszen a példaként felhozott szűk keresztmetszet az adott helyzetben általában egyszerűen orvosolható egy külső függvénygenerátor vagy oszcilloszkóp használatával, míg a problémát okozó jel mérésén kívül minden más mérés és kommunikáció meg van valósítva a rendszerben, amit pazarlás lenne nem felhasználni.

6.1 Kiindulási alapok

Külső eszközök CANoe-ba integrálására a diplomamunkát megelőzően egy megoldásról hallottam, úgyhogy kezdésképpen ennek igyekeztem jobban utána járni. Mivel a külső programkódok hívásának igénye egészen régre visszanyúlik a CANoe fejlesztési életútja során, ezért egy nem éppen mai megoldásról van szó, ami a céget nem meghazudtolóan az általuk fejlesztett CAPL nyelv használatán alapszik.

A használt módszer nem más, mint CAPL DLL-ek (Dynamic Link Library) [9] készítése, amivel külső C/C++ alapú kódokat tudunk ellátni olyan extra kódokkal, melyek segítségével a rendszer azokat CAPL formátumúvá konvertálhatja és alkalmasak lesznek a CANoe-ból való futtatásra. Egy ilyen módon elkészített eszközvezérlést volt

lehetőségem tanulmányozni egy régebbi rendszerben, ahol VISA (Virtual instrument software architecture) architektúrát használva SCPI (Standard Commands for Programmable Instruments) parancsok szöveges küldésével végezték a mérési eszközök vezérlését. Ennek a kódolt megvalósítására láthatunk példát a lenti kódrészletben, ahol egy oszcilloszkóp vezérlésének egy függvényét és az ahhoz tartozó konfigurációs kód egy részét láthatjuk a teljesség igénye nélkül, csupán szemléltetésként.

```
ViSession vi;

void CAPLEXPORT far CAPLPASCAL SIGSetFunction (char *function)
{
      viPrintf(vi, "FUNCtion %s\n", function);
}

CAPL_DLL_INFO CAPL_DLL_INFO_LIST[] = {
#ifdef _MSC_VER
      {  CDLL_VERSION_NAME,  (CAPL_FARCALL)CDLL_VERSION,  CAPL_DLL_CDECL,
0xabcd, CDLL_EXPORT },
#else
      {  CDLL_VERSION_NAME,  (CAPL_FARCALL)CDLL_VERSION,  CAPL_DLL_PASCAL,
0xabcd, CDLL_EXPORT },
#endif ScopeSetTimebase

//    Function   return   param   type   size
  { "SIGSetFunction", (CAPL_FARCALL)SIGSetFunction,  'V', 1, "C", "\001" },
      { 0, 0 }
};
```

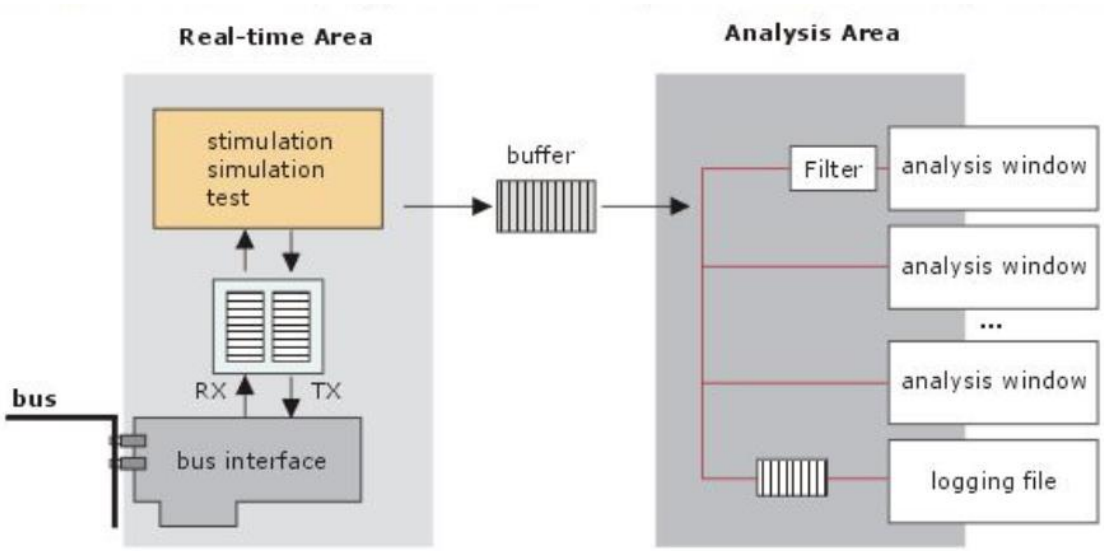
A bemutatott kódrészlet önmagában is beszédes, látszik, hogy az ilyen úton történő integrálás nagyon sok overhead-del járó, alacsonyszintű kódolást igényel, ami meglehetősen nehezen karbantartható.

Az említett rendszer felhasználóival beszélve az is kiderült, hogy az ilyen formájú eszközvezérlés gyakran a tesztrendszer teljesítményét rontja, például egy hosszabb végrehajtású oszcilloszkóp képmentés fagyásokat eredményez a CANoe felhasználói felületén.

A CAPL alapú integrálásnál látott és feljebb fel is sorolt tapasztalataim alapján magasabb szoftver szinten szerettem volna integrálni a műszervezérlést. Ennek megvalósítására a National Instruments LabVIEW [10] szoftverét választottam eszközül, egyrészt saját, már meglévő kompetenciáim alapján, másrészt mert a program kifejezetten jól használható műszerek vezérlésére. A megoldási lehetőségek keresése előtti legfontosabb pontnak azt tartottam, hogy minél mélyebben utánajárjak a CANoe belső működésének, hogy a saját alkalmazásom ne rontsa a rendszer teljesítményét.

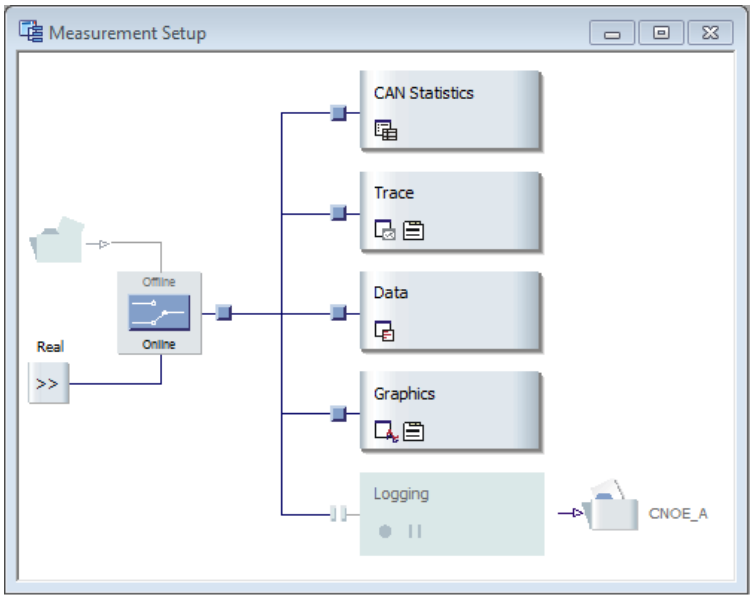
6.2 CANoe mélyebb működése

A CANoe működésének feltérképezéséhez igyekeztem átvizsgálni a rendelkezésre álló dokumentációkat [11][12], valamint volt alkalmam a Vector Informatik egyik munkatársának személyesen feltenni a kérdéseimet a program működését illetően. Az általam készített tesztek a VT-System valós idejű feldolgozó modulját használva a 26. ábra bemutatott módon futnak, a kihívás pedig ennek a konstrukciónak a bővebb megértése volt. A 35. ábra láthatunk egy funkciókat bővebben leíró blokkvázlatot a program kétszintű futásáról.



35. ábra A CANoe kétszálú végrehajtása

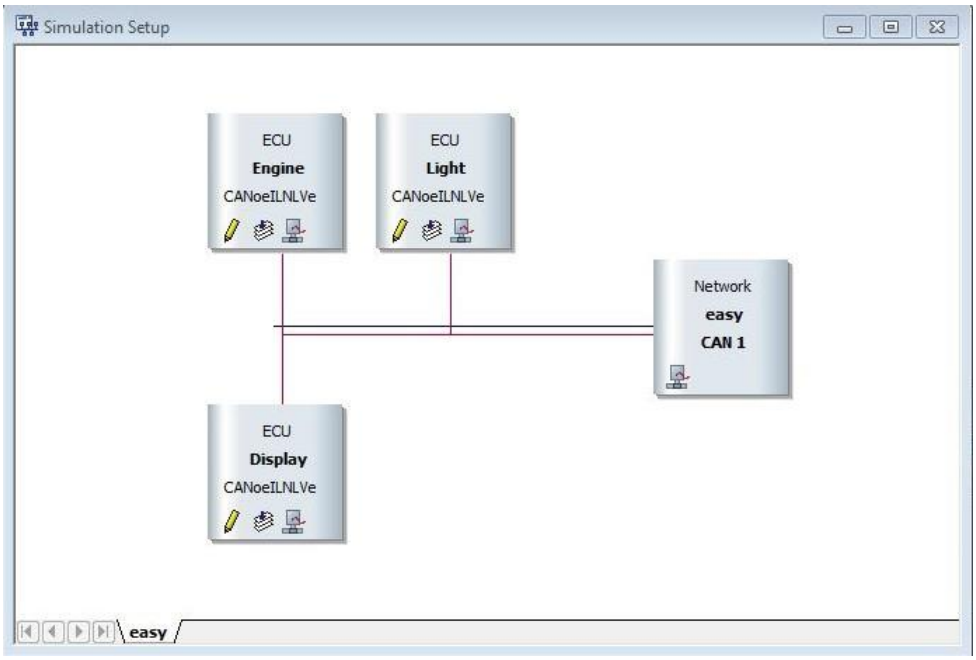
Kezdeném az egyszerűbb, analízist végző résszel, mely az ábra jobb oldalán helyezkedik el. Itt található az összes, nem-időkritikus folyamat, melyek a bemenő adataikat egy bufferen keresztül kapják meg a valós idejű egységtől, és a saját sebességüknek megfelelően dolgozzák fel őket. Ennek a tartománynak a 36. ábra látható Measurement Setup ablak felel meg, így a CANoe használatában kicsit jártasabbak tudják már kötni valahová ezt a területet.



36. ábra Measurement Setup ablak

Az analízist letudva át is térnék a valós idejű részre, mely a 35. ábra bal oldalán foglal helyet, ugyanis itt zajlanak a számunkra fontos események, mert a rendszer összes időkritikus folyamata ezen a blokkon belül kerül végrehajtásra. A különböző, külvilággal kapcsolatot teremtő interfészek a beérkező adatokat időbélyeggel látják el, majd egy kisebb méretű bufferen keresztül egyből feldolgozásra továbbítják azokat. Ugyanazon az ábrán az is látszik, hogy az adatok kiküldésének folyamata szintén itt játszódik le.

A legérdekesebb információt azonban a sárga blokk hordozza, miszerint a valós idejű szekcióban zajlanak a szimulációk, a különböző változók stimulálása és a tesztek ütemezése is. A teljes kép átlátásához hozzátartozik még egy kevésbé ismert, ám annál fontosabb háttérismeret, amit csak személyes kérdezés formájában sikerült kiderítenem, mégpedig az, hogy a CANoe-ban egy szál felel a valós idejű műveletek végrehajtásáért. Ez a gyakorlatban az jelenti, hogy minden egyes kódrészlet, vagy szimulált csomópont, amit a Simulation Setup ablakba helyezünk (37. ábra), az itt fog futni. Ennek a következményeit a CANoe-ban fejlesztőknek figyelembe kell(-ene) venniük, és itt csak kis számítási kapacitású blokkokat elhelyezni, hogy ne befolyásolják vele a rendszer reakcióidejét. Az interfészektől érkező adatok tárolására használt kisebb buffer szolgáltat valamennyi rugalmasságot a feldolgozó egység számára, és így megengedhető egy kis késés a feldolgozásban, de a szál komolyabb leterhelése már adatvesztéssel járhat. Az interfészek által szolgáltatott időbélyegnek tehát fontos szerepe van, mert ezzel biztosítják a konzisztens, valós idejű adatgyűjtést és feldolgozást.



37. ábra Simulation Setup ablak

A program működésének feltérképezése sikeresen zárult, és ezzel együtt a rendszer gyengeségeire is rávilágított, ami egyrészt a CAPL DLL-ek használatánál jelentkező hibát is megmagyarázta és a későbbi fejlesztéshez is hasznos információt jelentett.

6.3 A LabVIEW integrálása

6.3.1 LabVIEW

A LabVIEW egy grafikus programozási nyelvet használó fejlesztői környezet, mely napjainkra széleskörűen használttá vált a mérési adatok gyűjtésére, műszerek vezérlésére és az ipari automatizálás területén is. Népszerűségének oka az lehet, hogy program a könnyű kezelhetősége mellett kiemelkedő szoftveres és hardveres támogatottsággal bír, így műszerek százaihoz tölthetünk le ingyenes drivereket, melyek segítségével egyszerűen elérhetjük a kívánt funkcióikat.

A szoftver egyik legjellemzőbb tulajdonsága a grafikus-, vagy röviden G-programozás, melynek köszönhetően élesen kitűnik a többi szokásos fejlesztőkörnyezet közül. Ez olyannyira jellegzetes, hogy gyakran hívják ezt LabVIEW programozásnak is. A G-programozás során az egyes funkciók és műveletek egy-egy VI-ban vannak megvalósítva, ezek az úgynevezett „virtuális műszerek”, melyek valójában különböző feladatokat megvalósító csomópontok, amiket egymás után sorba kapcsolva hozzuk létre a programjainkat. A G nyelv alapja az adatfolyam jellegű lefutás, azaz az egyes VI-ok akkor kezdik el az adatok feldolgozását, amint azok már rendelkezésre állnak a bemeneten, szemben a közismert sorról sorra történő szekvenciális programlefutással.

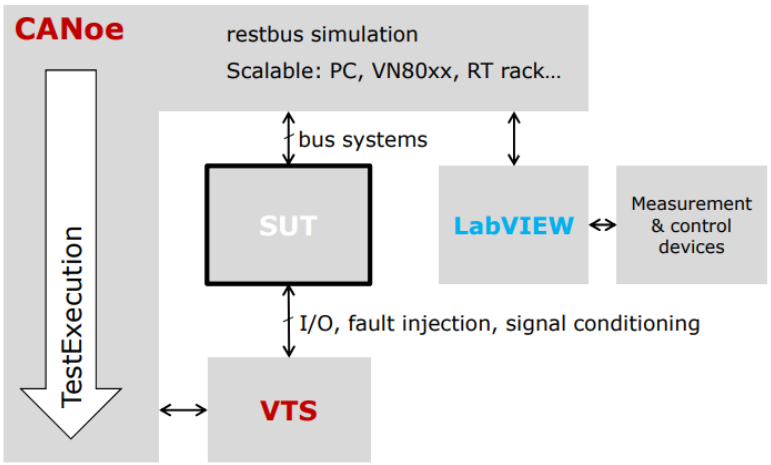
Az átlagostól eltérő megvalósítás ellenére a LabVIEW-n belül is ugyanúgy használhatunk konstansokat, lokális és globális változókat, különböző struktúrákat vagy ciklusokat, mint más programozási nyelvekben, illetve van lehetőségünk más forrású programrészletek importálására is DLL formátumban.

6.3.2 Általános motiváció

Miközben felmértem a LabVIEW és CANoe tesztrendszer összekapcsolásának lehetőségeit, világossá vált számomra, hogy a Vectoros rendszerek LabVIEW-val történő integrálhatósága nem csak egy általam elképzelt irányvonal, hanem már egy jó ideje fennálló, széleskörű igény a felhasználók részéről, aminek irányába már lépéseket is tettek a szoftver fejlesztői.

Ha csak nagy vonalakban áttekintjük a két szoftver erősségeit, könnyen látszik, hogy nagyon jól kiegészíthetik egymást. A CANoe nagyon robosztus, és számos feladatot lát el, de legnagyobb erőssége mégis az, hogy szinte minden elterjedt autóipari kommunikációs buszrendszert és protokollt támogat. A LabVIEW esetében

hasonlóképpen igaz ez a legtöbb méréseknél használt műszerre, függetlenül a gyártóktól vagy az éppen használt kommunikációs interfésztől. Ennek a két erősségnek a kihasználására láthatunk példát a 38. ábra, ahol a Vector a két szoftver kombinált felhasználásának egy lehetséges esetét mutatja be [13].



38. ábra A CANoe és LabVIEW együttes felhasználása

A két rendszer együttes használatának másik hozománya, hogy lehetőségünk adódhat arra, hogy kiküszöböljük a CANoe valós idejű feldolgozásának szűk keresztmetszetét. Ez abból következik, hogy azok a programok, amiket LabVIEW alatt futtatunk, mind a LabVIEW runtime environment alatt futnak és a Host gép erőforrásait használják, míg a CANoe szála zavartalanul futhat ettől teljesen függetlenül a célhardveren.

6.3.3 Megoldási lehetőségek vizsgálata

Először is pontosan meghatároznám a kitűzött feladatot: azt szerettem volna elérni, hogy a Vectoros rendszer és a LabVIEW közt kétirányú adatkapcsolat jöjjön létre, valamint hogy a LabVIEW-ban készített modulok „kívülről” irányíthatók legyenek a Vectoros rendszerből.

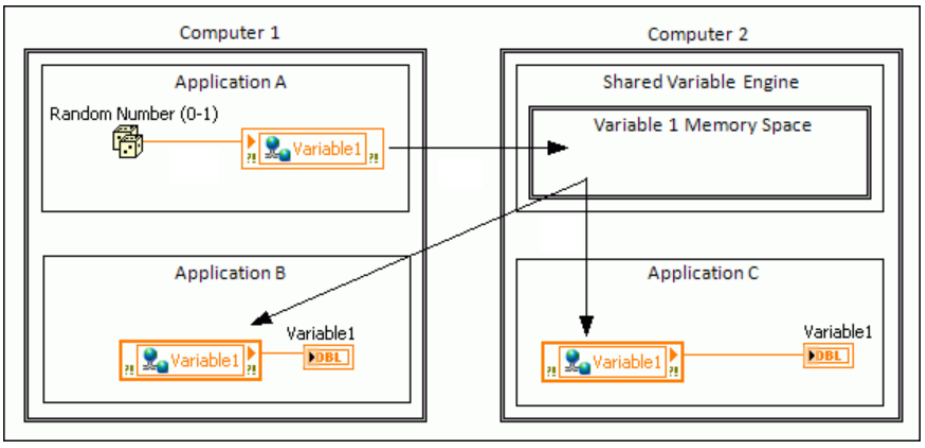
A lehetőségek feltérképezése után négy lehetséges megoldást vizsgáltam meg részletesebben:

- Megosztott változók használata
- FDX alapú kapcsolat
- Standard kommunikáción keresztüli kapcsolat
- ActiveX alapú integráció

Az első kettő megoldás ezek közül a Vector által rendelkezésre bocsátott, CANoe-ba beépülő modulok használatára épül, melyek kifejezetten ilyen célból készültek, az egyik megosztott változók használatán alapszik, a másik pedig egy FDX nevű protokollon. A harmadik megoldás valamilyen szabványos kommunikációs buszon keresztüli kapcsolatra épül, mindez virtuális buszon keresztül, míg a negyedik a LabVIEW ActiveX keretrendszeren keresztüli használatára. A következőkben ezeket ismertetném bővebben, végül a tanulságok levonása után az általam megvalósított kapcsolatot.

6.3.3.1 Megosztott változók használata

A megosztott változók használatának alapja a Shared Variable Engine (SVE) [14], ami a LabVIEW telepítésekor feltelepülő szolgáltatás arra a célra, hogy hálózaton keresztüli adatmegosztást végezhessünk a különböző alkalmazások között. A service ezt az adatmegosztást egy közös szerverként működő memória területen keresztül biztosítja az NI Publish-Subscribe Protocol (NI-PSP) használatával. Ez azt jelenti, hogy az alkalmazások nem képesek a változók közvetlen olvasására vagy írására, kizárólag a szerver publish vagy subscribe függvényeivel érhetik el a kívánt adatműveletet. Ezzel a szabályozott eléréssel biztosítják az egyes alkalmazások zavartalan adatkapcsolatát, akár különböző számítógépekről is, amire a 39. ábra láthatunk példát.

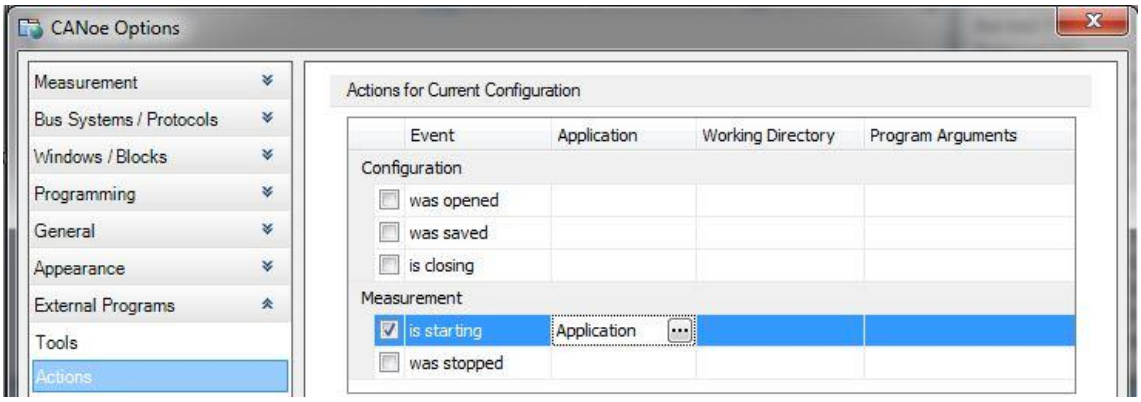


39. ábra A megosztott változók működése

CANoe-n belül lehetőségünk van az ilyen változók importálására, amiket azután interfészként belső változókhoz rendelhetünk és szabadon felhasználhatunk. Ennek a módszernek a használata teljes mértékben kielégítené a kétirányú adatkapcsolattal

szemben támasztott követelményeket, ám nem nyújt megoldást a kívülről vezérelhetőség problémájára.

A kiegészítő alkalmazások indítására egy némileg megfelelő lehetséges megoldást találtam CANoe-ban. A program beállításai közt van egy funkció, amivel lehetőségünk van a mérés vagy konfiguráció bizonyos eseményeihez, például az indulásához egy másik futtatható alkalmazás meghívását hozzárendelni, ezt mutatja be a 40. ábra.



40. ábra Külső alkalmazások meghívásának lehetőségei CANoe-ban

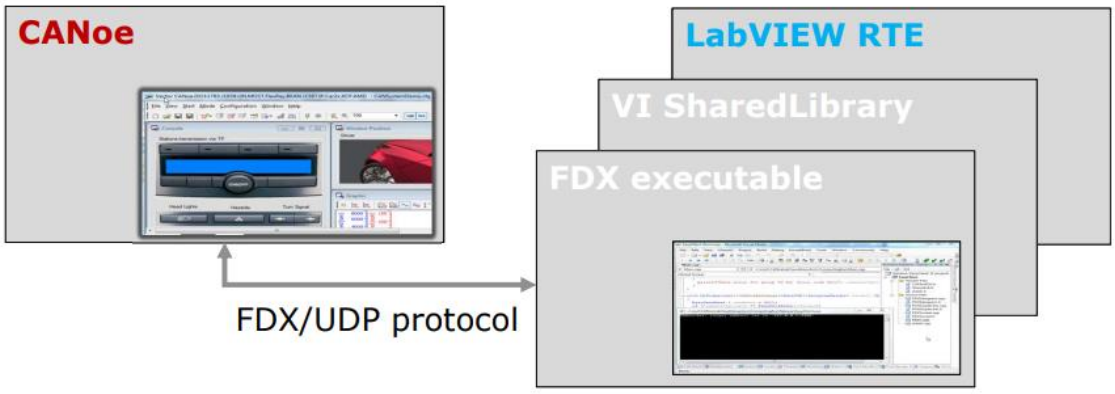
Ez a megoldás sajnos elég korlátozott lehetőségekkel bír, mert lényegében csak a mérés indításával tudnánk meghívni a külső modulunkat, és akkor is csak egy futtatható állomány hívását teszi lehetővé. Ez a két tényező bár nem zárja ki megvalósíthatóságot, de nagy kényelmetlenséget okozna, mert az összes használandó modulunkat egy alkalmazásba kéne integrálnunk, és annak is csak az elindítását tudnánk megoldani automatikusan. Ez azt is magával vonná, hogy minden almodulokon belüli, és modulok közti vezérlést, amit befolyásolni szeretnénk, LabVIEW-ban kellene pluszban implementálnunk, méghozzá megosztott változók használatára építve, hogy integrálhassuk CANoe-ba. Ennek implementálására megfelelő lehet egy megosztott változón keresztül vezérelt állapotgép elkészítése.

6.3.3.2 FDX alapú kapcsolat

Az FDX alapú adatkapcsolat az UDP protokollon alapszik, mely az internet egyik alapprotokollja, kifejezetten rövid, gyors üzenetek küldésére szolgál. Az FDX használatának lehetősége a CANoe 8.0-ás verziójával vált elérhetővé, ugyanis a szoftver fejlesztői ezt a megoldást választották arra a célra, hogy más alkalmazásokkal gyors adatkapcsolatot létesíthessenek szabványosított módon.

A protokoll működésének alapja egy XML formátumú leíró fájl, amiben az adott program változóit tudjuk összegyűjteni és azokat üzenetekbe, más néven datagramokba, csomagolni. Ezeket a datagramokat a program UDP-n keresztül adott port-címre küldi vagy kéri le értéküket ciklikusan. Az ilyen jellegű adatkapcsolat CANoe oldali leírására rendelkezésre áll egy külön szerkesztő program.

Az általam elképzelt rendszer esetében azonban nem a CANoe oldaláról kell FDX interfészt készítenünk, hanem a LabVIEW-t kellene ennek a segítségével integrálni. A Vector Informatik egy régebbi konferencián utalást tett arra, hogy forgalmaznak egy FDX Generator nevű toolt, mellyel a LabVIEW-ban készített shared library-nket tudjuk FDX interfészt használó, futtatható fájllá konvertálni, ami a LabVIEW Runtime Engine alatt működne [13]. Ezt a megoldást ott a 41. ábra látható módon szemléltették.



41. ábra A CANoe és VI-okból készített FDX alkalmazás kapcsolata

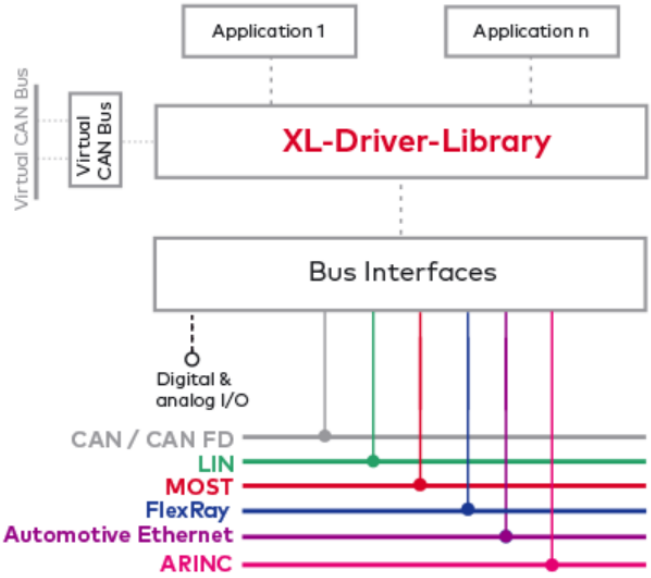
Ezzel a megoldási úttal a megosztott változók használatához hasonló eredményt érhetünk el. A megosztott változós módszerhez képest a CANoe oldali integráció némileg bonyolultabb az FDX miatt, de LabVIEW oldaláról semmit nem kell változtatni a modulokon, nem kell beépíteni a megosztott változókat a programokba. Ennek viszont az az ára, hogy szükség van a LabVIEW-s tool megvételére, ami extra költséget jelent. A vezérelhetőség kérdését tekintve ugyanaz érvényes itt is, mint a megosztott változók esetében.

6.3.3.3 Standard kommunikáción keresztüli kapcsolat

A következő megoldási lehetőségnél másféle megközelítést alkalmaztam, de a kiindulás még mindig az volt, hogy a Vector milyen módon támogatja a különböző harmadik fél által készített alkalmazásokat. A kiinduló ötlet, hogy az adatkapcsolat

megvalósítására használjuk fel valamelyik CANoe-ban könnyen elérhető kommunikációs csatornát (pl.: CAN, LIN), és virtuális buszon keresztül küldjük az adatokat.

Az elmélet technikai megvalósíthatóságát a cég által rendelkezésre bocsátott XL-Driver könyvtár [15] biztosítaná. A XL-Driver egy olyan DLL formátumú könyvtár, mely univerzális programozói interfészt nyújt a felhasználók számára az összes Vector interfész használatához más, külső alkalmazásokban is, valamint a virtuális buszok kezelését is lehetővé teszi, ezt foglalja össze a 42. ábra. A könyvtár felhasználásával adott a lehetőség, hogy LabVIEW oldalon is implementálhassuk az adatok fogadó oldalát.



42. ábra Az XL Driver elhelyezkedése a kommunikációs interfészek felett

Ezt a fajta megoldást magától értetődően viszonylag hamar kizártam a lehetséges opciók közül, mert a többihez képest nagyon sok plusz munkát kívánt volna a megvalósítása. Mindezek ellenére azért szerettem volna mégis megemlíteni, mert a többihez képest teljesen más, out-of-the-box gondolkodás útján jutottam el hozzá, és ez ékes példája annak, hogy kevésbé egyértelmű utakon is találhatunk megoldást egy-egy problémára, még ha nem is ez jut először eszünkbe.

6.3.3.4 ActiveX alapú integráció

A negyedik megoldási lehetőségnél az előzőektől eltérve a LabVIEW oldaláról próbáltam alkalmas interfészt keresni, amin keresztül integrálni tudnám a tesztrendszerbe. A lehetőségek kutatása folyamán megismertem a LabVIEW VI Server architektúrát és az ActiveX protokollt, ezért röviden összefoglalnám a róluk tanultakat.

VI Server

A VI Server nem más, mint olyan funkciók összessége, melyek használatával vezérelhető a LabVIEW Run-Time és a fejlesztőkörnyezet is, lényegben az összes LabVIEW környezet nyújtotta lehetőség. Ez azt jelenti például, hogy távolról programból manipulálhatunk frontpanel elemeket, futtathatunk VI-okat, de még VI forráskódot is generálhatunk vele, ez utóbbit nevezik VI Scriptingnek. A VI Server egy teljes mértékben objektum-orientált és platformfüggetlen megoldás, mely rengeteg lehetőséget ad a hozzáértő fejlesztők kezébe. A szerver kétféle interfészen keresztül érhető el a külvilág felé, ezek a TCP/IP és az ActiveX, melyek közül én az ActiveX alapú megoldással foglalkoztam. A VI Server ActiveX alapokon megvalósított verzióját ActiveX Automation Server névvel illették a fejlesztői, ezért én is így fogok rá hivatkozni.

ActiveX

Az ActiveX a Microsoft alkalmazások közti kommunikáció támogatására létrehozott szabványa. A szabvány szabályozza az alkalmazások közti adatok és utasítások módját és formátumát, ezáltal egységes kapcsolatot teremt a programok közt, akár egy gépen belül, akár ha hálózaton keresztül kommunikálnak. A működési elv alapja az, hogy az ActiveX-et használó alkalmazások egy sor szabadon elérhető objektumot, parancsot és függvényt bocsátanak rendelkezésre, amit a többi ActiveX használatára képes alkalmazás szabályozott formában képes felhasználni.

A szabvány a kommunikációban résztvevő alkalmazások kapcsolatát is definiálja, két alkalmazás a kommunikáció alatt az egyik szerverként míg a másik kliensként fog viselkedni. A LabVIEW mindkét szerep betöltésére képes, ami nem minden program esetén mondható el. Ahhoz viszont, hogy szerverként használhassuk a benne készített alkalmazásunkat, a fájl build-je előtt a beállításoknál engedélyeznünk kell az ActiveX Server használatát.

A begyűjtött információk alapján az ActiveX Automation Server mind adatkapcsolat, mind vezérlés szempontjából kielégítette az elvárásaimat, ezért első választásként ennek a megoldási lehetőségnek a megvalósításával kezdtem el foglalkozni.

6.4 Választott megoldás

A végső voksomat tehát az ActiveX használata mellett tettem le, ezért az első dolgom az volt, hogy felderítsem, milyen szoftveres megoldással érhetem el az ActiveX Automation Server [16] által nyújtott szolgáltatásokat.

Hosszasabb keresésem eredménye az lett, hogy találtam a National Instruments által rendelkezésre bocsátott C# nyelvű DLL-t, mely tulajdonképpen egy C# alapú implementáció az ActiveX interfészhez, szakmai terminológiával élve wrapper. A DLL nem rendelkezett megfelelő dokumentációval, de némi tanulmányozás és kísérletezés után sikerült létrehozni Visual Studion belül egy működő tesztprogramot a segítségével, így a koncepció kipróbálása után már csak az integráció maradt hátra. A Vector-os rendszerben a vTESTstudio C# szerkesztőjét választottam erre a célra, melyben ugyan nem áll rendelkezésre Solution nézet, References, vagy egyéb fejlesztést támogató tool, mint egy normális fejlesztőkörnyezetben, de alkalmas C# alapú kódok egyszerű kezelésére.

A vTESTstudio rendszere tartalmaz néhány korlátozást, például csak .NET 4.0 verzióját támogatja, illetve nem minden formátumú DLL importálása lehetséges, ám szerencsére a szükséges C# DLL-t sikerült használni. A programban továbbá csak olyan függvényeket használhatunk, melyeknek nincs visszatérési értéke, ezért az adatforgalmat CANoe-ban létrehozott rendszerváltozók használatán keresztül bonyolítottam le. Ennek bővebb megértését a függelékben csatolt C# példafüggvény forráskódja segítheti.

A példakód esetén is látott normál eszközkezelés esetén két értéket használok fel, melyeket szinte minden más esetben is. Ezek a kommunikációs csatorna referenciája, amelyen keresztül az adott eszközzel beszélünk, illetve egy error flag, a hibák jelzésére.

A hibajelzésre LabVIEW-n belül egy összetett típust alkalmaznak, melyben található hibakód, hibaüzenet és error flag is, ezek közül a csak a flaget használom fel, méghozzá közvetlenül visszaadva a hívó programnak. Annak érdekében, hogy ellenőrizhessem, hogy a kívánt eszközvezérlés hiba nélkül valósult meg, egy függvénydefiníciós lépést készítettem, mely ellenőrzi az említett flag értékét, ezt láthatjuk a 43. ábra. Az ilyenfajta hibakezelés fontos a külső eszközök használata esetén, mert ellentétben a VT-s erőforrásokkal, ezeknél szükség van inicializálásra és konfigurálásra a teszt elején, illetve a kommunikációs csatorna lezárására a használat végeztével. A

csatornák referenciájának tárolását szöveges alapon végzem, összetett típusok esetén szerializációt, azaz szabványos szöveggé konvertálást alkalmazva LabVIEW-ban.

43. ábra Hibakezelést tartalmazó függvénydefiníció

Az alkalmazott megoldást sikeres tesztelések után magabiztosan építhettem be a szoftveres környezetbe, és segítségével megvalósíthatóvá vált az automatizált fejlesztéstámogató tesztek futtatása a tesztrendszeren.

A működő megoldás alapján már viszonylag könnyen bővíthető a rendszer új eszközökkel. Az integrálandó szoftvermodulok esetén arra a továbbiakban is figyelni kell, hogy nem hívhatunk hosszantartó, blokkoló programot. Ökölszabály jelleggel a tesztrendszerből csak elindítjuk őket, illetve rövid hívásokon keresztül konfigurálunk, de a futtatásuk Windows alatt történik, ezzel tehermentesíthető a CANoe szálja.

A megoldás extra hozománya, hogy nem csak CANoe-n belül alkalmazható, hanem bármilyen más .NET-et támogató alkalmazásban, így sikerült egy működő, univerzális interfészt megvalósítani a LabVIEW-s alkalmazásaink számára.

7 Összegzés és értékelés

A diplomatervezés során mély betekintést nyerhettem egy valós, ipari környezetben zajló fejlesztési folyamat működésébe, amelybe sikerrel kapcsolódtam be a tesztelés oldaláról.

A munka eredményeként az elméleti háttér tanulmányozása után a tesztelendő eszközt megismerve összeállítottam egy hardver-szoftver együttesből álló tesztrendszert. Az elkészített rendszeren a követelményeknek megfelelően megalkottam a kívánt teszteket és a tesztrendszer szoftveres továbbfejlesztésével alkalmassá tettem fejlesztéstámogató tesztek futtatására.

A diplomatervezés alatt számos területen szereztem új ismereteket mind elméleti, mind gyakorlati oldalról. Ezek közt kiemelten a szoftverek használatában sikerült olyan mélységű ismeretekre szert tennem, melyek véleményem szerint jó alapot fognak biztosítani a szakmai fejlődésemhez a tesztelés terültén.

7.1 Kitekintés

Az elvégzett munkában két főbb fejlesztési lehetőséget emelnék ki. Ezek közül az első a vTESTstudioban készített tesztek struktúrája, melyekbe mindenképpen szeretném integrálni a legfrissebb szoftververzió által nyújtott állapotgép alapú leírást, amely még tömörebb tesztdesign-t eredményezne. Amennyiben adódik rá lehetőség a jövőben, úgy szeretném a kialakított tesztdesign-struktúrámat más termékek esetén is megvalósítani, hogy a várhatóan felmerülő hibák kiküszöbölésével még univerzálisabb és robusztusabb kialakítást érhessek el.

A másik potenciális fejlesztési lehetőség az ActiveX Automation Server felhasználásának vizsgálata más projektek esetén. Az itt megismert módszerrel sokkal szélesebbre nyílt számomra a LabVIEW felhasználási köre, ezért ezentúl sokkal nyitottabb szemmel próbálok tekinteni az adódó feladatokra a jövőben, ezt a lehetőséget is figyelembe véve.

Köszönetnyilvánítás

Először is szeretném megköszönni a tanszéki konzulensemnek, Krébesz Tamásnak a segítségét, aki az egész mesterképzés alatt tanácsaival és útmutatásával segítette munkámat, hogy az általam elkészített dolgozatok mind formailag, mind tartalmilag az egyetemhez méltó színvonalon készülhessenek el.

Szeretném továbbá megköszönni a munkatársaim segítségét, akikhez bátran fordulhattam kérdéseimmel. Kiemelt köszönet jár főnökeimnek, Erdei Andrásnak és Kaszás Balázsnak, akiknek teljes támogatását tudtam magam mögött az elmúlt egy év során, és minden tőlük telhető lehetőséget igyekeztek biztosítani számomra, hogy a lehető legjobb körülmények közt készíthessem el a diplomamunkámat.

Végül, de nem utolsó sorban szeretném megköszönni édesapámnak és édesanyámnak az egyetemi éveim alatt nyújtott támogatását!

Irodalomjegyzék

[1] ISTQB: *Certified Tester Foundation Level Syllabus*, www.istqb.org (2017. december)

[2] VDA QMC Working Group 13, Automotive SIG: *Automotive SPICE Process Assessment / Reference Model*, v 3.0, released 2015-07-16, http://www.automotivespice.com (2017. december)

[3] Sogeti Deutschland GmbH: *TPI Automotive* , v1.01, http://www.tpiautomotive.com (2017. december)

[4] Vector Informatik: *vTESTstudio-, CANoe-, VT-System manuals* , https://vector.com (2017. december)

[5] Scherer Balázs: *Tesztelési ág* , Rendszertervezés és –integráció (VIMIMA11) előadásanyag (2017/2018 / 1.)

[6] Wikipedia: *ISO/IEC 15504* , https://en.wikipedia.org/wiki/ISO/IEC_15504 (2017. december)

[7] Klaus Hoermann, Markus Muller, Lars Dittmann, Joerg Zimmer: *Automotive SPICE in Practice* , 1st edition, ISBN: 978-1-933952-29-1, Rockynook, 2008

[8] IBM: *DOORS, Rational Collaborative Lifecycle Management* , https://www.ibm.com (2017. december)

[9] Vector Informatik: *Programming With CAPL* , https://vector.com (2017. december)

[10] National Istruments: *LabVIEW* , http://www.ni.com/ (2017. december)

[11] Vector Informatik: *AN-IND-1-011 Using CANoe .NET API* , https://vector.com (2017. december)

[12] Vector Informatik: *AN-IND-1-021 Vector Testing Solution* , https://vector.com (2017. december)

[13] Gunnar Stein, Vector Informatik GmbH: *Use of Vector CANoe's open interfaces to integrate NI LabVIEW* , https://vector.com (2017. december)

[14] National Istruments: *Understanding Shared Variable Technology* , http://www.ni.com/ (2017. december)

[15] Vector Informatik: *XL-Driver-Library* , https://vector.com/vi_xl_driver_library_en.html (2017. december)

[16] National Istruments: *Calling LabVIEW VIs from Other Programming Languages* , http://www.ni.com/tutorial/5719/en/ (2017. december)

Rövidítések jegyzéke

ALM	Application lifecycle management
ASPICE	Automotive SPICE
BLDC	Brushless DC
CAPL	CAN Access Programming Language
CMM	Capability Maturity Model
DLL	Dynamic Link Library
DNG	DOORS Next Generation
ECU	Electronic Control Unit
FDX	Fast Data Exchange
HIL	Hardware-in-the-loop
ISTQB	International Software Testing Qualifications Board
NI-PSP	NI Publish-Subscribe Protocol
PWM	Pulse-Width Modulation
RQM	Rational Quality Manager
RTC	Rational Team Concert
SCPI	Standard Commands for Programmable Instruments
SVE	Shared Variable Engine
TFS	Test Feature Set
TPI	Test Process Improvement
UDP	User Datagram Protocol
VISA	Virtual Instrument Software Architecture
XML	Extensible Markup Language

Ábrajegyzék

1. ábra Fejlesztési folyamatok csoportosítása (Process Map) 15
2. ábra Részfolyamatok leírásának formája 15
3. ábra Részfolyamatok közti kapcsolatok 16
4. ábra Mérnöki fejlesztés a V-modell alapján 17
5. ábra Verifikáció és validáció 18
6. ábra A tesztelés Automotive TPI által meghatározott kulcsterületei 20
7. ábra Az üzemanyagpumpa elhelyezkedése a tankban 21
8. ábra Egy mechatronikai rendszer általános felépítése 22
9. ábra A termékek fejlesztési területekre bontása 22
10. ábra Plug-In fejlesztési koncepció 23
11. ábra Felül dízel-, alul benzinpumpa robbantott ábrája 24
12. ábra A dízelpumpa főbb alkatrészei 25
13. ábra Common Rail üzemanyag-rendszer vázlata 25
14. ábra Az elektromos vezérlő blokkvázlata 27
15. ábra Háromfázisú fejlesztési modell 28
16. ábra Platform fejlesztés modellje 29
17. ábra A megvalósítandó feladatok három csoportja 29
18. ábra Szöveges alapú táblás tesztszerkesztő 31
19. ábra Grafikus szerkesztőfelület 32
20. ábra Beépített hullámforma-szerkesztő 32
21. ábra A CANoe felhasználói felülete 33
22. ábra CANoe és vTESTstudio közti kapcsolat 34
23. ábra VT-System moduláris mérőrendszer 35
24. ábra HIL elrendezés 35
25. ábra A CANoe manuális kezelőfelülete a VT-System moduljaihoz 36
26. ábra A Valós idejű működés koncepciója 36
27. ábra A legegyszerűbb futtatott teszt: Smoke Test 39
28. ábra Variánsok kezelése a paraméterek szintjén 39
29. ábra A paraméterek variálása újrafelhasználható subdiagramban 40
30. ábra Jelek összerendelése CANoe-n belül 41
31. ábra Állapotgép alapú tesztleírás 43
32. ábra A CAPL programokat vezérlő események 43

33. ábra Az elkészült tesztek vTESTstudioban 45
34. ábra Az ALM tool és a Vector szoftverek integrálódása 47
35. ábra A CANoe kétszálú végrehajtása 50
36. ábra Measurement Setup ablak 51
37. ábra Simulation Setup ablak 52
38. ábra A CANoe és LabVIEW együttes felhasználása 54
39. ábra A megosztott változók működése 55
40. ábra Külső alkalmazások meghívásának lehetőségei CANoe-ban 56
41. ábra A CANoe és VI-okból készített FDX alkalmazás kapcsolata 57
42. ábra Az XL Driver elhelyezkedése a kommunikációs interfészek felett 58
43. ábra Hibakezelést tartalmazó függvénydefiníció 61

Függelék

Példakód egy LabVIEW-s eszközkezelő függvény C#-on keresztüli hívására:

```
  [Export]          // Exporting the function for further usage
  [TestFunction("DAQ_Init")]
  public static void daq_init()
  {
   // creating LV app reference
   _Application LabVIEWApp = new LabVIEW.Application();

   // error handling
   if(LabVIEWApp == null)
   {
      LabVIEW_Interface.Error = 1;
      return;
   }
   // path of the targeted VI
   string vipath = @"C:\LV_Interface\DAQ_init.vi";

   // opening VI reference
   VirtualInstrument vi = LabVIEWApp.GetVIReference(vipath, "",false);

   // Error handling
   if(vi == null)
   {
      LabVIEW_Interface.Error = 1;
      return;
   }

   // Referring to LV connectors by names
   object paramNames = new object[]
   {
       "InstrumentSession",
       "Error"
   };

   // Linking Values / variables to the connectors
   object paramValues = new object[]
   {
       LabVIEW_Interface.InstrumentSession,
       LabVIEW_Interface.Error
   };

   // Call of the VI with some window appearance option
vi.Call2(ref paramNames, ref paramValues, true, false, false, true);
  }
```