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

Pethő Zoltán Zsombor

Folytonos SW integrációs környezet fejlesztése

DIPLOMATERV

Konzulensek

Sujbert László
Méréstechnika és Információs Rendszerek Tanszék

Varga Ferenc
ThyssenKrupp Presta Hungary Kft.

Budapest, 2014

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Pethő Zoltán Zsombor**, 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, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulensek 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.

Budapest, 2014. 12. 17.

.....
Pethő Zoltán Zsombor

Köszönetnyilvánítás

Ezúton szeretném megköszönni a ThyssenKrupp Presta munkatársainak, többek között konzulensemnek, Varga Ferencnek, továbbá Szpisják Tamásnak és Galambos Róbertnek az egész éves áldozatos szakmai segítséget.
Szeretném megköszönni tanszéki konzulensemnek, Sujbert Lászlónak a diplomaterv készítés alatt nyújtott szakmai támogatást, és a folyamatos rendelkezésre állást.
Továbbá szeretném megköszönni a családomnak a teljes egyetemi képzésem alatt tanúsított támogatásukat.

Rövidítések

API ..... Application Programming Interface
ARXML ..... AUTOSAR Extensible Markup Language
ASM ..... Assembly
AUTOSAR..... Automotive Open System Architecture
BMW ..... Bayerische Motoren Werke AG
BSD ..... Berkeley System Distribution
BSW ..... Basic Software
CAN ..... Controller Area Network
CI ..... Continuous Integration
CD ..... Continuous Deployment
CDD ..... Component Driven Development
CPU ..... Central Processing Unit
DBI ..... Database Interface
DIO ..... Digital Input/Output
DOS ..... Disc Operating System
EA ..... Electric Accelerator
ECU ..... Electronic Control Unit
EHPAS ..... Electro Hydraulic Power Assisted Steering
EPAS ..... Electric Power Assisted Steering
FDD ..... Feature Driven Development
GB ..... Gigabyte
GCC ..... GNU Compiler Collection
GHS ..... Green Hills Softvare
GPIO ..... General-Purpose Input/Output
GUI ..... Graphical User Interface
HDD ..... Hard Disc Drive
HIL ..... Hardware In the Loop
HPAS ..... Hydraulic Power Assisted Steering
HW ..... Hardware
IANA ..... Internet Assigned Numbers Authority
IC ..... Integrated Circuit

IDE ..... Integrated Development Environment
IP ..... Internet Protocol
MCAL ..... Microcontroller Abstraction Layer
NAT ..... Network Address Translation
OS ..... Operating System
OSI ..... Open System Interconnection
PC ..... Personal Computer
PLL ..... Phase Locked Loop
PNEUPAS ..... Pneumatic Power Assisted Steering
POSIX ..... Portable Operating System Interface for Unix
PPM ..... Perl Packet Manager
PWM ..... Pulse Width Modulation
RAM ..... Random Access Memory
RTE ..... Real-Time Environment
SIL ..... Software In the Loop
SQL ..... Structured Query Language
SVN ..... Subversion
SW ..... Software
SWE ..... Software Edition
TCP/IP ..... Transmission Control Protocol/Internet Protocol
TDD ..... Test Driven Development
TKP ..... ThyssenKrupp Presta
VFB ..... Virtual Function Bus
XML ..... Extensible Markup Language
XP ..... Extreme Programming

Összefoglaló

A ThyssenKrupp Presta csúcstechnológiát képvisel az elektromechanikus kormányrendszerek fejlesztésének világában. Ezeket a kormányrendszereket a mechanikus kapcsolat megtartása mellett elektronikusan vezérelhető rásegítő motorokkal tervezik. Ez egy jövőbe mutató, de mégis hagyományos technológia, hiszen a közvetlen kapcsolat által könnyebben teljesíthetőek a biztonságkritikus kritériumok. Az elektronikus vezérlésnek köszönhetően megvalósíthatóak olyan kormányvezérléshez kapcsolódó extra funkciók, mint az automatikusan beparkoló rendszer, az oldalszél kompenzáció vagy az ütközések elkerülését segítő automatika.

A versenyképesség megtartásához a legfejlettebb technológiákat alkalmazzák mind a HW, mind a SW fejlesztésében. A SW tervezés alapvetően a beágyazott rendszereknél széles körben alkalmazott V-modellt követi. Azonban ezt az eljárást lehet ötvözni az agilis SW fejlesztés bizonyos elemeivel, így a folytonos SW integrációval. Ez legalább napi rendszerességű SW integrációt jelent az alapvető működést vizsgáló automatikus tesztek futtatása mellett. A folytonos integráció előnyei többek között, hogy folyamatosan rendelkezésre áll egy működő, a legújabb funkciókat tartalmazó SW, illetve a fejlesztés és az integrálás is sokkal gördülékenyebb, hiszen csak minimális módosítások történnek az előző működő SW verzióhoz képest.

Az agilis eljárások alkalmazása beágyazott fejlesztés esetén nem triviális, így ebből az aspektusból kell megvizsgálni, hogy az egyes módszerek hogyan növelhetik a folyamat hatékonyságát. A folytonos integráció bevezetésének és alkalmazásának gyakorlati oldalról történő vizsgálata mellett az ehhez szükséges feltételeket is biztosítani kell a jelenlegi fejlesztési folyamatban. Ehhez biztosítanunk kell, hogy a gyakori integráció nem jelent különösebb terhet a fejlesztőknek. Ezért egy könnyen kezelhető, automatizált felület létrehozása a cél, illetve a SW fordítások időtartamát a lehető legjobban le kell csökkenteni.

Erre vizsgálunk meg különböző lehetőségeket körbejárva a SW fordítás lépéseit, amíg a forráskódból futtatható állomány lesz. Emellett egy olyan tool fejlesztését tűztük ki célul, mely mérsékli a beágyazott rendszer jelentette korlátokat az automatikus build folyamat lefutása során. Az eszköz segítségével PC-s környezetben, az ECU-k jeleit szimuláltan előállítva konfigurálható, integrálható és tesztelhető lesz a SW.

Abstract

The ThyssenKrupp Presta is a high-tech company designing electro-mechanical power steering. In a nutshell, an electric motor assists the steering shaft, but the mechanical connection remains as well.

It is a forward-looking technology, but still traditional, with the direct connection satisfying the safety criteria as well. The electronic control can help us developing such extras, like automatic parking system, crosswind compensation or systems, that help us avoiding crashes.

Ensuring the competitiveness, the most advanced technologies are applied to both the HW and SW development. The SW development follows the V-model, which is widely used in embedded system design.

This procedure can be combined with some elements of the agile SW development, such as continuous integration. This means at least one daily integration of the whole SW with the basic validation tests.

The benefit of continuous integration is that we have an up to date running SW version with the latest functionalities. The development and the integration is much lighter as well, because only small modifications are made to the latest working version of the SW.

The usage of agile processes in the case of embedded development is not trivial, thus from this aspect, it should be examined how each method can increase the efficiency of the process. In addition to the examination of the implementation and the application of the continuous integration on the practical side, we have to ensure that the necessary conditions for our development process are met.

For the widely use of this practice, we have to configure the build process so that the daily integration will not be stressful for the developers. We have to create a user-friendly, automatized environment and minimize the build duration as much as possible.

We examine various options to speed up the builds, first by looking deeper into the build steps. In addition, we are planning to develop a tool that can help us reducing the effects that come from the development of an embedded system during the automated build process. With this tool, the SW will be able to be configured, integrated and tested in a desktop environment by the simulation of the ECU signals.

1. A ThyssenKrupp Presta bemutatása

Rövid betekintést teszünk a cég tevékenységébe, illetve az itt alkalmazott fejlesztési módszerekbe.

1.1 ThyssenKrupp Presta

A ThyssenKrupp Presta Hungary egy német székhelyű elektromechanikus kormányrendszerek fejlesztésével foglalkozó cég budapesti fejlesztőközpontja. A magyar központban az elektronikus fejlesztés zajlik, illetve a tesztelés a legalacsonyabbtól a legmagasabb szintig is itt kapott helyet.

Az elektromechanikus kormányrendszerek alapvetően elektronikusan vezérelt rásegítést alkalmaznak a továbbra is mechanikus összeköttetéssel rendelkező kormányművön. Véleményem szerint ez egy összekötő kapocs a jövő, kizárólag elektronikus összeköttetést használó (Steer by wire) és a múlt, mechanikus kormányrásegítései között. Így könnyebben teljesít biztonságkritikus követelményeket, viszont az elektronikus vezérlés segítségével megvalósíthatóak a manapság felső kategóriájú autókba beépített kormányzáshoz kapcsolódó extrák, mint az automatikusan beparkoló rendszer, az oldalszél kompenzáció vagy az ütközések elkerülését segítő rendszer.

1.2 A kormányrendszer bemutatása

Kezdetben a nehéz gépjárműveknél jelentkezett az az igény, hogy a kormány mozgatásához szükséges erő mérséklődjön. Ezt mechanikus áttételekkel egy szint után már nem lehetett tovább orvosolni, tehát valamilyen rásegítést kellett alkalmazni a kormányoszlopon. A szervokormánynak több kritériumnak kellett megfelelnie, többek között biztonságkritikus szempontoknak. Így az egyes meghibásodások esetén a szervo hatás megszűnése után is kormányozhatónak kell maradnia a gépjárműnek, a vezető nem veszítheti el teljes egészében a kontrollt, továbbá megfelelő válaszidővel, és

holtjátékkal kell rendelkeznie. Az alapvető igények mellett fontos, hogy a vezető érezze a kormányt, egyenes haladáskor mérsékelje a rásegítést, kanyarodáskor pedig a kikormányzottság mértékétől függő rásegítést alkalmazzon.

A hagyományos megoldások teljesen mechanikusak, ezek elsősorban hidraulikus (HPAS – Hydraulic Power Assisted Steering) és pneumatikus (PNEUPAS – Pneumatic Power Assisted Steering) rásegítést alkalmaznak. Később megjelentek az elektronikusan vezérelt szervokormányok, ezek három osztálya az elektrohidraulikus (EHPAS), az elektromechanikus (EPAS) és a tisztán elektromos (Steer by wire) rendszerek.

1.2.1. ábra : Kormányoszlopra szerelt elektromechanikus szervorendszer

Forrás : www.autoevolution.com

1.3. Az elektromechanikus szervokormány

Az EPAS rendszereknél tehát az elektromos rásegítés mellett megmarad a mechanikus kapcsolat a kerekek és a kormánykerék között. Ennek is több megvalósítási módja létezik, elsősorban a rásegítést végző villamos motor elhelyezkedése alapján osztályozzuk őket. Ezek típusai [1]:

- A kormányoszlopra szerelt (Column-assist type vagy ColPAS)
- A kormányműre szerelt (Pinion-assist type)

- A fogasléccel egytengelyű motorral hajtott (Direct-drive type vagy TubPAS)
- A fogasléccel párhuzamos tengelyű motorral hajtott (Parallel Power Assistant Steering vagy ParPAS)
- Két nyelestengelyes megoldás

1.3.1. ábra : Párhuzamos kialakítású elektromechanikus szervokormány

Forrás : ThyssenKrupp Presta

1.4. Az elektromechanikus szervokormány működése

A rásegítés vezérlését, mint egy személyautóban a legtöbb elektronikai feladatot, ECU-k, elektronikus vezérlő egységek végzik. Egy mai autóban közel 50 ECU található a legkülönbözőbb funkciók kielégítésére. Ezek egy hálózatot képeznek, a megbízható működéshez információt kell megosztaniuk egymás között. Erre leggyakrabban a CAN buszt, és az idő- és biztonságkritikus követelményeknek jobban megfelelő FlexRay hálózatokat használják.

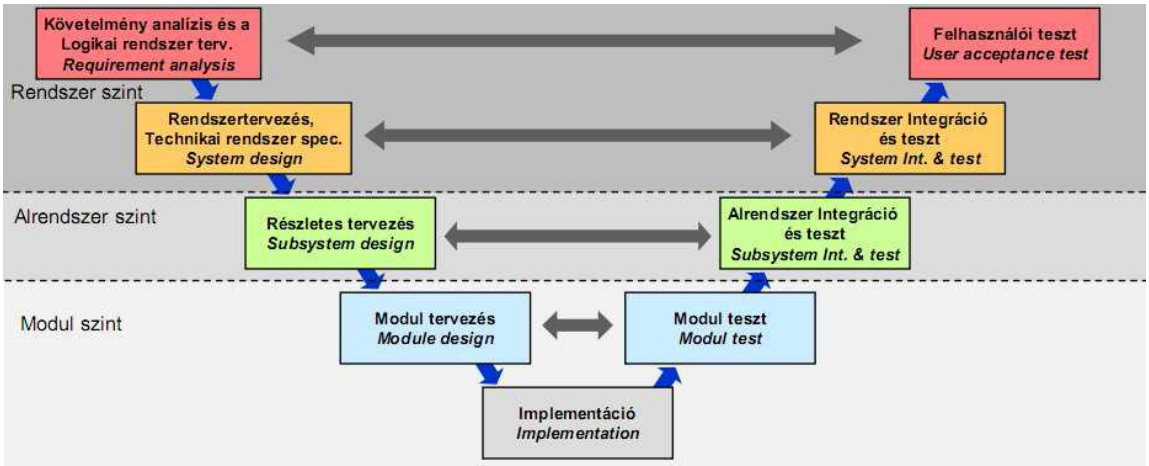
A kormánymű vezérléséhez többek között szükség van a nyomatékszenzor jelére, mely a kormányelfordulási szög adatait szolgáltatja, továbbá az autó sebességére, a motor fordulatszámára, tápfeszültség és a környezeti hőmérséklet értékekre is. A szervo vezérlésénél érdemes a sebesség növekedésével csökkenteni a rásegítés mértékét, melynek akkor is működnie kell, ha az autó úgy gurul, hogy nem jár a motor. Emellett fel kell készíteni a rendszert, hogy szélsőséges körülmények között is kiszámíthatóan működjön, így extrém hideg vagy meleg időben a követelményeknek megfelelően viselkedjen. Ez egy ilyen biztonságkritikus alkalmazásnál elengedhetetlen.

2. *Szoftverfejlesztés*

Röviden megvizsgáljuk, hogy milyen fejlesztési alapelvek használatosak a cégnél, illetve, hogy milyen előnyökkel járna az agilis fejlesztési eljárások használata.

2.1. *Fejlesztési folyamat a ThyssenKrupp Prestánál*

A SW fejlesztés történetében különféle modellek alakultak ki a fejlesztési lépések koordinálására. Az egyik kezdetleges modell a vízesésmodell, mely szekvenciális lépésekben állítja elő a szoftvert. Egy hasonló elveken alapuló, de már a biztonságkritikus beágyazott rendszerek fejlesztése során széles körben elterjedt modell a V-modell. A legfőbb különbség a tesztelési ág visszahajlása, mely által rendszer-hierarchia szintek alakulnak ki, mint a rendszer szint, alrendszer szint, illetve a modul szint. Ezekhez a hierarchia szintekhez specifikálunk a tervezési fázisban, majd az implementáció után, a bizonyos szinteken meghatározott követelményekre tesztelünk. A cég teljes tevékenységét természetesen nem lehet egyetlen ilyen V-modellel leírni, a folyamatokat kisebb absztrakciós szintekre bontva kialakul egy hierarchikus elrendeződés. A rendszer fa struktúrájában minden ághoz rendelhető egy V-modell.



2.1.1. ábra : A V-modell [2]

2.2. *Agilis fejlesztés*

A SW fejlesztésben az alkalmazott modellek mellett különböző fejlesztői metodikák is elterjedtek. Ilyen az agilis SW fejlesztés is, mely iteratív és inkrementális ciklusokon alapszik. Célja a kódminőség javítása, illetve a megrendelő igényeivel való folyamatos egyeztetés. Ezt gyakori release-ek kiadásával, rövid fejlesztési ciklusokkal valósítják meg, mellyel javul a hatékonyság és a checkpointok vizsgálatával új megrendelői követelmények kerülhetnek be a specifikációba. Így nem fordulhat elő, hogy a SW elkészülte után teljesen más funkciókat megvalósító implementáció készül el, illetve a kezdetben bizonytalan megrendelői specifikációk is kikristályosodhatnak a folyamat végére.
Az agilis fejlesztés egyik módszertana az Extreme Programming (XP)[3]. Ez egy több gyakorlati elemből álló ajánlás, mely segítségével könnyebben realizálhatóak az agilis fejlesztés előnyei. Ilyen elem többek között a páros programozás, mely során az egyik fejlesztő implementál, míg a másik figyeli, és minden sort azonnal felülvizsgál (code review). A figyelő foglalkozhat a kódminőség javításával és a jövőben fellépő megoldandó problémákkal, ezáltal a kódolónak elegendő az aktuális feladat implementációjára koncentrálni.

2.3. *Szoftverfejlesztési eljárások*

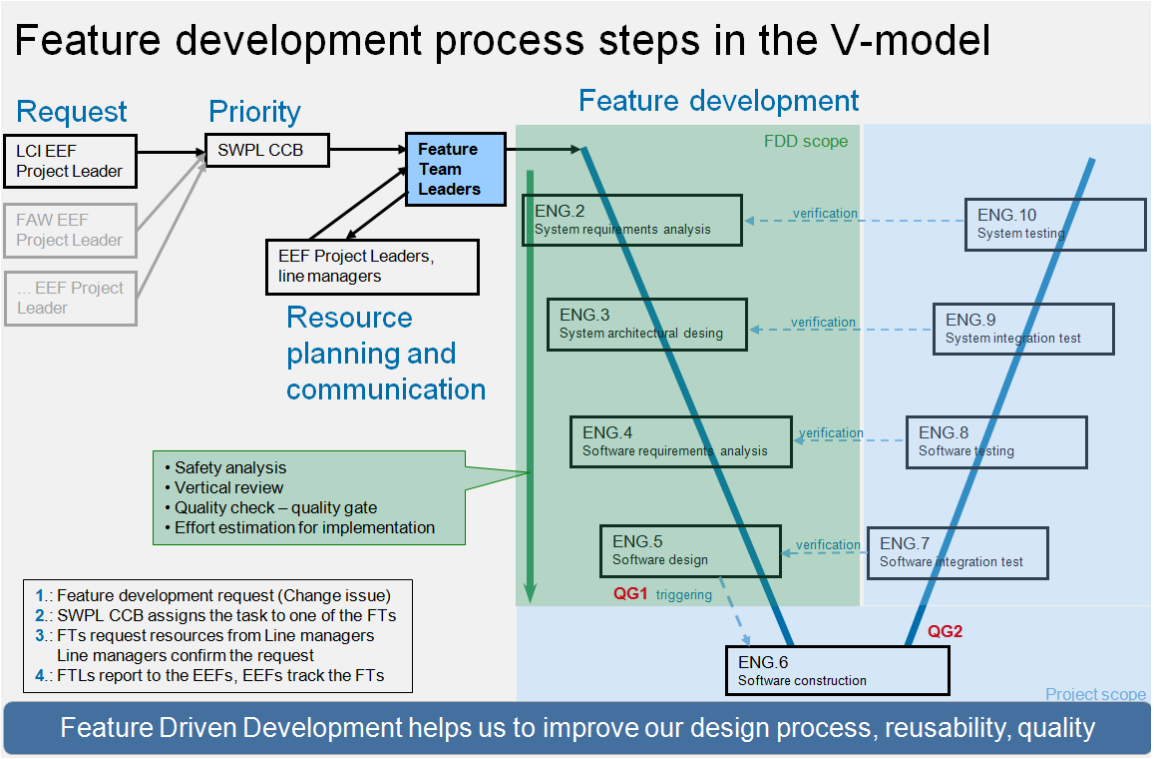
Konkrét alkalmazási területeken általában vegyes fejlesztési eljárások alakulnak ki. Így az alkalmazott módszertanok is keveredhetnek, a V-modell alkalmazása mellett további metódusok is szerephez juthatnak a SW fejlesztés során.

2.3.1. *Feature Driven Development*

Az egyik ilyen eljárás az, hogy minden fejlesztési folyamat egy megrendeléshez kapcsolódik. Ez a szolgáltatás alapú fejlesztés (Feature Driven Development - FDD),

mely szintén a vevő igényeit tartja szem előtt, így az agilis eljárások közé tartozik. Egy új megrendelés esetén, mely lehet egy korábbi verzió új funkcionalitásokkal való kiegészítése, egy csoport (feature team) ül össze, és tisztázzák a követelményeket a megrendelővel. Általában mindegyik, a fejlesztésben részt vevő csoport delegál egy tagot a feature teambe, mely így a teljes fejlesztési folyamat legtöbb aspektusát figyelembe tudja venni. Minden csoport követelménye szem előtt van tartva a fejlesztési folyamat elejétől kezdve. Ezáltal például nem fordulhat az elő, hogy a feature egyfajta implementációjával ugyan hamar végez a fejlesztő csoport, de az nem teljesít egyes előre nem specifikált biztonságkritikus követelményeket, melynek teljesítéséért a safety csoport felelős. Ennek a lépésnek az eredménye egy olyan előzetes specifikáció, mellyel elindítható a V-modell szerinti fejlesztési folyamat, rendszer-, alrendszer-, modul-követelmény analízis, stb.

Az FDD alkalmazásának egyik nagy előnye, hogy kialakulnak az egyes szolgáltatásokhoz tartozó szakértői gárdák. Ez a kód újrahasznosítás és a csapatmunka szempontjából rendkívül fontos. Egy új igény esetén érdemes felülvizsgálni, hogy korábbi projekteken volt-e hasonló alkalmazás, milyen módosítások szükségesek az új megrendelés személyre szabásához. Ez az adott szolgáltatás megvalósításában jártas mérnök számára könnyen átlátható.



2.3.1. ábra : Az FDD és a V-modell együttes alkalmazása a ThyssenKrupp Presta-nál [4]

2.3.2. Component Driven Development

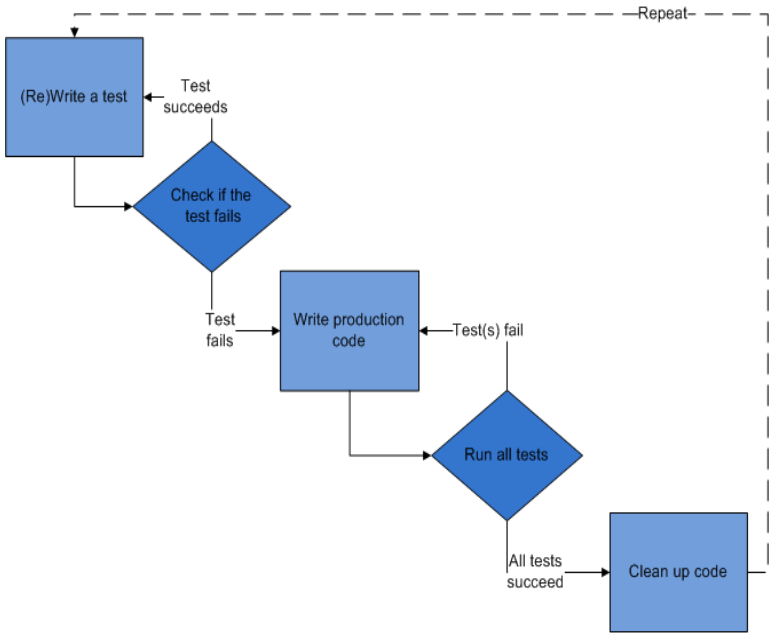
Egy ilyen, moduláris, jól definiált, újrahasznosítható kódrészekből álló fejlesztés a komponens alapú fejlesztés. A komponens egy olyan szoftver egység, amely által megvalósított funkció jól definiált formában, a komponens interfészén keresztül érhető el. A SW fejlesztés irányába hamar kialakult az igény a rövid elkészülési idő, kitűnő minőség, gazdaságosság, a változó igényekhez való gyors alkalmazkodás lehetősége, skálázhatóság, könnyű integrálhatóság, modellekhez való könnyű illeszthetőség felé. A komponens alapú fejlesztés alapgondolata az objektum-orientált paradigmának mintegy továbbfejlesztett változata. A már elkészített és több helyen is szükséges szoftver egységeket ne kelljen újra és újra megírni, azok egy új, esetleg teljesen más applikációba beilleszthetőek legyenek. Különösen igaz az újrafelhasználhatóságra való igény, ha olyan bonyolult, komplex funkciókat valósít meg egy komponens, amiket kifejleszteni meglehetősen időigényes, és ennek megfelelően költséges is. Ezek alapján a komponensek definiálása, elkészítése, tesztelése, továbbfejlesztése végül el is különül az egyes alkalmazásoktól, a komponens, mint funkcionális egység külön életet él. [5]
A komponens alapú fejlesztés hívta életre a generikus SW verziókat, mely egy alapfunkciókat tartalmazó, komponensekkel könnyen testre szabható verziójú program, ami jó kiindulási alap egy új feature fejlesztéséhez.

2.3.3. Test Driven Development

Egy további agilis eljárás a TDD, azaz a teszteken alapuló fejlesztés. Már a metódus elkészítése előtt megírjuk a hozzá tartozó *unit teszteket*, mely külön az alkalmazás adott részegységeit teszteli. Először a fejlesztők megírják a kívánt funkcionalitáshoz tartozó automatizált teszt eseteket, majd minimális szinten implementálják a funkciót, úgy, hogy a unit tesztet kielégítse. Ezután következik a kódszépítés (refactoring), mely során a már letesztelt, működő kódot lehet szépíteni, ami esetleg lassú, rugalmatlan, vagy egyszerűen csak csúnya. A kódszépítés előfeltétele, hogy legyen elegendő unit teszt. A szépítés során nem szabad megváltoztatni a kód funkcionalitását, de a szerkezet, pl. egy metódus törzse, szabadon változtatható. A szépítés után minden unit tesztet újra le kell

futtatni, nem csak a megváltozott kódhoz tartozókat, hogy lássuk, a változások okoztak-e hibát.

Az XP filozófia szerint kevés tesztelés kevés hibát talál, extrém sok tesztelés megtalálja majdnem mind. A tesztelés játssza a dokumentáció szerepét. Nem dokumentáljuk a metódusokat, hanem unit-teszteket fejlesztünk hozzá. Nem készítünk követelmény specifikációt, hanem átvételi teszteseteket fejlesztünk a megértett követelményekből [6].



2.3.2. ábra : a TDD fejlesztési lépései [7]

A TDD fejlesztési lépései [8]

1. A specifikációk alapján **írjuk meg a tesztet** (lehet egy korábbi teszteset módosítása), és adjuk hozzá a meglévő tesztesetekhez. A feature követelményei így hamarabb és mélyebben megérthetőek, és biztosan elkészülnek a szükséges tesztesetek.
2. **Minden tesztet futtassunk**, az új tesztesetünknek a hiányzó implementációnak megfelelően kell elbuknia, ezáltal valószínűleg az adott funkcionalitást sikeresen vizsgálja.
3. **A feature minimális implementálása**, csak annyi és olyan kódot írjon a fejlesztő, ami kielégíti funkcionálisan (még ha nem is túl elegáns módon) a tesztesetet. További, valószínűleg teszteletlen funkciók implementálása nem megengedett ebben a lépésben.

4. **Tesztek futtatása**, sikeres futás esetén biztosak lehetünk az implementált funkció sikerességében, így áttérhetünk a kódszépítés lépésére. Elbukott tesztek esetében visszatérünk az implementációs lépéshez.
5. **Kódszépítés**, tisztítsuk ki a kódot a szükségleteknek megfelelően. Helyezzük az implementációt a logikailag indokolt helyére, tüntessük el az összes lehetséges ismétlődést. Biztosítsuk, hogy a változók, metódusok nevei a funkciójukra utalnak. A tesztek újrafuttatásával biztosíthatjuk, hogy a kódszépítés nem okozott hibát a funkcionális működésben.

A TDD előnyei

A TDD alkalmazásával a kód sokkal modulárisabb lesz, flexibilisebb, tisztább interfészekkel, mert a fejlesztőknek sokkal kisebb egységekben kell gondolkodniuk, melyeket egymástól függetlenül implementálhatnak és tesztelhetnek. Emellett, mivel nem születik olyan kód, mely nem egy funkcionalitást elégít ki, tehát nem egy elbukott tesztet hivatott sikeressé tenni, az automatizált teszteknek minden lehetséges elágazást le kell fedniük a kódbázisban. Pl. nem kerülhet be egy *else* ág egy már meglévő *if* szerkezetbe, ha arra nem született előtte egy sikertelen teszteset. Így a végső automatikus tesztállomány kellőképpen átfogó lesz, nem juthat át rajta olyan változás, mely váratlanul módosítja a funkcionalitást.

2.4. AUTOSAR

Az autóiparban a felhasznált ECU-k számának növekedésével párhuzamosan kialakult az igény egy egységes standard alkalmazására. Egy autógyár számos beszállítóval rendelkezik, külön fejlesztőcégtől szerzik be a belső elektronikát, a váltóvezérlést, vagy éppen a kormányvezérlő elektronikát. Ezáltal több különböző gyártmányú IC kerül a rendszerbe, a beszállítók által támogatott különféle interfészekkel. A kész személygépjárműben azonban már gondtalanul kell ezeknek kommunikálni, mely gyakran bonyolult és felesleges portolási problémákat okozott.
2003-ban több gyártó összefogásával létrejött az AUTOSAR (AUTomotive Open System ARchitecture) szabvány, mely használatával több ECU kommunikálhat és

dolgozhat egy általános interfészen keresztül. A SW komponensek könnyen portolhatók egyik ECU-ról a másikra, ez a szigorúan rétegezett SW architektúrának köszönhető. Egy bizonyos réteg felett teljesen HW független komponensek találhatóak. Moduláris, skálázható, újrahasznosítható kódot eredményez, így támogatja a komponens alapú fejlesztést. Az alapító szervezet jelmondata [9] „*Cooperate on standards, compete on implementation!"* szerint, a fejlesztések során az implementáció hatékonyságára és megbízhatóságára kell fordítani az energiákat, nem az interfészek illesztésére. Ennek köszönhetően az utóbbi években igen elterjedtté vált az AUTOSAR, a 9 alapító tag mellett mára közel 150 prémium tagot számlál a szervezet.
Az AUTOSAR a szabályok meghatározása által a futtatható állományok nagy részét egy modellből generálja. Az egyes módosítások a modellen történnek, majd az ebből generálódott kód is tartalmazza a megvalósított funkciót. Az így legenerált állományokon kézzel már nem történik módosítás. Az AUTOSAR használatával tehát a komponens és a modell alapú fejlesztés is szerepet kap.

2.5. A folytonos integráció

Az agilis fejlesztési eljárások egyik legfontosabb gyakorlati eleme a folytonos integráció (Continuous Integration - CI) [10], mely a komplett metodika bevezetése nélkül is alkalmazható, annak előnyei kihasználhatóak nem agilis SW fejlesztési modell esetén is. Sok beágyazott fejlesztéssel foglalkozó cég alkalmazza hasonló módon. A folytonos integráció gyakorlatilag legalább napi rendszerességű SW integrációt jelent, a fejlesztő csapat által módosított kód így kerül be a korábbi fejlesztések közé. Minden új kód integrálása során automatizált tesztek ellenőrzik, hogy a rendszerbe való illesztés során okozott-e valamilyen hibát az új kódrészlet és ennek eredményeként a lehető leghamarabb visszajelzést ad az integráció eredményéről.

2.5.1. Motivációk a CI bevezetésére

Ha kellően gyors és gördülékeny az integráció, a fejlesztők viszonylag kevés kódmódosítással járó új funkcionalitást integrálnak a rendszerbe, ennek megfelelően

hibás működés esetén könnyebben megállapítható a hiba oka. Emellett az integráció során fellépő hibajelenségek is könnyebben kezelhetőek, így nehezen alakulhat ki olyan szituáció, hogy a projektleadás határidején sikertelen integráció miatt csúszik a projekt. A kódminőség is javulhat a folytonos integráció megfelelő alkalmazása mellett, hiszen ha az új funkcionalitásokat külön modulokba szervezik a fejlesztők, javul a kód áttekinthetősége, nő a modularitás. A folyamatosan a legújabb feature-ökkel rendelkező SW rendelkezésre állása alkalmas további tesztek futtatására, illetve fejlesztés alatti demonstrációs célokra. Ahhoz, hogy egy jól használható folytonosan integrált szoftverünk legyen megfelelően konfigurált automatikus tesztelési folyamatokat kell implementálni. Ez a TKP esetében a korábban elkészült saját fejlesztésű Quick Check Tool alkalmazását jelenti, valamint az alapvető funkcionalitásokat ellenőrző teszt (smoke test, ad-e a rendszer asszisztot). Később a kielégítő validációs vizsgálatokhoz további modul teszteket alkalmazunk.

3. *Folytonos Integráció a gyakorlatban*

Ebben a fejezetben közelebbről megismerkedünk a folytonos integráció hatékony alkalmazásának gyakorlati oldalával, majd megvizsgáljuk, hogy hogyan lehet részévé tenni a mindennapi beágyazott fejlesztési folyamatnak.
Az alábbi fejezet Martin Fowler: Continuous Integration [10] című angol nyelvű cikke alapján állt össze. Fowler egy brit fejlesztőmérnök, akit sokan az agilis eljárások egyik atyjának tekintenek. Ő is részt vett a 'Manifesto for Agile Software Development' [11] összeállításában, mely összefoglalja az agilis fejlesztés 12 alapelvét. Ezt 2001-es kiadása óta közel százezer SW fejlesztő támogatta aláírásával.

Az agilis fejlesztés 12 alapelve az alábbi:

1. Legfontosabbnak azt tartjuk, hogy az ügyfél elégedettségét a működő szoftver mielőbbi és folyamatos szállításával vívjuk ki.
2. Elfogadjuk, hogy a követelmények változhatnak akár a fejlesztés vége felé is. Az agilis eljárások a változásból versenyelőnyt kovácsolnak az ügyfél számára.
3. Szállíts működő szoftvert gyakran, azaz néhány hetenként vagy havonként, lehetőség szerint a gyakoribb szállítást választva.
4. Az üzleti szakértők és a szoftverfejlesztők dolgozzanak együtt minden nap, a projekt teljes időtartamában.
5. Építsd a projektet sikerorientált egyénekre. Biztosítsd számukra a szükséges környezetet és támogatást, és bízz meg bennük, hogy elvégzik a munkát.
6. A leghatásosabb és leghatékonyabb módszer az információ átadásának a fejlesztési csapaton belül, a személyes beszélgetés.
7. A működő szoftver az elsődleges mércéje az előrehaladásnak.
8. Az agilis eljárások a fenntartható fejlesztést pártolják. Fontos, hogy a szponzorok, a fejlesztők és a felhasználók folytonosan képesek legyenek tartani egy állandó ütemet.

9. A műszaki kiválóság és a jó terv folyamatos szem előtt tartása fokozza az agilitást.
10. Elengedhetetlen az egyszerűség, azaz az elvégezetlen munkamennyiség maximalizálásának művészete.
11. A legjobb architektúrák, követelmények és rendszertervek az önszerveződő csapatoktól származnak.
12. A csapat rendszeresen mérlegeli, hogy miképpen lehet emelni a hatékonyságot, és ehhez hangolja és igazítja az működését.

Fowler cikke mellett Jez Humble és David Farley: Continuous Delivery [12] című könyvének a folytonos integrációval foglalkozó fejezeteit használtam fel.

3.1. A folytonos integrációs fejlesztés folyamata

A CI-al való fejlesztést legkönnyebben egy egyszerű feature fejlesztésének lépésein keresztül lehet bemutatni. Tegyük fel, hogy egy kis SW módosítási feladatot kapok, mely néhány óra alatt megvalósítható.
Első lépésben letöltök egy *working copy*t az aktuális *trunk*ról a fejlesztői gépemre a verziókezelő rendszerből. A verziókezelő rendszereknek megvan a maguk szakzsargonja, röviden igyekszem tisztázni a legfontosabb fogalmakat. A verziókezelő rendszer tárolja a projekt összes forrás kódját a *repository*ban (adattár). A rendszer aktuális állapotára a *mainline* vagy *trunk* kifejezések hivatkoznak. A *checking out* művelete során a fejlesztő egy *working copy*t (másolatot) tölt le a saját munkaállomására. Ha a fejlesztés menete új irányba indul el, akkor egy új fejlesztési ág *(branch)* készül. A szükséges módosítások elvégzése után a fejlesztő vissza szeretné tölteni *(commit)* a saját verzióját, melynek módosításait össze kell fésülni *(merge)* a trunkon lévő SW verzióval.
Tehát leszedek mindent a verziókezelőből, ami szükséges a feladatom elvégzéséhez. Ez a kód módosítása mellett automatikus tesztek hozzáadásával vagy módosításával is jár. Miután ezekkel elkészültem, elvégzek egy automatikus SW build folyamatot a saját gépemen. Ez bekéri a forráskódot, lefordítja, linkeli (ezekről a folyamatokról részletesebben olvashatunk az 5. fejezetben), majd lefuttatja az automatikus teszteket. Kizárólag hiba nélküli lefutás esetén tekinthetjük a buildet sikeresnek.

Egy sikeres build után szeretném feltölteni a módosításaimat a repositoryba. Azonban számítanom kell arra, hogy amíg a feladaton dolgoztam, a többi fejlesztő már töltött fel módosításokat a trunkra. Először tehát frissítem *(update)* a gépemen lévő working copyt a trunk aktuális változatára, és újraindítom a build folyamatot. Ha a többiek módosítása ütközik az enyémekkel, az hibát fog okozni a fordítás vagy a tesztelés során. Ez esetben az én feladatom, hogy úgy módosítsam az implementációmat, hogy az sikeresen összeforduljon az aktuális trunk verzióval. Amint ez megtörtént, commitolhatok a trunkra, és a verziókezelő automatikusan frissíti a repositoryt.
Ezzel azonban még nem ért véget a feladat megvalósítása, újrafordítjuk a teljes trunk verziót, ezúttal azonban egy integrátori gépen. Ha itt is sikeresen lefut a build, akkor tekinthetjük befejezettnek a módosítást. Mindig megvan az esélye, hogy a saját munkaállomásom és a repository között nem teljes az átfedés, ezért van szükség egy semleges integrátori buildre.
A két fejlesztő módosításai közti konfliktus általában a második fejlesztő commitja során jön elő. Ha akkor nem, akkor az integrált build során. Így vagy úgy, a hiba hamar kiderül. Ezen a ponton a legfontosabb feladat megoldani, hogy a build folyamat minél hamarabb újra sikeres legyen. Egy jól működő CI környezetben sosem lehet az aktuális integrátori build túl sokáig sikertelen.
Ennek eredményeképp a trunkon egy megbízhatóan működő SW lesz, kevés buggal. Mindenki egy stabil alapra fejleszt, és sosem távolodik el annyira ettől, hogy sokáig tartson beintegrálni a módosításait.

3.2. Gyakorlati tanácsok a CI hatékony alkalmazásához

Az alábbiakban témakörökbe szedve megyünk végig azokon a gyakorlati elemeken, melyek segítenek egy hatékony CI környezet kialakításában.

3.2.1. Verziókövető rendszer

- Manapság minden projektnél alkalmazandó, nincs annyira kicsi projekt, aminél ne lenne érdemes verziókövetést használni (még ha csak a saját gépünkön, a saját igényeinket kielégítő programokat írunk is).

- A verziókezelők nagy előnye, hogy több branch alkalmazásával több szálon, több irányba indulhat el a fejlesztés. Azonban törekedni kell a branchek számának minimalizálására, és minél gyakrabban visszatérni a trunkhoz, egyéb esetben egy-egy feature fejlesztése nagyon eltávolodhat ettől a verziótól, amit utána körülményes lehet visszaintegrálni. Ezáltal értelmét veszti a CI lényege, tehát csak indokolt esetben engedélyezett sok branch használata.
- Gyakori hiba, hogy nem található meg minden a buildhez szükséges file a repositoryban (így a teszt scriptek, adatbázis séma, install scriptek, stb.). Ökölszabály, hogy egy tiszta géppel fellépve a szerverre, checkout után le tudjuk fordítani a projektet. Környezeti alkalmazások lehetnek a kliens gépen – általában nagyméretű, nehezen installálható, stabil szoftverek. Így operációs rendszer, Java IDE, vagy adatbázis kezelők.
- A fordításhoz nem feltétlenül szükséges fájlok is lehetnek a verzió kezelőben, mint az IDE konfigurációs fájljai, melyek hasznosak, ha ugyanazokat az IDE beállításokat szeretnénk használni.
- A fordításhoz minden szükséges fájlt rendelkezésre kell bocsájtanunk a verziókezelőben, de magukat a lefordított fájlokat nem ajánlott, hiszen ezen fájlok jelenléte téves következtetéseket eredményezhet egy korábbi fordítás megismétlésekor.
- Egyes esetekben a fejlesztőcsapatok nem tudnak közös hálózatot elérni (földrajzi vagy egyéb okok miatt). Ekkor lép életbe a megosztott vagy offline verziókezelés. A felhasználók peer-to-peer rendszerben frissítik egymás között a verziókat, ahelyett, hogy egy szerverhez csatlakoznának kliensként (mint a centralizált esetben). Kommunikáció csak a peerek közti változások megosztása alatt zajlik. Gyorsabb a használat, illetve nem egy gépen múlik a rendszer robusztussága. Kezdeti kialakítása ugyanakkor hosszadalmasabb, mert minden branchet, revision historyt kimásol, illetve a lockolás funkció hiányzik, ami körülményessé teheti a verziók összefésülését.

3.2.2. Automatizált fordítás

- Fontos, hogy eliminálju k a különböző hibaforrásokat és felesleges, monoton teendőkkel (mint a parancssorba kézzel írás, párbeszédablakokban különböző helyekre kattintgatás) ne terheljük a mérnököket, ezért ezeket a feladatokat automatikussá

tehetjük a SW fordítása során. Az automatizált fordítás támogatása a legtöbb környezetben adott. (A TKP a GNU Make toolját használja).

- Általános hiba, hogy nem inkludálnak be minden, a fordításhoz szükséges komponenst. Az adatbázis sémát is a repositoryból kell checkoutolni, majd így elindítani a fordító környezetet.
- Egy nagy projekt esetén a teljes fordítás sok ideig tarthat, így egy-egy kisebb módosítás esetén nem biztos, hogy minden lépést újra akarunk csináltatni. Egy jó tool analizálja, hogy a módosítás sikeres fordításához mely komponenseket kell újrafordítani. Alapesetben megvizsgálja a fájlok módosításának dátumát, és csak a később módosított fájlokat fordítja újra. Azonban különböző függőségeket is fel kell térképezni, így ha egy objektum fájl megváltozott, azokat is újra kell fordítani, melyek ettől függtek. Megfelelően szofisztikált függőségvizsgáló fordítóval ez nem jelenthet komolyabb problémát.
- Bizonyos esetekben különbözően kell bekonfigurálnunk a fordítások lefutását, (például ha különböző automatizált teszteket akarunk futtatni), erre is alkalmasnak kell lennie a fordítást vezérlő toolnak.
- A mai CI toolok, (melyek támogatják az automatizált build folyamatot) már kellőképpen szofisztikált felépítésűek ahhoz, hogy egyáltalán ne kelljen parancssor szintre lemenni a fordításhoz vagy a tesztek futtatásához. Ugyanakkor fontos, hogy ne hagyjuk teljesen felügyelet nélkül a build toolt, és folyamatosan fejlesszük a scripteket, melyekkel az IDE használata nélkül, manuálisan is el tudjuk végezni a buildet. Hiba esetén hasznos egy ilyen kódbázis, valamint könnyebben megérthető, debuggolható a build folyamat.

3.2.3. Öntesztelő fordítások

- Egy SW lefordulása nem jelenti, hogy hibátlanul működni is fog, elkerülhetetlen, hogy maradjanak bugok a lefordított szoftverben. Célszerű rögtön fordítás után automatikus tesztekkel ezek számát és hatását minimalizálni. A folytonos integráció is egy XP eljárás, hasonlóan a TDD-hez, azonban nem kell feltétlenül előre megírt unit-testeket alkalmazni az öntesztelő fordítások előnyeinek kihasználásához.

- Egy öntesztelő kódhoz szükség van egy automatizált teszt csomagra, mely a forráskód nagy részéből ki tudja szűrni a bugokat. Amennyiben bármelyik teszt hibát ad, az öntesztelő fordítás sem futhat le sikeresen.

3.2.4. Mindenki a trunkra commitoljon minden nap

- Az egyik előfeltétele annak, hogy egy fejlesztő a trunkra commitolhasson az, hogy a módosított kódja hibátlanul lefordul. Először updateli a working copyt az aktuális trunkról, összefésüli a saját kódjával, feloldja az esetleges konfliktusokat, majd sikeres fordítás után szabadon commitolhat a trunkra.
- Megfelelően rövid ciklusokat alkalmazva a fejlesztők közötti esetleges konfliktusoknak hamar a végére lehet járni. Ha a fejlesztők néhány órás gyakorisággal vissza tudják illeszteni a rendszerbe a módosításaikat, akkor a fellépő hibákat is fel lehet oldani pár óra alatt. Amennyiben a konfliktusok hetekig rejtve maradnak, úgy megoldásuk igen sokáig is eltarthat.
- Mivel a frissített working copyn hajtjuk végre a fordítást, kibuknak mind a fordítási, mind a szintaktikai konfliktusok. A beépített öntesztelések köszönhetően a kód futása alatti hibákat is felfedezhetjük.
- Regressziós hibák esetén érdemes megvizsgálni a korábbi, működő verziókhoz képest történt változásokat. Regressziós hiba, ha egy korábban működő szoftverfunkció hibásan vagy egyáltalán nem működik. Az ilyen hibák tipikusan javítások nem várt következményeként jelentkeznek – vagy a tesztesetekben változott valamilyen feltétel, vagy valóban a tesztelt feature funkcionális hibájáról van szó. Ez esetben a teszt feltételein kell módosítanunk, a funkcionalitást a kódban kijavítanunk vagy elhagyni a tesztet, ha ez a funkció már nem része az elvárt működésnek. Néhány órás ciklusok esetén viszonylag kis mennyiségű kód változik az előző (commit esetén sikeresen lefutott) SW verzióhoz képest, így hamar eredményes lehet ez a fajta hibakereső módszer.
- A rövid ciklusok alkalmazása segít a fejlesztőknek kisebb egységekre lebontani a feladatokat, mely hosszú távon jobb minőségű, átláthatóbb, modulárisabb kódot eredményez.

3.2.5. Minden commitnak le kell fordítania a trunkot egy integrátori gépen

- A frekventált, napi commitok alkalmazásával elméletben a trunkon folyamatosan egy leforduló SW verziónak kellene lennie. A gyakorlatban azonban ez nem mindig van így. Általában vagy azért, mert a fejlesztők nem mindig végzik el a frissítést a trunkról, vagy a commit előtti fordítást mellőzik, esetleg a fejlesztői környezetek közti különbségek miatt jön elő a hiba. Ennek eredményeképp biztosítanunk kell, hogy a commit befejezése előtt egy sikeres fordítást végre kell hajtanunk az aktuális SW verzióval egy integrátori gépen (commit build).
- A fejlesztői gépen történt sikeres fordítás után, egy sikertelen integrátori fordítás úgy jöhet létre, ha a saját gépünkön való fordításunk alatt valaki commitolt a trunkra, vagy egy frissen fejlesztett komponensünk, vagy bizonyos megváltozott konfigurációs fájlok nem kerültek fel a repositoryba.
- A fejlesztőnek figyelnie kell a trunkon lévő verzió fordítását, hogy amennyiben hibás, azonnal ki tudja javítani és ne kerüljön hibás SW a trunkra. Ökölszabályként, ha a fejlesztő commitol egy módosítást, addig nem mehet haza, amíg az nem fordul a trunkon.

3.2.6. Sikeres fordítás a CI tool segítségével

- Az automatikus build folyamat támogatására célszerű egy CI szervert alkalmazni. Minden alkalommal, amikor egy commit érkezik a repositoryba, a szerver azonnal checkoutolja a forrásokat egy integrátor gépre, elindítja a fordítást és értesíti a fejlesztőt az eredményről. A commit folyamata nem teljes, amíg a fejlesztő nem nyugtázta az eredményt.
- A legtöbb CI tool támogatja az ún. pretested commitokat (preflight build, vagy personal build), mely során ahelyett, hogy manuálisan töltenénk fel a változtatásainkat, a CI tool fogja a kódban lévő módosításainkat és azzal buildeli a szoftvert a szerveren. Sikeres fordítás esetén a verzió automatikusan felkerül a trunkra, egyéb esetben tájékoztatást kapunk a hibákról. Ezáltal ellenőrizhető a módosításunk hatása a szoftverre

úgy, hogy ez alatt nem kell várni a commit során lefutó automatikus tesztek eredményére és foglalkozhatunk a következő taszkkal, vagy a hibakereséssel.

- Ezt érdemes egy megosztott verziókezelővel is megtámogatni, hogy a változásaink ne a központi szerveren, hanem ideiglenesen helyileg tárolódjanak, így egy sikertelen fordítás esetén könnyebben visszaállítható legyen a korábbi, működő verzió. Mindig legyünk felkészülve arra, hogy ha pár percen belül nem találjuk meg a sikertelen fordítás okát, akkor gyorsan vissza lehessen állítani az előző működő verziót (hiszen bukott build nem kerülhet fel a trunkra).
- A kényszer, miszerint hibás integrátori build után maximum például 10 perc elteltével vissza kell állítani az előző verziót a trunkon, gyakran arra készteti a fejlesztőt, hogy a hibás teszteseteket kikommentezze. Ez csak nagyon indokolt esetben lehet megoldás, akkor is csak különös hozzáértéssel alkalmazható. Számon kell tartani a kihagyott teszteseteket, illetve a build toolban beállítható, hogy sikertelen legyen a fordítás, ha a tesztkódok bizonyos százaléka ki van kommentezve.
- Ajánlott eljárás a build elbuktatása túl lassú tesztesetekre. Ezzel ösztönözhetőek a fejlesztők, hogy minél effektívebb teszteket implementáljanak, így fenntartva a CI optimális alkalmazásához elengedhetetlen megfelelően rövid, commit hatására lefutó integrátori build futási időt. Teljesítmény teszteknél nem ennyire szigorú a feltétel, illetve a gyorsítás sosem mehet a tesztek minőségének rovására.
- Egy módosítás lefordítása mindig a fejlesztő felelőssége, ha sikertelen a fordítás természetes, hogy az azt kezdeményező fejlesztő javítsa ki, akkor is, ha olyan teszteseteken bukott el, amit nem ő írt. Ehhez azonban minden fejlesztőnek hozzá kell férnie a teljes kódbázishoz, tehát nem érdemes programrészeket levédetni, hogy csak egy adott fejlesztő módosíthassa.
- Sok szervezet manuálisan végzi a fordításokat napi gyakorisággal, általában éjszaka. Ez azonban nem elegendő a CI előnyeinek kihasználásához, a hibákat a lehető leggyorsabban fel kell derítenünk, ez esetben viszont akár egy egész napig is a rendszerben lehetnek.
- A módszer lényege, hogy egy stabil trunkra épüljön az összes feature fejlesztése, melyhez a trunkon lévő SW, commit buildet elbuktató bugjait a lehető leggyorsabban meg kell találni.
- Nem hiba, ha elbukik egy fordítás a trunkon, túl gyakori esetben azonban jelezheti, hogy a fejlesztők nem elég körültekintőek a commit előtti frissítésekkel és fordításokkal. Ezen a problémán segíthet egy olyan, több CI tool által támogatott

funkció, amely több szintű trunkot használ. A SW frissítést az igazi trunkból szedi, azonban a commitot egy ún. pending-head branchbe tölti fel először. Sikeres fordítás után a módosítás felkerülhet a valódi trunkra.

3.2.7. Legyen gyors a fordítás

A CI lényege az azonnali visszajelzés. Az egész módszer elveszti a létjogosultságát, ha túl sokat kell várni egy fordítás eredményére. Az, hogy mi számít rövid fordítási időnek, relatív, bizonyos projektek esetén az 1 órás fordítási idő is előrelépés, de az XP útmutatója szerint 10 perces fordítási idő az ideális. Bármeddig is tart a jelenlegi fordítás, érdemes komoly erőfeszítéseket tenni a gyorsítás érdekében, hiszen minden egyes fordítási időből megtakarított perc sok-sok mérnökórát jelenthet a folytonos integráció gyakori fordításai mellett.

A hosszú build és teszt folyamatok hátrányai:

- A fejlesztők nem fognak teljes fordítást csinálni és minden szükséges tesztet futtatni commit előtt, így egyre több hiba jön elő a build során.
- Az integrálási ciklus olyan sokáig tarthat, hogy több módosítás érkezik be egy fordítás lefutása alatt, ezáltal nem tudhatjuk, hogy melyik módosítás miatt bukott el a fordítás.
- A fejlesztők ritkábban fognak integrálni.

A fordítási folyamat gyorsítása egy új projekt esetén ijesztő kihívásnak tűnhet, vállalati alkalmazásoknál a szűk keresztmetszet általában a tesztelési fázis, különösen, ha külső szolgáltatást, mint pl. adatbázist is használnak.
Ennek megoldásaként érdemes több lépcsős build folyamatot alkalmazni (deployment/build pipeline, staged build). Első lépésként mindenképp a trunk commit által triggerelt commit buildet futtatjuk le. Ennek a buildnek gyorsan kell eredményt szolgáltatnia, ugyanakkor kellőképp átfogónak kell lennie ahhoz, hogy releváns információt adjon a commit sikerességéről, ezáltal egy valamelyest stabil alapot adva a további fejlesztésekhez. Meg kell találni tehát az egészséges kompromisszumot a commit build során a SW lefordítása utáni tesztek mennyiségére és minőségére

vonatkozóan az ehhez szükséges idő függvényében. Általában egyszerűbb unit tesztek futnak ekkor, melyek az alkalmazás apró részegységeit tesztelik különállóan (pl. metódus, függvény, vagy ezek kölcsönhatásai). Futtatható a teljes alkalmazás elindítása nélkül is. Nem vizsgálja a külső komponenseket (pl. az adatbázist, a fájlrendszert vagy a hálózatot).

A következő lépésben átfogóbb teszteket alkalmazunk, ezek a komponens tesztek. Az alkalmazás komponenseinek viselkedését tesztelik. Hasonlóan a unit tesztekhez, nem kell a teljes alkalmazást a végső termék környezetben futtatni, de már használhat valódi adatbázis hozzáférést, hálózatot, stb. Ennek megfelelően gyakran több ideig tart, azonban ezeknél a teszteknél is szem előtt kell tartani a futási idő csökkentését, amennyiben fél óránál tovább tart, érdemes megfontolni a párhuzamos, virtuális gépeken történő futtatást. Ezekben az esetekben az elsődleges, commit buildet használják a fejlesztők a folytonos integráció ciklusához.

A többi buildet az átfogóbb tesztekkel akkor indítják, amikor van rá kapacitás azon a szoftveren, amire utoljára sikeresen lefutott a commit build. Ilyen tesztek lehetnek pl. az átvételi (acceptance) tesztek, melyek az átvételi követelményeket vizsgálják üzleti szempontból. Elsősorban a specifikált működést vizsgálja olyan aspektusokból, mint a kapacitás, teljesítmény, hozzáférhetőség, biztonság, stb. Leghatékonyabban a teljes szoftveren a végső termékkörnyezetben futtathatóak, emiatt akár egy napnál is tovább tarthat futtatásuk.

Ezen három teszttípus megfelelően kombinált futtatása magas szinten garantálhatja, hogy egy módosítás nem megy a megkövetelt működés rovására.

A tesztek futási idejét is érdemes minimalizálni, bizonyos toolokkal (pl. JUnit, NUnit) kiszűrhetőek a lassú lefutású tesztek, melyeket vagy időre optimalizálunk vagy megvizsgáljuk, hogy hasonló lefedettséget és bizonyosságot milyen egyszerűbb, rövidebb idei futó tesztek futtatásával érhetünk el.

Egy sikertelen második build, ellentétben a commit builddel, nem állítja le azonnal a további fejlesztéseket, de a fejlesztőknek minél hamarabb fel kell oldani a hibákat amellett, hogy a sikeresen commit buildelt szoftvert tovább fejlesztik.

A másodlagos buildek során elbukott tesztekkel kapcsolatban érdemes megfontolni, hogy a commit buildbe bekerüljenek. Amennyire lehetséges meg kell valósítani, hogy minden, a második build során előkerülő bug egy új tesztet eredményezzen a commit

buildben. Ezáltal a commit buildek minden alkalommal egyre hatékonyabbak lesznek, amint egy SW sikeresen átment rajtuk.

Ez a fajta több lépcsős build eljárás akármennyi szintre kibővíthető, a commit build utáni tesztek több gépen párhuzamosan is futtathatók, ezáltal későbbi automatikus átvételi, integrációs vagy teljesítmény tesztek is beilleszthetőek a teljes build folyamatba. A modern CI toolok támogatják a több lépcsős teszt folyamatot, melyekkel párhuzamosan futtathatóak taszkok, és összesítve jeleníti meg a lefutás eredményeit.
Az átvételi tesztek csoportosítását is érdemes megfontolni, ezáltal a rendszer egyes funkcionalitásaira fókuszálva futtathatunk teszt csoportokat, miután az adott területen módosult a kód.
Emellett a teljes rendszer is feloszthető különböző modulokra, melyek a működésük tekintetében egymástól függetlenek. Ezt körültekintően kell megvalósítani mind a verziókezelés, mind a CI tool konfigurációja szempontjából, viszont komoly előrelépést jelenthet a build gyorsítása szempontjából.
A build gyorsításának gyakorlati megoldásairól az 5. fejezetben olvashatunk.

3.2.8. Teszteljünk egy duplikált termék környezetben

A tesztelés lényege, hogy ellenőrzött körülmények között bukkanjanak fel a termék működése közben fellépő hibák. Ennek fontos feltétele, hogy ugyanolyan környezetet állítsunk elő, mint amilyenen később az alkalmazás során a SW futni fog. Minden apró különbség növeli annak kockázatát, hogy az, ami a tesztelés alatt valahogy működött, az másképp viselkedik a későbbi futtatás során.
Ezért a tesztelés során a lehető leghasonlóbb környezetet kell előállítanunk: ugyanazt az adatbázis-kezelőt, operációs rendszert, könyvtárszerkezetet a szoftverek azonos verzióival biztosítsuk a két rendszerben, még akkor is, ha látszólag nem használja minden elemét. Ugyanazokat a hálózati paramétereket használjuk, és természetesen ugyanolyan hardveren futtassuk.

A gyakorlatban azonban vannak határok. Desktop szoftverek fejlesztése esetén nem lehet tesztelni minden elképzelhető PC-s környezetben, különböző telepített 3rd party szoftverek hatásait figyelembe venni. Hasonlóan, bizonyos esetekben nem megengedhető megfizethetetlenül drága környezeteteket előállítani (licencelt szoftverek). Ez esetben fontos, hogy végiggondoljuk a duplikálás tökéletlenségeinek hatásait, lehetséges következményeit.

Gyakori, hogy egy bonyolultabb kommunikációt használó SW a commit build tesztek során a duplikált termék környezetben lassan, szakaszosan válaszol. Ekkor ezekhez a tesztekhez egy mesterséges, sebességre optimalizált környezetet biztosítunk, a duplikált környezet pedig a másodlagos, átfogó tesztek során használatos.

A duplikált termékkörnyezetre egy ideális megoldás a virtuális gépek használata. Minden szükséges komponens, környezeti beállítás elmenthető a virtualizációba. Ezután már magától értetődő a legújabb fordítás installálása és a tesztek futtatása. Emellett könnyedén futtathatunk több tesztet egy gépen, vagy szimulálhatunk több gépet egy szervergépen.

3.2.9. Tegyünk elérhetővé mindenki számára a legfrissebb lefordított szoftvert

A SW fejlesztés egyik legnehezebb része a követelményeket tökéletesen előre specifikálni. Az emberek, természetüknél fogva általában könnyebben találják meg a hiányosságokat egy félig kész működő rendszerben, minthogy minden alap nélkül, előre, minden feature-t meg tudjanak határozni. Az agilis SW fejlesztés inkrementális módszerei ezt az emberi tulajdonságot használják ki.

Ehhez azonban mindenki számára hozzáférhetővé kell tenni az aktuális futtatható állományt demonstrációs, tesztelési célokra, vagy csak áttekinteni, hogy mi változott az előző ciklus óta. Egy jól meghatározott helyen célszerű tárolni a futtatható fájlokat néhány korábbi verzióval együtt. Fontos, hogy a legutóbbi, commit builden átmenő verzió elérhető legyen, hogy egy stabil alapot adjon a fejlesztőknek pl. további tesztelésre.

3.2.10. Mindenki láthassa, hogy mi történik

A CI egyik alapelve a folyamatos kommunikáció a fejlesztők között, amihez elengedhetetlen, hogy mindenki lássa a rendszer állapotát, illetve, hogy milyen változások történtek benne.

Talán a legfontosabb a trunk fordítási állapotának közzététele. Erre a CI toolok nagy része webes nézetet alkalmaz, melyben minden, jogosultsággal rendelkező felhasználó informálódhat az egyes fordítási folyamatok állapotáról. Emellett az előzmények által bármelyik korábbi fordítás visszaállítható, az egymást követő fordítások közötti módosítások is visszakereshetőek. Az egyes szerverek állapota is nyilván van tartva, így ha pl. egy licenccel rendelkező gépre várunk, megtalálhatjuk az azon valamit épp futtató fejlesztőt. A webes ablak segítségével nem kell egy helységben lennie a fejlesztő csapatnak, vagy bárkinek, akinek fontos lehet a fordítások állapota. Ez egy több száz fejlesztőt alkalmazó cégnél elkerülhetetlen.

3.2.11. Automatikus telepítés

A Continuous Deployment (CD) szintén egy egyre nagyobb teret kap az agilis fejlesztést alkalmazó cégeknél. A tesztelést a korábban bemutatottak szerint nem csak egy környezetben végezzük, a commit build tesztjei egy egyszerűsített, gyors eredményt szolgáltató gépen, az átfogó másodlagos tesztek pedig a termék környezetéhez minél inkább hasonló környezetben futnak. A futtatható állományokat tehát naponta többször kell különböző gépekre másolni és telepíteni, amit célszerű automatizálva, scriptekkel végrehajtani.

Ennek természetes következménye, hogy a készterméket is ugyanilyen egyszerűen telepíthetjük a céleszközre. Ugyan erre nincs napi rendszerességgel szükség, de mivel ugyanazokat az erőforrásokat használja, ez is segíti a folyamatok gyorsítását és a hibák számának redukálását. Ez esetben meg kell valósítani az automatikus visszaállítás lehetőségét. Ha valami hiba jelenik meg a céleszközön futó szoftveren, fontos, hogy minél előbb visszaállítható legyen egy működő verzió. Egy ilyen automatikus funkció

leveszi a telepítéssel járó nyomást a fejlesztőkről, bátrabban telepítenek új verziókat, ami így hamarabb eljuthat a fogyasztókhoz.
Hálózatos környezetben megvalósítható, hogy a szoftvert egyszerre csak egy node-on telepítik, majd optimális működés esetén fokozatosan cserélődik le a korábbi verzió az egész hálózaton pár óra leforgása alatt.
Hasonló megoldás webes alkalmazások fejlesztése esetén, hogy a szoftvert csak a felhasználók egy alcsoportjánál telepítik, majd a tapasztalatok alapján döntik el, hogy az egész csoport használja-e az új verziót. Ezáltal tesztelhetőek az új funkciók és a GUI, mielőtt meghoznánk a végső döntést a verzió sikerességéről.

A Continuous Deployment bevezetése

A legtöbb CI tool támogatja, hogy sikeres build után egy script automatikusan telepítse az alkalmazást egy teszt szerverre, ahol demózhatóvá válik az új feature. Ennek alkalmazásával is lerövidül a fejlesztési ciklus, kisebb projektek esetén akár 50 telepítés is történhet egy nap.
Bevezetéséhez kell egy alkalmas CI tool, mely támogatja, hogy egy központi helyen mindenki számára hozzáférhetőek legyenek az automatizált tesztek eredményei (pl. BuildBot). Továbbá egy verziókövető rendszer, ami támogatja a *hook*okat (esemény hatására lefutó kódrészek). Futtathasson kódokat a commit után, de még mielőtt felkerülne a szerverre – ha valahol hibás lenne az új feature (bármelyik automatizált teszten elbukott), akkor a hook tiltsa le, hogy új kód kerülhessen be a repositoryba. Ez az első visszajelzési hurok, mely biztosítja, hogy megfelelő minőségű kód jöjjön létre, és a fejlesztők ne dolgozzanak "előre" feleslegesen.
Ezután fokozatosan felépítjük a telepítő szervert. Egy telepítő script fokozatosan telepíti gépről gépre a szoftvert, folyamatosan monitorozva és logolva azt, hogy hiba esetén könnyen vissza lehessen térni a hiba helyére. Emellett a telepítések után is kell tesztelni a szoftvert, mert az automatizált teszteken átjuthatnak olyan kódrészletek, melyek csak a telepítés után órákkal, vagy napokkal okoznak problémát. Erre első körben alkalmasak lehetnek terhelési, memória használati statisztikákat vizsgáló szoftverek. Amennyiben ilyen jellegű problémát észlelünk, érdemes leállítani a fejlesztést a hiba okának felkutatásának idejére. Ezalatt érdemes megvizsgálni, hogy a bug miért nem bukott ki a

CI vagy a CD során, és ez alapján tovább fejleszteni, optimalizálni ezeket a folyamatokat további automatizált tesztek hozzáadásával [13].

3.3. A folytonos integráció előnyei

A CI igazi fegyverténye a kockázat redukálása. A halogatott integráció nagy veszélye, hogy nehezen becsülhető meg az integrációhoz szükséges idő. Ennek eredményeként kialakul egy bizonytalan szituáció a fejlesztés egyik legkritikusabb pontján, mely abban a ritka esetben is problémát okoz, hogyha az addigi határidőket sikerült tartani.

A CI ezt a problémát teljesen feloldja; nincs hosszú integráció, mindig ismert, hogy hol tart a projekt, mi működik, mi nem, milyen bugok vannak a rendszerben. A CI nem tűnteti el a bugokat, de nagyon hatékonyan megtalálhatóak és eliminálhatóak általa. A gyakori integrációk, és a kis módosításoknak köszönhetően hamar felfedezhetőek, és javíthatóak a hibák.
A CI alkalmazásával tehát kevesebb buggal találkozunk mind a fejlesztés során, mind a késztermékben. Ugyanakkor nem lehet eléggé hangsúlyozni, hogy optimális eredmény csak megfelelő tesztesetek konfigurációjával érhető el. Nem nehéz úgy módosítani a teszteseteket, hogy jelentős előrelépést érjünk el a hibák felderítésében, de általában hosszas vizsgálódásokkal és a tesztek folyamatos fejlesztésével juthatunk el az alapos, átfogó hibakereséshez.
A folytonos integráció lehetővé teszi a gyakori telepítést is, mely által sokkal gyorsabban demonstrálható egy új feature, ezáltal gyorsabb lesz a visszajelzés a megrendelő részéről. A hatékony együttműködés lebontja a korlátokat a megrendelő és a fejlesztő csapat között, mely a sikeres SW fejlesztés egyik alapfeltétele.

3.4. A folytonos integráció bevezetése

0. Mindenekelőtt szükséges a fejlesztő csapat hozzájárulása: a CI nem egy tool, hanem egy módszer. A fejlesztő csapat elkötelezett támogatása szükséges a sikeres bevezetéséhez. Be kell látniuk, hogy a legnagyobb prioritás mindig a sikertelen fordítás helyreállítása. Ha a csapat nem áll megfelelő fegyelemmel a

módszertan mellé, akkor nem várhatjuk el, hogy a CI bevezetése a remélt eredményeket hozza.

1. Az első lépés, hogy a fordítás automatizált legyen. Mindent gyűjtsünk össze egy verziókezelőbe, hogy az egész rendszer egyetlen parancs kiadásával fordítható legyen. Ez egyes projektek esetén nem kis vállalkozás, de elengedhetetlen a további eljárások alkalmazása előtt. Először igény szerint elég alkalomszerűen fordítani, vagy az éjjeli fordításokat elindítani.

2. Alkalmazzunk automatizált teszteket a build folyamatban. Próbáljuk feltérképezni a kritikus részeit a szoftvernek, melyek sok hiba forrásai, és ezekre kihegyezve fejlesszük a teszteseteket.

3. Próbáljuk minél jobban felgyorsítani a fordítást. CI alkalmazása néhány órás fordítás mellett is megvalósítható, de törekedjünk, hogy elérjük a tíz perces álomhatárt. Ez gyakran súlyos beavatkozásokat igényel a kódban, a függőségek feloldása a lassú részeken gyakran sokat gyorsít a fordításon.

4. Ha új projektet indítunk, alkalmazzuk a folytonos integrációt a kezdetektől. Kövessük figyelemmel a fordítási időket és cselekedjünk, ha tíz perc fölé megy. Gyors beavatkozásokkal kevesebb befektetéssel megvalósítható a kód átkonstruálása, amíg nem túl nagy a kódbázis.

4. Folytonos integráció a ThyssenKrupp Prestánál

Ebben a fejezetben megismerjük a ThyssenKrupp Prestánál alkalmazott automatikus fordítási eljárásokat, illetve megvizsgáljuk, hogy a cégnél milyen agilis eljárások bevezetése ajánlott.

4.1. Build Szerver

A fordítási idők az AUTOSAR bevezetésével jelentősen megnőttek, több, hosszabb forrásfájt kellett fordítani, illetve a beinkludált headerek sorainak száma is közel százezres nagyságrendű. Ezen kívül bizonyos projekteknél több körös fordítási folyamatra kellett áttérni, mely egy kód optimalizációs eljárás (a kód kisebb helyet foglal, gyorsabban fut), azonban a SW fordítási ideje emiatt megnő.

A fejlesztők az implementált módosításokat a saját gépeiken fordították, ezáltal projekt és platformfüggő fordítási idők 20 perctől akár másfél óráig is terjedhettek. További probléma, hogy ezalatt a gép erőforrásai olyannyira ki vannak használva, hogy más alkalmazás használata szinte lehetetlen, így nő a felesleges mérnökórák száma.

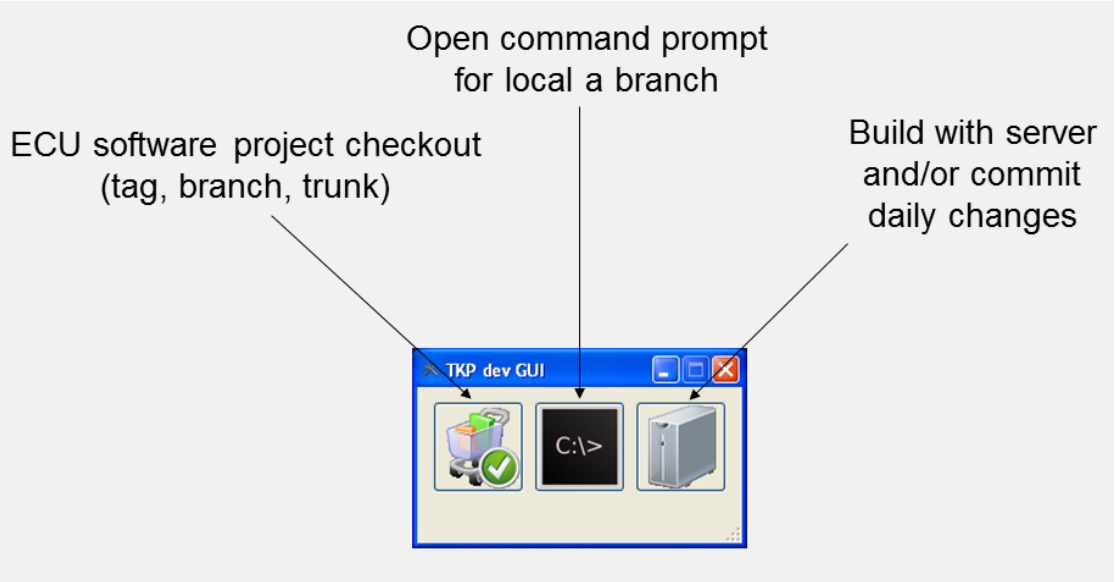
Emellett a rendelkezésre álló licencek száma is korlátozza a fordítások lefutását, amennyiben minden licenc használatban van, várni kell valamelyik felszabadulására. A korábban használt Elan licenc-menedzser kényszerű lecserélésével minden licencváltásnál egy közel 5 perces üresjárat lép fel, ameddig nem használható a kiharcolt licenc. Egyes becslések szerint a jelenlegi licencek 120%-a lefedné az átlagszükségletet, illetve kétszer ennyi licenc elegendő lenne a bizonyos csúcsidőkben fellépő maximális szükséglet kielégítésére [14].

Ezek együttes eredményeként jelentősen megnőtt a fordítási idő, mely a folytonos SW integráció bevezetésénél megoldandó probléma. A fordítási idők csökkenésével nem csak mérnökórákat spórolhatunk meg, de a gyorsabb lefutás által a licenceket is rövidebb ideig foglalják a fordítások.

A folytonos integráció egyik sarkalatos pontja, hogy a fejlesztőknek ne legyen „terhes” egy nap akár többször integrálni a szoftvert, és megfelelően kevés módosítás kerüljön vissza a rendszerbe, ezzel segítve a korábban taglalt folytonos integrációval járó előnyöket. Amennyiben sokáig tart egy integráció, vagy körülményes annak kivitelezése, a fejlesztők ritkábban szánják rá magukat és adott esetben összevárnak több módosítást integráció előtt, mellyel a folytonos integráció lényege veszne el. Tehát meg kell valósítani, és be kell tudni vezetni, hogy a fejlesztési issue-kban kiírt feladatok egy egységes felületről lehessenek megvalósíthatók. Csak a módosítani kívánt komponensek kerüljenek checkoutra a verziókezelt szoftverből, az ezeken végzett módosítások bizonyos alapvető tesztek után kerüljenek vissza a rendszerbe. Ezután automatikusan, esemény- vagy idővezérelten fusson le a fordítás, majd a lefordult szoftveren az automatikus tesztek futása után álljon rendelkezésre az adott funkcionalitás. Egy riport is generálódik, hogy mely fázisokon jutott át a SW, illetve milyen erőforrásokat használt ezalatt.

Ezen feladatok ellátására készült a saját fejlesztésű Build Server Tool első verziója, mely a grafikusan kiválasztható módosítandó komponensek mellett a közös szerveren történő fordítást is támogatja. Célunk, hogy minél több projekten általánosan használt tool legyen, és a fejlesztők fel tudják adni a korábban rutinszerűen alkalmazott fordítási procedúrát a tool használatának javára. Ehhez azonban biztosítani kell a SW buildek lehető leggyorsabb lefutását, mely egy apró részfeladatnak tűnhet a folytonos integráció bevezetésében, azonban a lehető legjobb futási idő elérése érdekében megvizsgált különböző lehetőségek meghatározhatják a használt fordítási eljárást (preprocesszor, compiler váltás, cloud megoldások).

A jelenlegi Build Server Tool a TKP dev GUI a szerver mellett helyi fordításra is alkalmas (szerverleállások esetén is lehessen fordítani).

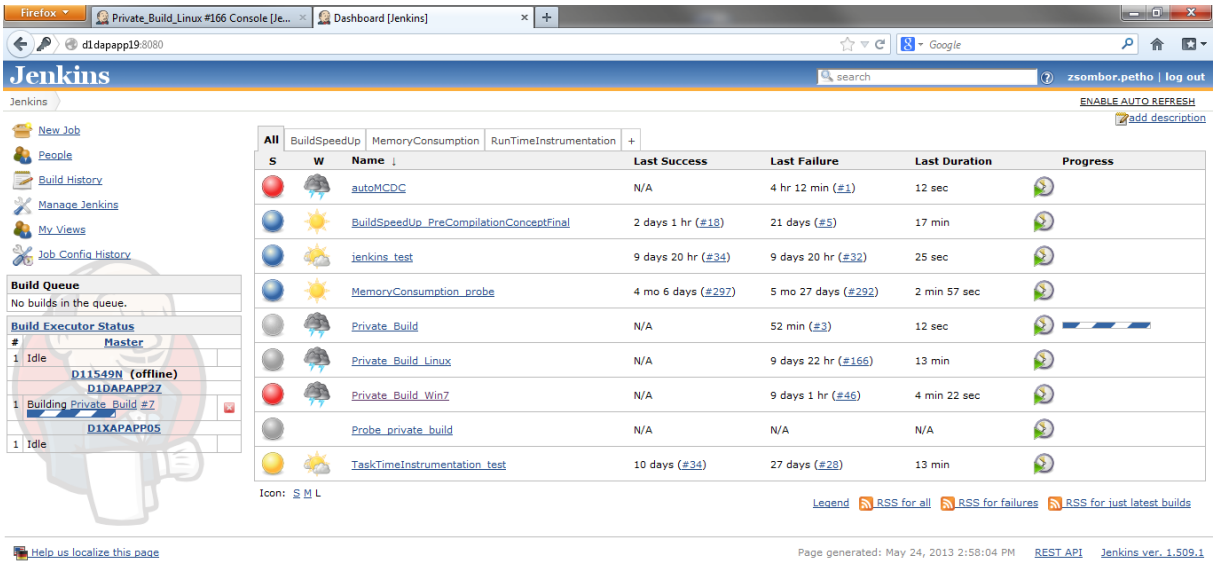


4.1.1. ábra : a TKP dev GUI főablaka [15]

Minden fordítás eredménye *Jenkins*ben visszakereshető. A Jenkins egy JAVA-ban fejlesztett, nyílt forráskódú, folytonos integrációs tool. Szerver alapú, böngészőből bejelentkezve elérhetőek a fordítási feladatok (jobok). Real-time követhető a fordítás állapota, illetve az így készült teljes logok később is elérhetőek [15].

A TKP dev GUI-ban az alábbi folyamatokat kezeli le a Jenkins:

- a privát branchet kikéri a verziókezelőből
- ellenőrzi, hogy mely build toolokat kell fordítani (ha nem létezik, ezeket is leszedi az SVN-ből)
- átmappel a T virtuális meghajtóra
- beállítja a környezeti változókat
- lefordítja a szoftvert
- az eredményt becsomagolva bemásolja a build szerver megosztási mappájába
- e-mail értesítést küld a felhasználónak



4.1.2. ábra : A Jenkins főablaka [saját forrás]

4.2. *Ajánlás*

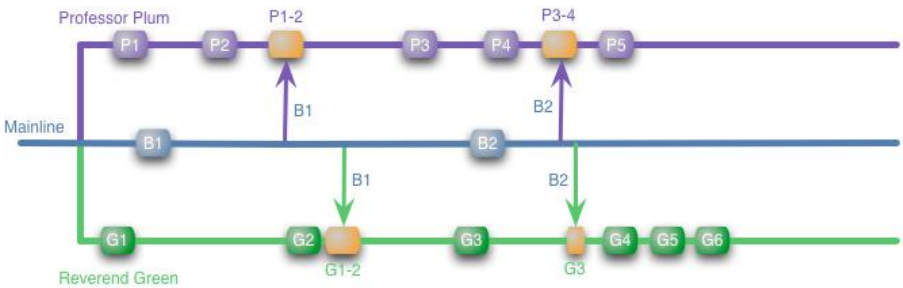
Ebben a részben megvizsgáljuk, hogy milyen további agilis technológiákat érdemes alkalmazni a folytonos integráció bevezetése mellett. A szakirodalom kutatás eredményeit a csoportvezetőknek egy meeting keretében mutattam be. Ennek során az alábbi javaslattal álltam elő a ThyssenKrupp Prestánál bevezetésre kerülő folytonos integrációs fejlesztő rendszer támogatására.

4.2.1. *Feature Branch technikák [16]*

A CI hatékony alkalmazása során fontos, hogy a verziókezelő lehetőségeit előnyé kovácsoljuk, így érdemes átgondolni, hogy egy feature fejlesztése során milyen stratégiát alkalmazunk a fejlesztési ágak kezelésében.

1. Izolált branchelés

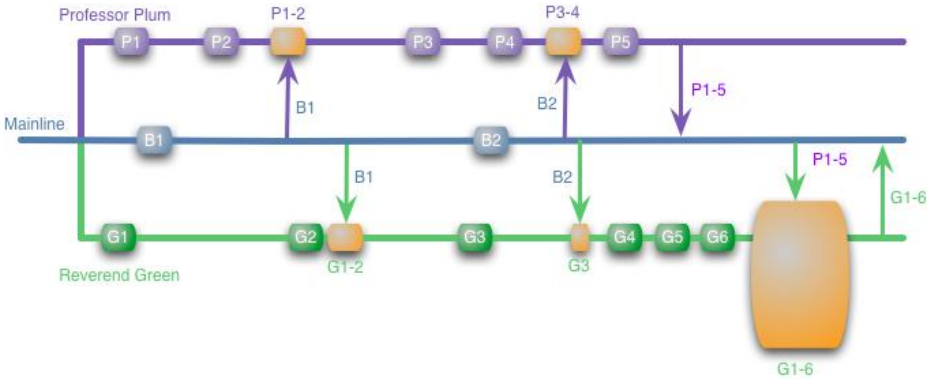
Minden fejlesztő a saját feature-höz nyit egy branchet a trunkról, bugfixek esetén updatelnek a mainline-ról, naponta egyszer commitolnak a saját branchükre. Megfelelő érettségi szint mellett, és ha a fejlesztői csapat is úgy gondolja, felcommitolják a feature-üket a trunkra. Ezáltal a fejlesztői csapat dönthet arról, hogy melyik feature legyen benne a release-ben, és melyik nem.



4.2.1. ábra : Izolált branchelés [16]

Azonban, ha mindkét fejlesztői feature bekerül a release-be, felléphetnek bizonyos problémák a merge során. Amíg az első fejlesztő (Prof. Plum) minden gond nélkül felcommitolja a módosításait (mivel folyamatosan frissítette a verzióját a trunkról),

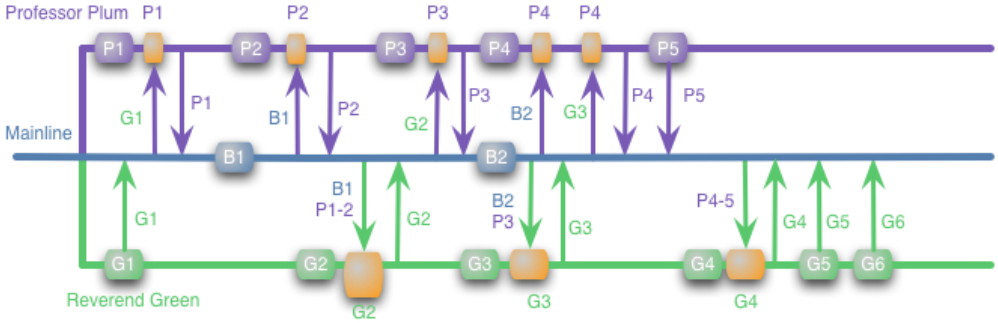
addig a hasonló módon fejlesztő, de csak később commitoló 2. fejlesztőnek (Rev. Green) már össze kell fésülnie a kódját az első fejlesztőnek eddig nem ismert módosításaival (P1-5).



4.2.2. ábra : Körülményes összefésülési eljárás izolált branchelés esetén [16]

Ez szintaktikailag manapság már kielégítően támogatott a szofisztikált verziókezelők által, de a szemantikai problémákat egyelőre nem képesek hatékonyan feloldani. Pl. ha Prof. Plum átnevezett egy metódust/függvényt, amit Rev. Green fejlesztő meghívott.
Ezek a nagy kódrészletek elrettentik a fejlesztőket a kódszépítéstől, mely tovább nehezíteni az amúgy is körülményes összefésülést. Ez hosszú távon nagyon rossz hatással lesz a kódbázisra.

2. Continuous Integration Branchelés

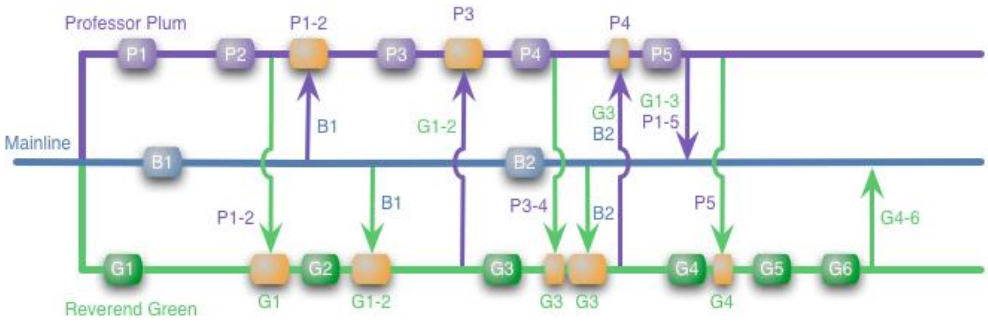


4.2.3. ábra : CI branchelés [16]

A folytonos integrációs branchelés során a fejlesztők legalább napi rendszerességgel commitolnak a mainline-ra, így a több fejlesztő közötti konfliktus hamar kibukik, és a kisebb mennyiségű módosítások miatt könnyebb a debuggolás is. Természetesen folyamatosan frissítik a mainline-ról a branch verziójukat.

További előnye, hogy a közös mainline folyamatos kommunikációra készteti az azonos feature-ön dolgozó fejlesztőket, ami nagyban megkönnyíti a SW fejlesztést.
A CI hatására mindig release-re kész verzió van a mainline-on, azonban ez az eljárás nem engedi meg az egyes szükségtelen vagy félig elkészült feature-ök eliminálását a kódbázisból. Ezt azonban a GUI-ból könnyedén kihagyhatjuk, és biztosak lehetünk abban, hogy a felesleges kódrészletek nem okoznak problémát a szükséges funkcionalitások használatakor. A klasszikus verziókezelők ez a technikát teljes egészében támogatják.

3. *Vegyes Branchelés*



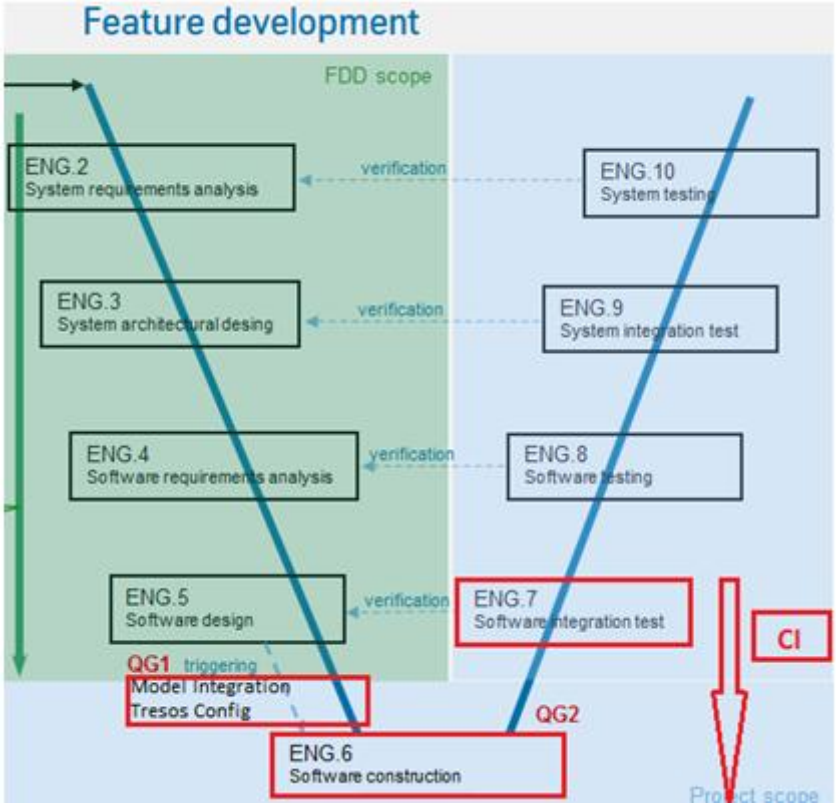
4.2.4. ábra : Vegyes branchelés [16]

A vegyes branchelés során nem a mainline-ra commitolnak a fejlesztők, hanem amennyiben tudják, hogy ugyanazon a kódrészleten módosítanak mind a ketten, akkor csak az egymás verzióit fésülik össze. Így ijesztően nagy mergelési feladat nem alakulhat ki, mint az izolált branchelés esetén.
Ez esetben nem is feltétlenül van szükség mainline-ra, hasonlóan az elosztott verziókezelőkhöz a fejlesztők egymás verzióira frissítenek.
Az izolált branchelés előnyeit is magában hordozza, hiszen az egyik fejlesztő eldöntheti, hogy a másik fejlesztései közül melyik feature-t tartalmazó verzióval mergeli össze a saját branchjét.
A legnagyobb nehézség a vegyes branchelés során a kommunikáció fenntartása, hogy minden fejlesztőhöz eljusson az információ, hogy ki melyik kódrészleten mit módosított. Megfelelő toolok alkalmazásával ez is áthidalható.

4.2.2. Fejlesztési modellek

A különböző fejlesztési modelleket megvizsgálva nyilvánvalóvá vált, hogy az AUTOSAR alkalmazásával a modell és a komponens alapú fejlesztés már jelen van a cég fejlesztési kultúrájában. A Feature Driven Development pedig meghatározza a fejlesztési hierarchiát, ahogy az egyes új funkciók a feature teamen keresztül bekerülnek a V-modellbe és néhány fejlesztési ciklus után elkészül a termék. Ebben már felfedezhetőek az agilis fejlesztés jegyei, mely hosszú távon valóban az alapvető cél. Ehhez kapcsolódnak a TDD alapelvei, melyeknek bizonyos mértékű beépítését a fejlesztési folyamatba megfontolásra érdemesnek találtam. Így azt, hogy a fejlesztők a hozzájuk tartozó komponens módosítása előtt írják meg a teszteseteket, mely által minden új funkció bekerül a tesztesetekkel lefedett kódrészletek közé. Ez a unit tesztelés során megvalósítható, a csoportvezetők támogatták ezt a felvetést, csak idő kérdése, hogy ez is hatékonyan beépüljön a fejlesztői kultúrába.

4.2.3. A folytonos integráció beépítése a V-modellbe

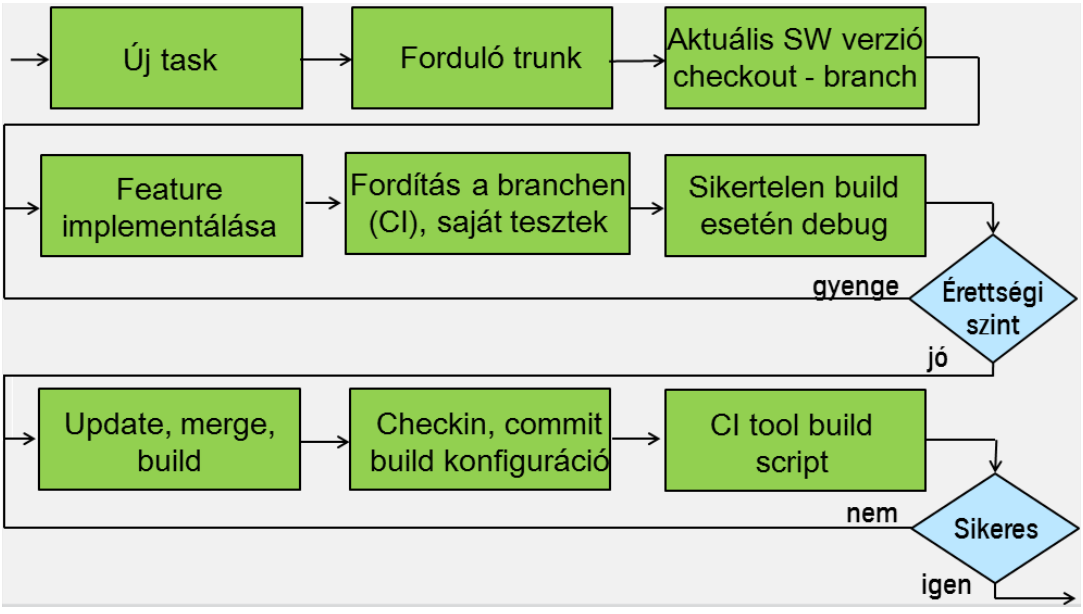


4.2.5. ábra : V-modell a CI lépéseivel [4]

Miután egy új funkció a Feature Team követelmény specifikációja után eljut a V-modell fejlesztési ágáig és később létrejön a SW design, elegendő néhány kiegészítő lépést eszközölni ahhoz, hogy a folytonos integráció alkalmazható legyen a további fejlesztési folyamat során. Így az integrátorok segítségével egy összefordítható integrált modellt kell előállítani, melybe folyamatosan beilleszthetőek a fejlesztők módosításai. Ehhez létre kell hozni a Tresos konfigurációt, azt, hogy milyen komponensek milyen interfészeken fognak kommunikálni az adott feature működése során. Ez a fejlesztés korai fázisában megfelelően körvonalazódik, így a továbbiakban legfeljebb apró módosítások várhatóak ebben a tekintetben.

Ebbe a modellbe integrálják be a fejlesztők a SW konstrukciós lépésben a módosításaikat, és a CI alkalmazásával a tesztelési ág integrációs tesztjét is elvégzik. Ez a jelenleg Quick Check-ként futtatott smoke teszt sikeres lefutását jelenti. A folytonos integráció tehát ezeket a lépéseket fedi le (a 4.2.5. ábrán pirossal jelölve).

A részletes fejlesztési lépéseket az alábbi, 4.2.6. ábrán láthatjuk.



4.2.6. ábra : Egy új taszk fejlesztési folyamata [saját forrás]

4.2.4. Fejlesztés lépései

1. *Új taszk*

   A fejlesztői csoport megkapja az implementálandó feature-höz tartozó specifikációt. Ez alapján a TDD szerint megírják a taszkhoz tartozó unit teszteseteket.

2. *Forduljon sikeresen az aktuális trunk*

   Amennyiben éppen fordítják a SW verziót, segítsék a fejlesztők a minél előbbi sikeres buildet a trunkon. Sikertelen fordítás esetén, ha rövid időn belül nem sikerül feloldani a hibát, akkor állítsák vissza a legutóbbi működő verziót.

3. *Aktuális SW verzió checkout, branch*

   A fejlesztő leszedi a fejlesztői gépére a trunkon lévő aktuális SW verziót, és csinál egy branchet az adott feature fejlesztésének.

4. *Feature implementálása*

   A fejlesztő a letöltött verzióban minimális szinten implementálja a funkciókat, majd a fejlesztői gépén vagy a CI toolt triggerelve fordítja a szoftvert, illetve lefuttatja a unit teszteket.

5. *Fordítás a branchen (CI), saját tesztek*

   A feature-höz nyitott branchen zajlik a fejlesztés CI támogatással, azaz nem feltétlenül a saját munkaállomásukon, hanem egy build szerveren fordítanak. A fejlesztők a saját teszteseteiket futtatják a funkció validálásához.

6. *Sikertelen build esetén debug*

   Csak a feladatért felelős fejlesztői csapat munkáját akadályozza a hibás build, az egész csapat segítse a debuggolást. Amennyiben az alkalmazás nem ért el egy megfelelő érettségi szintet, vissza kell térni az implementációs lépéshez.

7. *Update, merge, build*

Megfelelő érettségi szint mellett a fejlesztői csoport frissíti a workspace-t a trunkra, mergeli és fordítja a szoftvert. Sikeres futtatás esetén húz egy taget és futtathatja a modul teszteket. A jelenleg rendelkezésre álló verzióval az integrátorok segítségével fésülik össze a módosításokat.

8. *Checkin, commit build konfiguráció*

Ha megfelelően friss verzióval a branchen helyesen működnek a módosítások, akkor commitolhat a verziókezelőbe. Emellett be kell illesztenie a commit build folyamatba a funkcionalitáshoz tartozó teszteket.

9. *A CI tool automatikusan elindítja a build scriptet*

A CI tool a commit hatására automatikusan elindítja a commit build folyamatot (fordítás + commit tesztek)

10.

a. *Sikertelen fordítás esetén javítsa a hibát*

Ha a commit build elbukott, a commitoló fejlesztőnek azonnal ki kell javítania a hibát. Ha ez rövid időn belül nem teljesül, akkor állítsa vissza az előző verziót a trunkon, és a branchen javítsa ki a hibát.

b. *Sikeres fordítás esetén új taszk*

Sikeres fordítás esetén a kommentezze a commitot és folytathatja a következő taszkkal.

4.2.5. *CI támogatás AUTOSAR-os környezetben*

Az irodalomkutatás során találtam rá az AUTOSAR-os programcsomagokat támogató Elektrobit cég egy, az utóbbi években kiadott kiegészítő SW eszközére. Ez a Tresos Wincore [17] tool, mely a leírás szerint megkönnyíti az agilis eljárások, így a CI alkalmazását is az AUTOSAR-os fejlesztés során.

A tool gyakorlatilag elfedi a HW függő részeket, ez az AUTOSAR-os architektúra megfelelő rétegének átkonfigurálásával valósítható meg. Minden HW függő SW modult

tartalmaz, amit az ECU használna (AUTOSAR OS, MCAL: Microcontroller Abstraction Layer). Ennek eredményeként HW nélkül futtatható lesz az AUTOSAR-os ECU SW, tehát korábban elkezdhető a fejlesztés és a build folyamat (konfiguráció, paraméter validáció, kódgenerálás, fordítás és linkelés Windowsos környezetben). A tesztelés elkezdésével a fejlesztés különböző fázisaiban különböző jellegű hibákra következtethetünk. Így máshol kell keresnünk a hiba forrását HW független funkcionális tesztelés során, a konfiguráció vagy a HW illesztése után. További előny, hogy az ütemező szabadon konfigurálható, tehát nem kell real-time debuggolni az ECU-n futó kódot. Egyes becslések szerint a kódbázis közel 80%-a tesztelhető előzetesen a Wincore tool segítségével. Emellett létezik egy kiegészítő, Debug & Trace nevű tool, mely az architektúra rétegei közti adatforgalom követését teszi lehetővé.

Egy ilyen jellegű tool a ThyssenKrupp Presta esetében is hatékonyan támogatná a CI bevezetését, hiszen ezáltal gyakorlatilag eliminálnánk a beágyazott fejlesztésből fakadó nehézségeket. Automatizáltan, teljesen PC-s környezetben működhet az ECU SW komponensei jelentős hányadának build folyamata. A tool fejlesztéséről a 7. fejezetben olvashatunk.

5. *A SW fordítás folyamata*

Korábban megismertük a CI bevezetésének követelményeit, így a megkerülhetetlen build futási idő optimalizálást. Ennek érdekében megismerkedünk a SW build folyamatával, mialatt a forráskódból futtatható állomány keletkezik. Továbbá megvizsgáljuk az optimalizációs eljárásokat, illetve a cégnél alkalmazott fordítási eljárást.

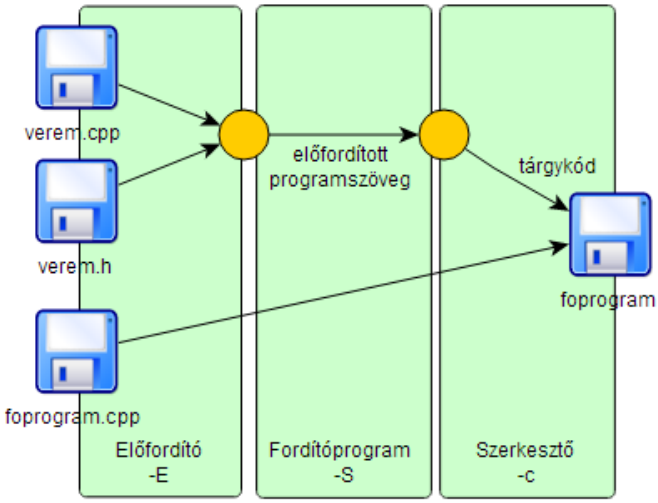
Ahhoz, hogy optimalizálni tudjuk ezt a folyamatot, vizsgáljuk meg, hogy mi is történik, amíg a forráskódból futtatható bináris állomány keletkezik.
A fordítás tehát a forráskód elkészítésével kezdődik, mely esetünkben egy programozási nyelv definícióit, utasításainak sorozatát tartalmazza. Ez a nyelv a TKP-s kormányvezérlő szoftverek esetében, a beágyazott rendszerek fejlesztésében szinte egyeduralkodó C nyelv, mi ennek a fordítási folyamatát szeretnénk optimalizálni. Ezen kívül számos nyelven íródnak még segédprogramok. Különböző környezeteket, akár tesztelésekhez magasabb szintű nyelveken valósítják meg (IronPython, melyben a korábbi fejlesztések születtek és keretet ad a *dll* fájlok segítségével .NET és JAVA függvények beintegrálásához). Script nyelveken is gyakran íródnak tool-ok, nagy adatmennyiséggel könnyebb automatizáltan műveleteket végezni (PHP, Perl), valamint az adatbázis kezelő (MySQL).
Ezt a forráskódot átadjuk általában egy komplex fordítóprogramnak, mely több almodul segítségével állítja elő az adott platformon futtatható kódot. Ezt hívjuk SW build folyamatnak, a magyar terminológiában az összeállítás kifejezés nem igazán terjedt el, így vagy buildként vagy kicsit helytelenül csak fordításként hivatkozunk rá, mely valójában a preprocesszálás és a linkelés közti kódgeneráló lépésre (compile) utal. Azonban erre viszonylag ritkán hivatkozunk, így a továbbiakban, a könnyebb magyar olvashatóság érdekében én is fordításként utalok a build folyamatra, a köztes lépésre pedig compile-ként.
Először az előfordító (preprocessor) kapja meg a forráskódot, melynek feladata a programozók számára könnyebben olvasható kód előzetes szöveges szerkesztése. Többek között behelyettesíti a beinkludált header file-okat, illetve makrókat, a

megjegyzéseket szóközre cseréli. Ezen kívül ellenőrzi az *#ifdef*, illetve *#ifndef* makróknak megfelelő feltételeket, mely a header guard funkciót valósítja meg. Ez ugyanannak a fejállománynak a többszörös beillesztéséből fakadó problémákat (pl. többszörös definíció hiba) küszöböli ki. Ezeken kívül az összes # jellel kezdődő sor az előfordítónak szól, illetve használható feltételes fordításra (kódrészletek beillesztése és eltávolítása). Nem végez szintaxiselemzést, nem értelmezi a forrásszöveget [18].

Ezután maga a fordító program modul kapja meg a fájlokat. Ekkor történik a gép számára futtatható kód létrehozása, egy vagy több lépésben. A GCC esetében először ASM kód keletkezik (compilation), majd az assembler alakítja gépi kóddá. A fordító gépfüggetlenül elemzi a magas szintű nyelv utasításait, majd kódot generál, mely függ a processzor utasításkészletétől. Ennek eredményeként jön létre a tárgykód (.o), mely a program gépi kódjának részleteiből áll [19].

Az összes forrásfájl lefordítása után ezeket a kódrészleteket kapja meg a szerkesztő (linker). Feloldva a kereszthivatkozásokat beépíti a könyvtárakat és a különböző objektumokat egy végső, futtatható programmá szerkeszti. Ez alacsonyabb szinten általában statikusan történik a fordítás utolsó fázisában, futási időben nincs szükség további szerkesztésre.

A komplex fordítóprogramok tartalmaznak bizonyos kapcsolókat, melyekkel megállhatunk a fordítás különböző fázisait követően. Így nem kell feltétlenül egyféle fordítóprogrammal végigmennünk a fordítás fázisain, adott esetben a gyorsabb lefutás érdekében használhatunk különböző modulokat.



5.1.1. ábra : A SW fordítás folyamata [20]

Egyszerűbb programok esetében egyetlen fájl is tartalmazhatja a fordítandó forráskódot, viszont egy olyan komplex SW esetében, mint a TKP kormányvezérlő szoftvere természetesen ez nem szempont. A könnyebb fejleszthetőség érdekében kifejezetten ajánlott diverzifikálni a bizonyos SW modulokat. A fordítás során egy build tool program segítségével több különböző módon fordíthatunk le egy szoftvert, ezáltal különböző verziókat létrehozva. A TKP a make build toolt használja, mely speciális makefájlokban tartalmazza a fordítások koordinálását. Ha a forráskód egy bizonyos fájlban nem változott, nem érdemes, sőt nem ajánlott újrafordítani. Ez nem csak időben jelent előnyt, hanem mivel ez a fájl függhet más, időközben megváltozott fájloktól, a sikeres fordulást is elősegíti.

5.1. *A Compiler*

A compile folyamat tehát az, amikor a magasabb szintű forráskódból futtatható objektumokat állítunk elő. Ha az így előállított állományok más platformon vagy más OS-en futnak cross-compilerről beszélünk. A beágyazott rendszerekre tehát cross-compilereken fordítunk, hiszen ezek nem támogatnak SW fejlesztő rendszereket. Fordított irányba, tehát alacsony szintről magas szintre a decompiler alakítja vissza a kódot. Ez nem egy klasszikus inverz művelet, a decompiler nem tudja tökéletesen reprodukálni a magas szintű kódot, de így is nagy jelentősége van a SW reverse engineering témakörben.

5.1.1. *A compiler rétegei*

A ***front end*** réteg ellenőrzi az adott programozási nyelv szintaxisát és szemantikáját. Típusellenőrzést is végez, az esetleges hibákat eltárolja. Egy típus rendszer meghatározza, hogy a programnyelv hogyan sorolja be a változók értékeit és a kifejezéseket típusokba. A rendszer feladata, hogy megvalósítsa és verifikálja a kód egy bizonyos szintű megbízhatóságát feltárva a helytelen típus műveleteket. A front end a

kód egy közbenső reprezentációját (intermediate representation - IR) valósítja meg, melyet átad a következő rétegnek.

A legtöbb optimalizációs folyamat a ***middle end*** rétegben zajlik. Jellemző műveletek a hatástalan vagy nem elérhető kódok eliminálása, a konstans változók feltérképezése, számításigényes kódrészletek áthelyezése ritkábban végrehajtott részekre (pl. a ciklusból kivéve). Mikrokontrollerek esetén ügyelni kell az automatikusan optimalizált kódrészletekre, pl. egy megszakítás rutinban növelt számláló értékének módosítása nem hívódik a kódból, így a fordító az inicializált konstanssal helyettesítené a változót. Ennek kivédésére a *volatile* kulcsszót használhatjuk, mely jelzi a fordító számára, hogy az adott változóval kapcsolatban nem szabad optimalizációt végeznie.

A ***back end*** réteg felelős a middle endtől érkező IR assembly kóddá alakításáért. Minden IR utasításhoz ún. *target* utasítást/utasításokat választ. A regiszter allokáció CPU regisztereket rendel a program változókhoz ott, ahol lehetséges. Két regiszter típusú target életciklusának páronkénti egybevetésével határozhatjuk meg, hogy ábrázolhatóak-e egy regiszterben. Így használhatjuk ki legideálisabban a párhuzamos végrehajtó egységeket, tölthetjük ki a delay slotokat (pl. az arbitráció során kódfüggetlenül végrehajtandó utasításokat) [21].

5.1.2. Multi-pass compiler

A több körös fordítás az első generációs számítógépek szegényes HW erőforrásai miatt alakult ki. Nem volt elegendő memóriájuk ahhoz, hogy egyetlen program hajtsa végre a fordítási feladatokat. Így ezeket a feladatokat több, kisebb erőforrás igényű program látta el, melyek egymás után futottak le a forráskódon, illetve az előző lefutás kimenete lett a következő bemenete.

A később alkalmazhatóvá vált single pass compile haszna az egyszerűbb compiler fejlesztés, illetve, hogy általában gyorsabban fordít, mint egy több körösen lefutó compiler. Bizonyos esetekben azonban úgy történik a kódfejlesztés, hogy több lefutású fordítást igényel. Pl. egy deklaráció a 20. sorban befolyásol egy a 10. sorban található kifejezés fordítását. Ekkor az első körben információt kell gyűjtenie az egyes kifejezések utáni deklarációkról, amennyiben azok befolyásolják a kifejezés fordításának aktuális állapotát. Egyes nyelveket lehet single pass compilerrel fordítani,

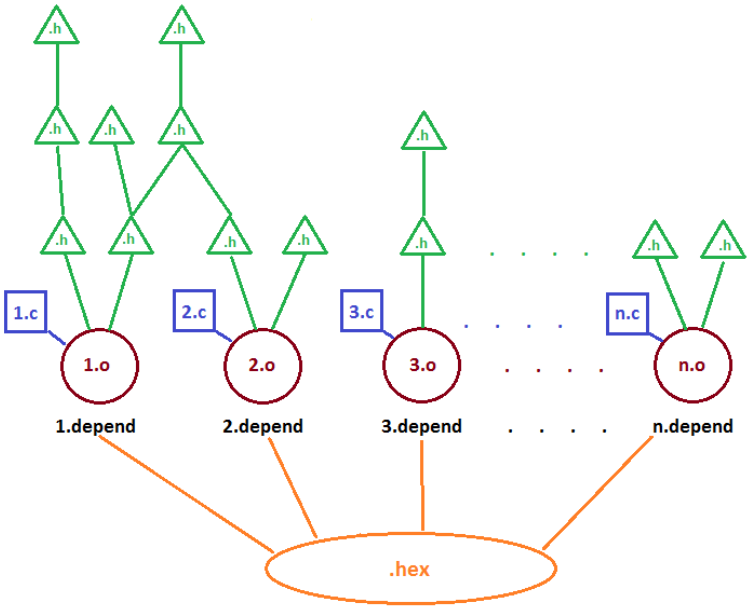
így a PASCAL-t, mely minden deklarációt bekér a használatuk előtt. Azonban pl. a JAVA több körös fordítást igényel, mert nem szükséges definiálni egy változót a használata előtt. Így a fordítás szemantikai analízis fázisában foglalja le a memória helyet ezeknek a változóknak. A single pass fordítások hátránya, hogy nem lehetséges bizonyos kifinomult optimalizációs technikák alkalmazása (kisebb kódméret, rövidebb futási idők,…) [21].

5.2. A make

A make egy segédprogram, mellyel konfigurálható többek között a szoftverek fordításának mikéntje és az ezáltal keletkező target fájlok tartalma. A konfiguráció ún. makefájlokban van leírva, melybe gyakorlatilag bármilyen utasítást meghívhatunk, mely forrásfájlokkal végez valamilyen műveletet, így nem csak a build toolként használható a make.

A make használata során meghívva a make parancsot a program megkeresi a megadott gyökérben található makefájlokat (.mak, .make, Makefile), majd futtatja az ott meghatározott műveleteket. A make egyik gyengesége, hogy platformfüggő műveleteket hívunk meg általa, így az ezek közti átjárhatóságot nehéz biztosítani. Ezt fokozottan tapasztaltuk, amikor Windowsról Linuxos szerverre konfiguráltuk át a makefájlokat. A make felügyeli, hogy változások esetén újra kell-e fordítani a targetet, melyhez a fájlok módosítási idejét veszi alapul [22].

A fordítás folyamán először a make depend utasítást adjuk ki, mely feltérképezi a függőségeket a *.c,* és az ezekbe inkludált *.h* fájlok között. Amennyiben akár a *.c* forrásfájl, akár valamelyik *.h* fájl módosult, a make automatikusan újrafordítja az ebben a lépésben targetként meghatározott *.d* depend fájlt. Mivel egy *.h* fájl több *.c* fájlban is be lehet inkludálva, módosulása esetén természetesen mindegyik depend fájl újragenerálódik. Ezt legegyszerűbben egy fa struktúrában képzelhetjük el, melyet az 5.2.1. ábra is mutat.



5.2.1. ábra : A SW fordítás fa struktúrája [saját forrás]

5.2.1. A make build folyamat

Az optimalizációhoz szükséges futtatási körök száma az egyes fordítóknál ismert paraméter. A TKP-s SW jelenleg használt GHS fordítója esetében 2 körös fordítás elegendő. Az első körös fordítás (firstpass) azért kell, mert a GHS ekkor generálja le az átadott *.c* és a bennük beinkludált *.h* fájlok függőségeit, melyet a második körös fordításnál használ fel. Ez az információ a minden *.c* fájlhoz tartozó *inf* fájlokban található meg. A GHS fordító sajnos módosult fájlok esetén nem generálja újra automatikusan a függőségek vizsgálatát, így a külön depend lépés ezért szükséges. Amikor a több szálon indított második körös fordításnál (secondpass) ezeket az *inf* fájlokat a hozzájuk tartozó *.c* fájlokkal együtt olvassa be a compiler, hibát észlelünk, ugyanis az *inf* fájlokba bizonyos adatokat vissza szeretne írni a compiler, mely több szál esetén összeakadást eredményez. Ennek kivédésére korábban csak egy szálon indítottuk el az *inf* fájlok beolvasását (és írását), majd amikor elkészült visszaálltunk a több szálú fordításra. Ezt egy script segítségével valósítottuk meg. Az *inf* fájlok elkészülte után *.o* fájlokat generál a fordító, így az egy szálon elindított második körös fordítással párhuzamosan figyeltük, hogy hány *.o* fájl keletkezett. A makefájl módosításával is orvosolható a probléma. Elegendő, ha egy szálon elindítva „lerekesztjük” az első *.o* fájl

létrehozását, elkészülte után a többi object fájl már használhatja az összes rendelkezésre álló szálat. Minden *.c* fájlhoz ebben a lépésben keletkezik a már említett, linkelés előtt álló objektum fájl, melyet a linker állít össze a kimeneti *hex*, illetve *map* fájlokká.

5.3. Az optimalizáció

A GHS fordító az optimalizációs lépéseket az *.o* fájlok létrehozásával már elvégzi. Így az *inf* fájlokban eltárolt információk alapján megvalósítható *inline* műveleteket. E műveletek során a gyakran hívott függvények hívásainak helyére a compiler behelyettesíti a függvény gépi kódját, így ugyan a kódméret nagyobb lesz, viszont eliminálható a függvényhívásokkal járó time overhead. Kis kódméretű függvényeknél még indokoltabb a használata. A fejlesztő tehet javaslatot, hogy mely függvényeket helyettesítse be a compiler, de az felülbírálhatja a döntést [23].

Az optimalizálás további lehetősége a preprocesszálással gyakran helytelenül együtt emlegetett precompiled headerek használata. Bizonyos headereknél, melyek sok kódot tartalmaznak, sok egyéb header fájlt inkludálnak be vagy sok egyéb fájl kéri be őket, érdemes egy előzetes feldolgozást végezni (precompile) és egy köztes állapotra hozni. Az ilyen típusú header fájlok feldolgozása szignifikánsan lerövidíti a fordítási időt. A GHS fordító sajnos nem támogatja [24].

A kód hatékonyabbá tétele a linkelés alatt is megtörténhet olyan nyelvek esetén, melyek fájlról-fájlra fordítanak (így a C-nél, szemben a JAVA-val, ami egy lépésben fordítja az összes forrást). Amint minden forrásfájlból egyesével lefordult az object fájl, a compiler összeilleszti egy végrehajtható fájllá. Mivel ezen a szinten már az összeállítási információk is rendelkezésre állnak, az újonnan keletkezett fájlon már interprocedurális optimalizációs eljárásokat is lehet alkalmazni. Ezek közé tartozik a feleslegesen duplikált számítások eliminálása, a memóriahasználat optimalizálása, illetve az iteratív kódrészek (pl. ciklusok) egyszerűsítése. A GHS jelenlegi verziója ezt sem támogatja.

A GHS compilerként való használata jelenleg nem opcionális. A végrehajtható fájlok 32 bites beágyazott rendszerekre kifejlesztett mikrokontroller perifériákkal rendelkező PowerPC-n futnak, melyre jelenleg az egyetlen megbízható, tanúsítványokkal rendelkező compiler a GHS. A továbbiakban tervezett az alternatív lehetőségek

vizsgálata, azonban a manapság csúcsra járatott compilerek (Clang, GCC) open source programok, így idő és biztonságkritikus alkalmazásokban körülményes a bevezetésük.
A kód optimalizáció a TKP projectjei esetén maximálisan ki van használva (a használt mikrokontrollerek memóriáiban, a futási időkben), így nehezen elképzelhető egy olyan trade-off, mely akár a mérnökórákat hatékonyabban kihasználó egy körös fordítást támogatná a több körös fordítást igénylő optimalizációs eljárások nélkülözésével.

6. *SW build gyorsítás*

Ebben a fejezetben körbejárjuk, hogy milyen konfigurációs megoldások alkalmazhatóak a fordítási idők csökkentésére, így megteremtve a folytonos integráció lehetőségét.

6.1. *Konfigurációs lehetőségek [14]*

Vizsgáljuk meg, hogy milyen lehetőségeink vannak az aktuális fordítási eljárás optimalizálására. Jelenleg Windows 7-es környezetben zajlik a fordítás, mind a preprocesszor, mind a compiler a GHS 5.17–es verziója, melynek egyben kiadjuk a fordítási feladatot, majd az automatikusan végigmegy a build lépéseken.

6.1.1 *Áttérés más környezetre*

Az operációs rendszerek közötti különbségek kihathatnak a fordítási időkre. Az esetek többségében Windowsról Linux rendszerekre való áttérés az átlagfelhasználó számára ismeretlen rendszer hátrányai mellett általában stabilabb viselkedéssel és futási időben optimálisabb eredményekkel járhat. Ez a nyílt forráskódú fejlesztés (a kódolók valódi igényeit előtérbe helyezve, akár a felhasználói élmény rovására), illetve a Unix alapú rendszerfelépítésnek köszönhető.

Előzetes becslések alapján közel 30%-os fordítási idő csökkenésre számíthattunk Linuxos környezetben azonos fordító (GHS 5.17) használatával. A többlet a Windowsos fordítás esetében egy további POSIX átalakító (*wrapper*) könyvtár használata miatt léphet fel. A Wrapper könyvtár egy kódrétegből áll, mely átalakítja a könyvtár aktuális interfészét egy univerzális, adatformátumban kompatibilis szabványra (pl. POSIX). Ez a Linux esetében, mivel a POSIX Unix operációs rendszerek API-jának meghatározása, nem szükséges. A GHS fordító 6-os kiadásában ezt a különbséget fordító szinten kezelték, áttértek egy natív fájlrendszerre, mely által a fordítási időknek platform

függetlennek kell lenniük. Az első mérések eredményeként a Windowsos futási idő 7%-kal csökkent. Ez alapján a GHS 6 esetén a Linuxos fordítási időnek romlania kellene, ez a vizsgálat még hátra van, de valószínűleg így is megmarad a Linux előnye, tehát érdemes Linuxos fordításra áttérni.

A Linux rendszerre való áttérés nem teljesen triviális, a két környezet között kompatibilitási problémákat kell feloldani. Ilyen a Linux Case Sensitive viselkedése, a fájl elérési útjában perjelek dőlési iránya (a Windows erre sem érzékeny), a forrásfájlokban az üres inkludálások hibát adnak Linuxos fordítás esetén, illetve bizonyos előre definiált makrók is máshogy érhetők el Linux alatt. Ezen a különbségek nagy részét script programokkal (Perl nyelven) könnyen feloldhatjuk, a SW minden fájlját beolvasva reguláris kifejezésekkel Linux kompatibilissé tesszük a forráskódokat.

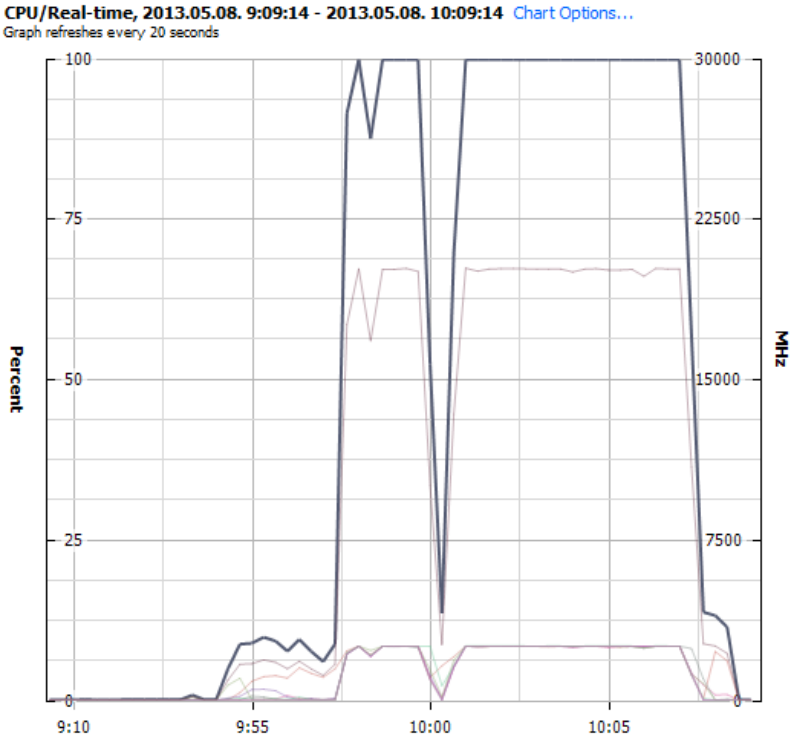
A Windowsos fordítások során bizonyos környezeti változókat batch fájlok meghívásával módosítunk. Ezek gyakorlatilag DOS parancsok egymásutánjai, így nem követik a Linuxos szintaktikát. Ezeket a módosításokat Linuxon *shellscript*ek hívásával hajthatjuk végre. Az egyes makefájlok is hívnak DOS-os parancsokat, pl. könyvtár létrehozása, törlése, melyeket szintén Linux parancsokra cserélünk le. Ezenkívül természetesen módosítani kell a fordítás futtatását vezérlő makefájlokban a SW és a fordítóprogram elérési útját.

Miután eljutottam odáig, hogy egy szoftverben minden módosítást alkalmaztam a sikeres Linux szerveren történő futtatáshoz, automatizáltam a folyamatot, és Jenkinsben egy jobot létrehozva vezéreltem a fordítást. Ebben először kikértem a verziókezelőből a szoftvert, majd a korábban fejlesztett, kompatibilitást megvalósító Perl scriptek, és a batch műveleteket helyettesítő shell scriptek meghívása után rendelkezésre állt a Linuxon is fordítható SW. Ezután a már szokott módon hívtam meg a fordítás utasításait (depend, firstpass, secondpass).

A SW sikeres lefordulása után össze kell vetni, hogy a Linuxos fordítás eredményeként előálló *hex*, és *map* fájlok megegyeznek-e a Windowsos referenciával. Ezekben mindössze egy-két helyen adódtak sorrendi különbségek, melyet a makefájlok további apró módosításával orvosolni lehetett.

Ezután egy releváns mérést kellett előállítani, hogy a korábbi 30%-os becsléshez képest mennyi a valódi megtakarítás a fordítási időben. Ehhez azonos feltételeket kellett teremteni a két környezetben való fordításhoz. Legegyszerűbben úgy valósítható meg, ha a két virtuális gép ugyanazon a szerveren üzemel, más feladattal nem terheljük az

adott szervert és egymás után futtatjuk a két fordítást. A szerver egy 8 magos gép volt, melynek a maximális processzor frekvenciát engedélyeztük. Ezen először 8, majd 12 és 16 szálon is vizsgáltuk a futási időket. Fontos, hogy a fordítások alatt mindig álljon rendelkezésre a fordítóhoz licenc, így ezt érdemes volt munkaidő után végezni. Azt, hogy a szerver kihasználtsága a fordítás ideje alatt teljes volt-e, egy real-time loggerrel vizsgáltuk. A 6.1.1. ábrán látható módon 100%-osan kihasználta a kivezérelt frekvenciát a CPU, a rövid letörés az egy szálon feldolgozott *inf* fájlok miatt tapasztalható.



6.1.1. ábra : A szerver kihasználtsága a Linuxos fordítás ideje alatt [saját forrás]

A legrelevánsabb mérésnek a 12 szálon futtatott fordítást választottuk, melynél végül 82%-ra esett vissza a fordítási idő. Ez a 8. ábra alján látható #114-es Linuxos és #40-es Windowsos Jenkins jobok futási időinek összevetése.

						Sw			BMW_17.10.1.3										Sw	BMW_17.10.1.3	
		Nr of threads																			
			Sw checkout start	Sw checkout end	Sw checkout	Scripts start	Scripts end	Scripts	Depend start	Depend end	Depend	FirstPass start	FirstPass end	FirstPass	SecondPass start	SecondPass end	SecondPass	Total compilation	Total
Linux	#104	12	18:18:36	18:20:45	0:02:09	18:20:45	18:20:53	0:00:08	18:20:53	18:21:38	0:00:45	18:21:38	18:23:41	0:02:03	18:23:41	18:32:01	0:08:20	0:11:08	0:13:25
Win7	#32	12	0:43:11	0:49:17	0:06:06	0:49:17	0:49:48	0:00:31	0:49:48	0:52:05	0:02:17	0:52:05	0:57:01	0:04:56	0:57:19	1:08:51	0:11:32	0:18:45	0:25:22
					0,352459016			0,258065			0,328467			0,415541			0,72254335	0,593777778	0,528909
Linux	#105	16	23:44:29	23:46:53	0:02:24	23:46:53	23:47:01	0:00:08	23:47:01	23:47:47	0:00:46	23:47:47	23:49:50	0:02:03	23:49:50	23:58:38	0:08:48	0:11:37	0:14:09
Win7	#35	16			##########	1:13:31	1:13:49	0:00:18	1:13:49	1:16:08	0:02:19	1:16:08	1:21:04	0:04:56	1:21:23	1:33:24	0:12:01	0:19:16	#######
								0,444444			0,330935			0,415541			0,73231623	0,602941176	-0,26228
Linux	#106	13	1:13:29	1:15:48	0:02:19	1:15:48	1:15:57	0:00:09	1:15:57	1:16:43	0:00:46	1:16:43	1:18:46	0:02:03	1:18:46	1:27:12	0:08:26	0:11:15	0:13:43
					0,352459016			0,258065			0,328467			0,415541			0,72254335	0,593777778	0,528909
Linux	#112	12	7:33:44	7:36:14	0:02:30	7:36:14	7:36:19	0:00:05	7:36:19	7:37:04	0:00:45	7:37:04	7:39:04	0:02:00	7:39:04	7:47:16	0:08:12	0:10:57	0:13:32
Win7	#36	12	21:41:44	21:46:36	0:04:52	21:46:36	21:47:02	0:00:26	21:47:02	21:48:40	0:01:38	21:48:40	21:52:01	0:03:21	21:52:16	22:00:57	0:08:41	0:13:40	0:18:58
					0,51369863			0,192308			0,459184			0,597015			0,94433781	0,801219512	0,713533
Linux	#113	12	7:53:21	7:56:00	0:02:39	7:56:00	7:56:06	0:00:06	7:56:06	7:56:50	0:00:44	7:56:50	7:58:50	0:02:00	7:58:50	8:07:01	0:08:11	0:10:55	0:13:40
Win7	#37	12	6:49:23	6:53:06	0:03:43	6:53:06	6:53:32	0:00:26	6:53:32	6:55:09	0:01:37	6:55:09	6:58:31	0:03:22	6:58:46	7:07:28	0:08:42	0:13:41	0:17:50
					0,713004484			0,230769			0,453608			0,594059			0,94061303	0,797807552	0,766355
Linux	#114	12	10:05:17	10:08:26	0:03:09	10:08:26	10:08:32	0:00:06	10:08:32	10:09:16	0:00:44	10:09:16	10:11:16	0:02:00	10:11:16	10:19:29	0:08:13	0:10:57	0:14:12
Win7	#39	12	10:52:09	10:55:56	0:03:47	10:55:56	10:56:22	0:00:26	10:56:22	10:57:58	0:01:36	10:57:58	11:08:00	0:10:02	11:08:14	11:16:51	0:08:37	0:20:15	0:24:28
					0,832599119			0,230769			0,458333			0,199336			0,95357834	0,540740741	0,580381
Linux	#114	12	10:05:17	10:08:26	0:03:09	10:08:26	10:08:32	0:00:06	10:08:32	10:09:16	0:00:44	10:09:16	10:11:16	0:02:00	10:11:16	10:19:29	0:08:13	0:10:57	0:14:12
Win7	#40	12	10:37:11	10:40:48	0:03:37	10:40:48	10:41:02	0:00:14	10:41:02	10:42:40	0:01:38	10:42:40	10:45:56	0:03:16	10:46:11	10:54:35	0:08:24	0:13:18	0:17:09
					0,870967742			0,428571			0,44898			0,612245			0,9781746	0,823308271	0,827988

6.1.2. ábra : A fordítási idők összevetése Windows és Linux környezetben [saját forrás]

Az eredményekből látható, hogy a legnagyobb nyereség a scriptek futtatása, a függőségi vizsgálatok, illetve az első körös futtás alatt jelentkezett (akár 55%-os csökkenés). A második körös fordítás alatt azonban minimális volt a különbség (2%), és mivel a fordítási idő 72%-át ez a folyamat teszi ki, így a csökkenés 18%-ra zsugorodott.

A folytatásban a cél, hogy ugyanazon scriptek futtatásával álljon elő a már valóban minden környezettel kompatibilis SW. Ez a makefájlokban környezetfüggő változók vizsgálatával oldható meg. A script beilleszti ezt a *flag*et a globális változók közé, a `$^O` környezeti változó szerint behelyettesíti az értékét (`WINDOWS` vagy `LINUX`), majd az összes korábban elvégzett módosítás elé beilleszt egy feltételvizsgálatot, hogy szükség van-e arra a módosításra, vagy olyan környezetben vagyunk, ahol fordítható az eredeti SW.

6.1.2. C fájlok csoportosítása

Az alábbiakban taglalt gyorsítási eljárások már teszt jelleggel bekerültek egyes projektek fejlesztési folyamataiba. Mivel ezek is a build gyorsítást segítik, érdemes megismerni működésüket.

Az AUTOSAR RTE (Run-Time Environment) header fájljai minden forrás fájlban inkludálva vannak. Ezek minden *.c* fájl fordításánál nyelvi elemzésen (parsing) esnek át. Ezen fájlok esetén tehát jelentősen rövidülhetne a fordítási idő, ha a GHS fordító támogatná a precompiled header fájlok beolvasását. Azonban elérhetünk egy hasonló viselkedést a *.c* fájlok csoportosításával.

Fájlokat csoportosítunk egy *group_xx.c* fájlba, majd ezt a fájlt fordítjuk. Így a közösen ikludált headereket csak ennél az egy fájlnál, egyszer kell feldolgozni. A header guardok megakadályozzák a többszörös inkludálást. A fájlokat fordító konfiguráció vagy linker template-ben található fájlcsoportok alapján (EPAS, BSW, SWE2, SWE3) csoportosíthatjuk. A tesztelés során kiderült, hogy a fordító teljesítménye romlik, és a fordítási idő exponenciálisan nő a fájl méretével és a komplexitással (valószínűleg a keresési algoritmusok nem megfelelő implementálása miatt), így a maximális fájlméretet korlátozni kell.

A headerek interfész információi (private, public) automatikusan a fordító által generálódnak a különböző fordítási egységekhez az előfordítási beállításoknak megfelelően. A *group_xx.c* fájl csak egyféle interfésszel tud forrás fájlt eltárolni (a header egyszer lesz inkludálva). Tehát figyelembe kell venni az ilyen headereket a csoportosítás megvalósítása során.

A globális változóknak egy csoportban egyedi névvel kell szerepelniük (a global static változók minden, a csoportban szereplő fájlból elérhetőek, nem csak abban a forrásban, ahol deklarálták őket).

Először létrehozzuk a *group_xx.c* fájlokat, majd legeneráljuk a linker leírót, mely már a megfelelő linker sectionökben tartalmazza *group_xx.o* bejegyzéseket. Ezután a megszokott módon kezdhetjük a fordítást.

Ezzel az eljárással a becslések szerint 50%-ot takaríthatunk meg a fordítási időből. Más preprocesszor ugyan elő tud állítani precompiled headereket, de amíg a compiler (GHS) nem támogatja, a preprocesszor ezen tulajdonságát nem tudjuk kihasználni.

6.1.3. Gyorsítás más preprocesszor használatával

A GHS nem a legkiélezettebb és a leggyorsabb fordító, a GCC a preprocesszálást kétszer, a Clang négyszer gyorsabban hajtja végre. Azonban nem triviális a GHS preprocesszort lecserélni. A GHS közel 60 előredefiniált makrót tartalmaz compiler beállításoktól függően. Ahhoz, hogy a Clang használni tudja a GHS standard makróit, a GHS compiler manuáljában definiált makrókat parancssorból adjuk át a Clangnak, melyhez előtte letiltjuk a Clang saját előredefiniált makróit.

A preprocesszornak emellett a # kezdetű sorokkal tudunk utasításokat adni. Ebben is vannak compiler specifikus elemek, pl. a *#pragma* egy speciális GHS specifikus preprocesszor utasítás. Ezt a Clang nem ismeri, így figyelmen kívül hagyja. Ebből és egyéb különbségekből fakadóan a preprocesszálás kimenete sem lesz azonos GHS és Clang esetén. Mivel a GHS compiler egyelőre rendíthetetlen, a Clang preprocesszor kimenetét kell a GHS kimenetével azonossá tenni.
Ilyen különbségek a Clang esetén *whitespace* karakterek hozzáadása, a bent maradt *pragma* utasítások miatt a sorok száma elcsúszott, illetve a GHS specifikus makrók pl. __LINE__ máshogy fejti ki az adott sor számát. Ez nem probléma a compiler szempontjából, csak formai különbségek, illetve olyan funkciókat érintenek, melyek elhagyhatóak a sikeres fordítás szempontjából (pl. line információk nem állnak rendelkezésre). Ezzel az eljárással is közel 50% takarítható meg.

6.1.4. Gyorsítás szeparáltan preprocesszált fájlokkal

A GHS esetén preprocesszor és a compiler interfésze nem optimális. Így ha nem automatikusan adjuk ki a build utasítást a GHS-nek, hanem külön állítjuk elő a preprocesszált fájlokat és ezt adjuk tovább a compilernek, projekttől függően 20-40% különbség léphet fel fordítási időben. Fordítási lépések módosítása: preprocesszált fájlok előállítása (az object fájlok könyvtárába generálódnak *.i* kiterjesztéssel), majd az előfeldolgozott fájlok fordítása.
Legnagyobb meglepetésre az automatikusan, illetve szeparáltan lefordított szoftverek kimeneti binárisai nem egyeztek. Erre a GHS támogatása egy kapcsoló átállítását javasolta, mely megoldja a problémát, de nem állít elő #line sorokat, mellyel hiba esetén a debuggolás szinte lehetetlen. Nincs információ a hiba forrásfájlban lévő helyéről, csak a preprocesszált fájl valahányadik sorában talált hiba a rendelkezésünkre álló információ.

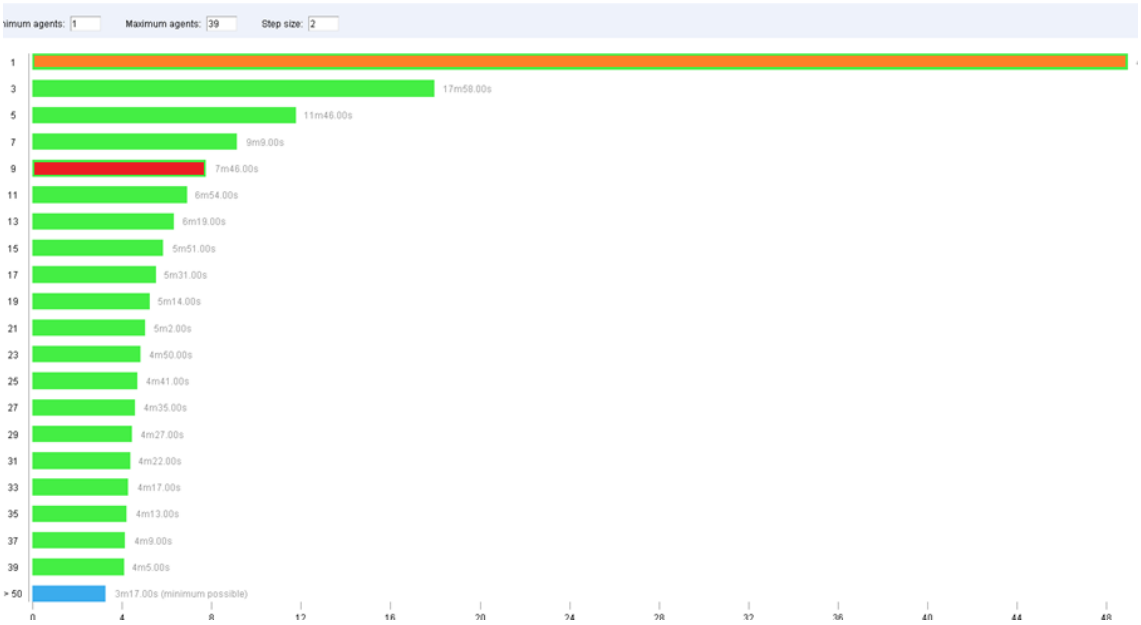
Ezen eljárások együttes alkalmazása természetesen nem fogja ortogonálisan összesen 130%-kal rövidíteni a fordítási időt, de mivel a fordítások különböző területein optimalizálunk, remélhetőleg minél kevesebb átfedés lesz az egyes gyorsítási eljárások hatásaiban.

6.2. *Build optimalizáló cloud megoldások*

Érdemes megvizsgálni a jelenleg a piacon fellelhető build optimalizáló cloud szoftvereket. Ezek a céljainknak megfelelő funkcionalitással bírnak, integrálhatóak verziókezelő szoftverekbe, egy központi szervert elérve fordíthatjuk szoftvereinket, és ami a valódi fegyvertényük, hogy speciális build optimalizációs eljárásokat alkalmaznak, mellyel radikálisan csökkenthető a fordítási idő.

Ilyen szempontok alapján vizsgáltam meg az Electic Cloud cég termékeit, ezen belül az elsősorban alacsonyszintű fordítás optimalizációra készített ElectricAccelerator (EA) toolt.

A hatékonyabb fordítás lelke, mint cloud alkalmazás, a több agent gépen, párhuzamosan végrehajtott fordítás. Ehhez az EA szofisztikált eljárásokat alkalmaz a függőségek és a skálázhatóság feltérképezésére, ezáltal sokkal hatékonyabb párhuzamos futtatásra alkalmas, mint a korábbi hasonló SW toolok. Természetesen a párhuzamosítás eredményessége függ az agent gépek számától. A 6.2.1. ábrán látható egy, a cég honlapján fellelhető statisztika az agent gépek számának a fordítási időre gyakorolt hatásáról. A narancssárgával kiemelt vonal a szekvenciális, egy szálon futtatott fordítás idejét mutatja, mely az ábra szerint közel 49 percig tart. Az agent gépek számának növekedésével exponenciális jellegű fordítási idő csökkenést tapasztalhatunk. A TKP-nál jelenleg használható fordító licencek száma 9, így ennyi agent gép használatával 8 perc alá csökkenthető a fordítási idő (az ábrán piros vonal). Kérdés, hogy lehet-e ugyanazzal a licenccel több agent gépen, vagy egy licenccel rendelkező gépen futtatott több virtuális gépen fordítani az EA párhuzamosított feladatait. Ez esetben érdekes lehet az utolsó (kék) adat, mely 50-nél több szálon párhuzamosított fordításra alig több mint 3 percet igényel.



6.2.1. ábra : A fordítások időtartama az agent gépek számának függvényében Electric Accelerator használatával [25]

Emellett további előnyökkel is járhat az EA alkalmazása. A régebbi kiadású, hasonló programok esetében a párhuzamos végrehajtást végző node-ok számának növekedésével egyre több lett a hibás fordítás. Az EA automatikus konfliktus vizsgálata és korrigáló eljárásai eltárolják, hogy mely fájlok voltak szükségesek az egyes objektumok, könyvtárak vagy végrehajtható fájlok létrejöttéhez. Amennyiben a függőségi sorrend a párhuzamos végrehajtás során felborul, az EA automatikusan újra végrehajtja azt a megfelelő sorrendben. A függőségek vizsgálatával megbízható inkrementális fordítások végezhetőek, mely során nem fordul újra a teljes SW, csak a módosított kódrészletektől függő komponensek [25].

Ez egy tisztán szoftveres optimalizációs technológia, melynél az agent gépeknek kisebb HW igényük van (a cluster manager gép 2 GB RAM, az agent gépeknek 1 GB RAM és HDD-ből a lefordított SW méretének háromszorosa). A TKP által használt jelenlegi szerverek esetében ennél több erőforrás áll rendelkezésre, melyet virtuális gépek és a többszálú fordítás tesz hatékonyá (8 magos gépek, 12 szálon fordítva hasonló párhuzamosított viselkedésre képesek, és ezalatt csak 1 licencet használnak). Ha a gyorsító eljárások alkalmazásával pl. a BMW-s projekt esetében 10 percre le tudjuk csökkenteni a jelenlegi 30 perces fordítási időt, akkor a hamarosan rendelkezésre álló 10 licenccel 10 perc alatt 10 SW fordulhat le. A EA alkalmazásával 10 agentet használva kb. az 1/7-ére eshet vissza a fordítási idő, mely egy fordítás esetén ideális (~4 perc),

azonban ha csúcsidőben 10 fordítást akarunk egyszerre elindítani, ennek lefutásához az utolsó fordításnak már 40 percet kell várnia. További aspektusok lehetnek, ha több szerveren fordítunk EA használatával, és kevesebb licencet használunk, mellyel megkereshető az ideális egyensúly a párhuzamosan fordítandó szoftverek száma és az ehhez használt szerverek száma között.

6.3. Tervek

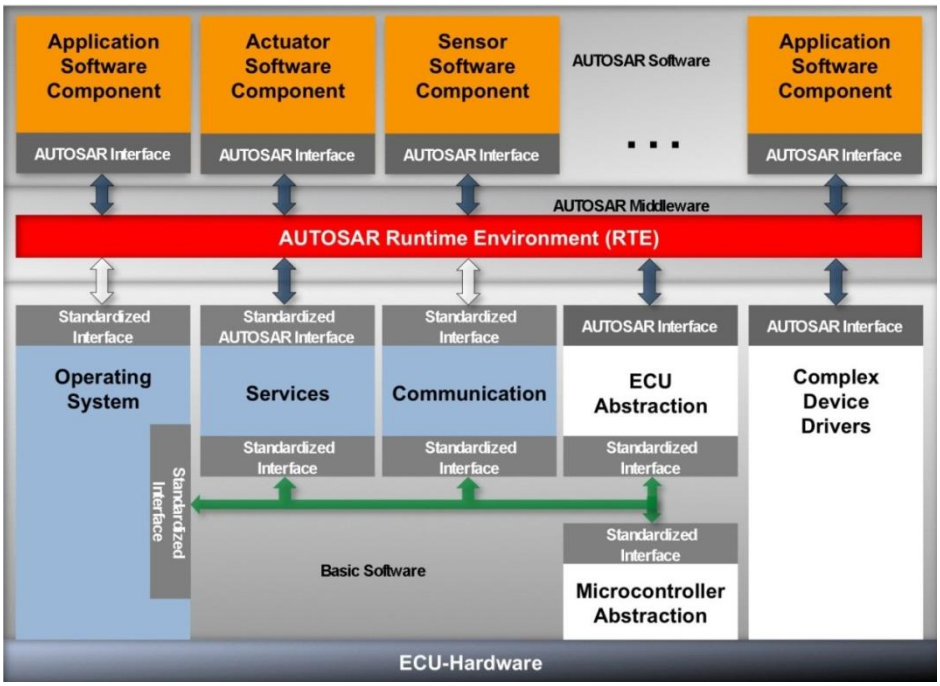
A fent említett eljárások jelenleg csak egyes projekteken, teszt jelleggel működnek, azonban kecsegtető eredményeket szolgáltatnak a fordítási idők csökkentésére, így a jövőben minden projektre érdemes bevezetni. Megvizsgálva együttes hatásukat meg kell keresni az ideális egyensúlyt a ráfordított munka, az esetleges új fordító, vagy a cloud alkalmazások licenceinek költsége és a hatékonyság között. Ezeket az eljárásokat automatizáltan alkalmazhatóvá kell tenni a fordítások során, így a jelenlegi Build Server Toolban lehetővé tenni pl. a *.c* fájlok csoportosításának lehetőségét. Emellett el kell érni, hogy ez a tool megfelelően széleskörű és felhasználóbarát legyen ahhoz, hogy minden projekten alkalmazható legyen és így az egész cég élvezhesse a gyors fordítások és ezáltal a folytonos integráció előnyeit.

7. AUTOSAR szimulációs tool fejlesztése

Az ajánlást bemutató meetingen a csoportvezetők érdeklődését leginkább a HW függő rétegeket elfedő szimulációs tool saját implementációja keltette fel. Ennek első körben egy egyszerű, csak bizonyos modulokat lefedő, a megvalósítás lehetőségét alátámasztó verziójának megvalósítását tűztük ki célul. Ehhez először az AUTOSAR architektúrát vizsgáltam meg közelebbről.

7.1. AUTOSAR [26]

Az AUTOSAR szabvány réteges felépítést határoz meg, hogy a kívánt működést függetlenítse a támogató HW és SW szolgáltatásoktól. Alapvetően az alkalmazás logikát megvalósító applikációs réteg, a HW szintű modulokat leíró Basic SW (BSW) réteg és a kettő közötti kommunikációt megvalósító Runtime Environment (RTE) rétegekre osztható fel. Az AUTOSAR-os terminológiában a modulok a BSW részei, és a komponensek az alkalmazás réteg elemei. Az AUTOSAR rétegek felépítése a 7.1.1. ábrán látható.



7.1.1. ábra : Az AUTOSAR rétegeinek struktúrája [27]

7.1.1. Alkalmazás réteg

Alkalmazás szintű SW komponensek építik fel, melyek tartalmazzák a beágyazott rendszer funkcionalitását, és használják a BSW szolgáltatásait. A komponensek kommunikációját szabványosított portok biztosítják. Ezek között az összeköttetést egy virtuális hálózat, a *Virtual Function Bus* (VFB) valósítja meg. A komponensek kimenetét továbbítja a megfelelő inputra, így nem kell applikáció szinten foglalkozni a kommunikáció megvalósításával, illetve az implementáció előtt is ellenőrizhető a komponensek közti kommunikáció. Ez a szabványosított interfészeknek köszönhető, melyek a be- és kimeneti portok mellett az adatformátumot is meghatározzák.

7.1.2. Runtime Environment

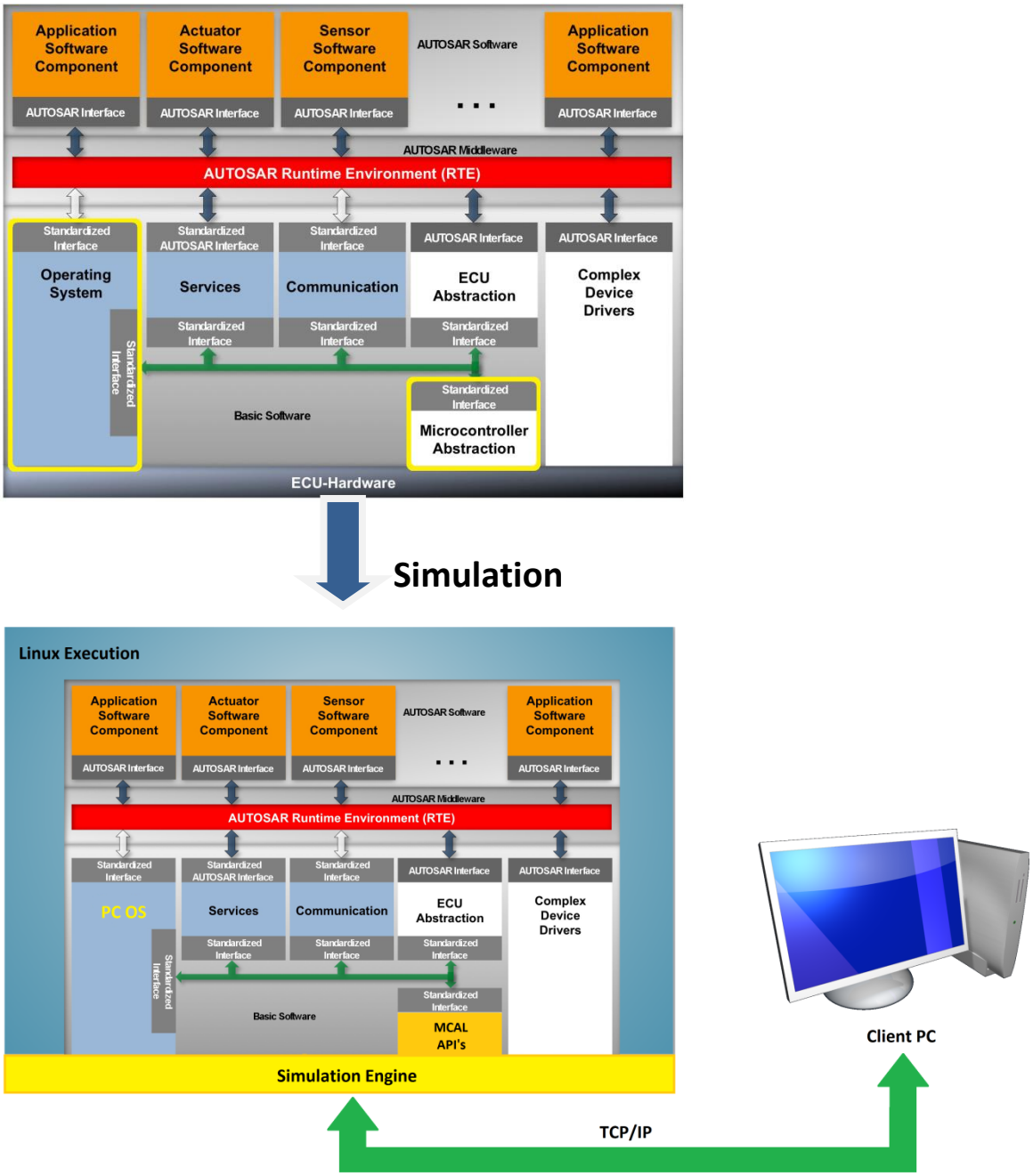
A futásidejű környezetnek (RTE) alapvetően két fontos feladata van: megvalósítani a VFB-t, illetve kommunikálni az OS-sel a taszkok időzítésével kapcsolatban. A VFB valósítja meg a kommunikációt az összes modul és komponens között, az OS kommunikációt pedig a Basic SW Scheduler modul biztosítja.
Az RTE réteg tartalmaz minden belső összeköttetést, kommunikációs és adminisztratív részét a szoftvernek. Az RTE egyfajta belső SW busz, mely kapcsolatban áll minden modullal és komponenssel, ezáltal nem megfelelő konfigurációja esetén lehet, hogy a rendszer egyáltalán nem fog működni. Előállít egy interfészt az applikációs réteghez, ezáltal összeköti a BSW modulokkal. Ezeknek a moduloknak az interfészei az RTE Generátor segítségével jönnek létre miután megadtuk neki a rendszer konfigurációját és interfészeit. A rendszer konfigurációját szabványosított XML fájlokban adjuk meg, melyhez egy AUTOSAR-os hitelesítő sémafájlt használunk, így a végső formátum az ARXML. A generált ECU konfiguráció egyik nagy előnye, hogy a generátor scriptek optimális konfigurációt tudnak megvalósítani, ha a megfelelő szabályok definiálva vannak. A generátor konfigurációjára is szükség van az implementációs eljárások meghatározására az adott ECU HW kapacitásainak függvényében. A generálási folyamat másik előnye, hogy ezáltal a teljes rendszer konfigurációja definiálva van a leírásban. Ebből a szempontból az ECU egy alkalmazása ennek. A konfigurációs fájlok határozzák meg a rendszert, hiszen a HW már adott.

7.1.3. Basic SW réteg

Az architektúrában az ECU HW felett álló réteg, mely nem valósít meg konkrét funkciót, de az API-k segítségével HW független felületet biztosít a feljebb lévő rétegeknek. A környezet működését garantálja, platformot ad az alkalmazás rétegben megvalósított ECU funkcionalitásokhoz.
A BSW-t több kisebb rétegre oszthatjuk. Ezek közül a legfelső a szolgáltatási réteg, mely közvetlen kapcsolatban áll az alkalmazási réteggel az RTE-n keresztül. Itt találhatóak a kommunikációért felelős modulok, memóriakezelés, diagnosztika, illetve az OS is itt helyezkedik el.
Alatta helyezkedik el az ECU absztrakciós réteg, mely perifériák és külső eszközök érhetőek el, és teljesen ECU független felületet biztosít.
A BSW legalsó rétege az MCAL (Microcontroller Abstraction Layer), mely mikrovezérlő független elérést biztosít a kontroller belső perifériáihoz. Itt helyezkednek el azok az API-k, melyek átkonfigurálása nélkülözhetetlen egy HW független szimulációs eszköz létrehozásához (így a DIO API, mely a digitális I/O műveleteket valósítja meg).
Emellett megtalálható egy speciális funkciókat kiszolgáló alréteg, mely közvetlen kapcsolatot biztosít az RTE és a HW között. A komplex meghajtók (Complex Device Drivers) a szabvány által nem támogatott HW elemek kezelését, illetve speciális, időkritikus folyamatok megvalósítását teszi lehetővé.

7.2. A szimuláció koncepciója

A megbeszélés eredményeként egy olyan tool létrehozása a feladat, mely a PC-n futtatható AUTOSAR-os ECU projekteknek biztosít HW oldali jeleket, adatokat. A klasszikus AUTOSAR struktúra az alábbi ábrának megfelelően módosul.



7.2.1. ábra : Az AUTOSAR szimuláció [27]

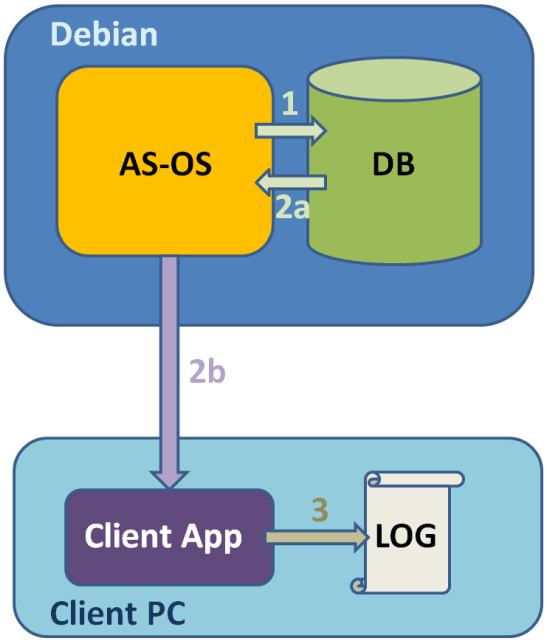
A könnyebb érthetőség és a fejlesztési sorrendnek megfelelően a kliens oldali PC felől mutatom be az architektúrát. Először létrehozunk egy TCP/IP kliens-szerver alkalmazást, értelemszerűen a kliens alkalmazás fut majd a kliens PC-n, a szerver pedig a PC-n futtatott ECU SW projekt Linux szerverén.

Tehát kommunikációt szeretnénk kezdeményezni a kliens alkalmazás és a szerveren futó AUTOSAR ECU projekt között. Ehhez a szerver gépen egy szimulációs engine

alkalmazás fog futni, mely egyrészről tartalmazza a TCP/IP kommunikáció szerver funkcióját, másrészt egyfajta interfészt valósít meg a TCP kommunikáció (így a kliens gép) és a futó AUTOSAR-os ECU projekt között. Ez az AUTOSAR MCAL API-jainak módosításával valósulhat meg, mivel ez az interfész az AUTOSAR felsőbb, HW független rétegei és az általános használat során a vezérlést megvalósító ECU-k között. Tehát amennyiben a felsőbb applikációs komponensekben megvalósított funkcionalitásnak pl. a HW perifériáiról szüksége lenne adatokra, ezen az MCAL rétegen keresztül kezdeményezi a kommunikációt. Amint egy ilyen kérés érkezik, a módosított MCAL API a párhuzamosan futó szimulációs engine alkalmazásba hív át. Az továbbítja a kérést a kliens alkalmazás felé, ahol az adott konfigurációnak megfelelően átküldjük az adatokat a szerver oldalnak. Ezután a szimulációs alkalmazás hív át a futó AUTOSAR alkalmazásba, mely az MCAL API interfészein keresztül, a normális működésnek megfelelően továbbítja az adatokat a felsőbb rétegek felé. A kliens az adatokat egy adatbázisban tárolja, egy-egy periféria írás-olvasás mind egy adatbázis lekérdezésnek felel meg.

7.2.1. A szimuláció architektúrája

Először a működés architektúráját kell tisztázni. A szerver és a kliens gép külön hoszton futnak, viszont az adatbázis-szerver helye megválasztható. Egyik lehetőség, hogy az AUTOSAR OS szerverrel egy helyen fut. Ekkor jobban elkülönül a kliens alkalmazás és a többi (szerver) funkció. Mivel ugyanazon a szerveren fut az AUTOSAR OS és az adatbázis, melyből információkat akar lekérdezni, jogos a felvetés: miért nem közvetlenül kommunikálnak? Tehát az adatbázis lekérdezések is az AUTOSAR OS taszkjai között lennének megvalósítva, és a kliens szerepe leredukálódik a monitorozás szintjére. Minden vezérlési és konfigurációs feladatot is a taszkok között kellene lehetővé tenni, egy-egy utasítást lekérdezve a klienstől. Ez a megoldás a gyorsaság szempontjából talán a legideálisabb. Egy adat lekérdezése mindössze 2 lépésből áll, emellett a kliens értesítése a meghívott függvényekről várakozási időben is megtörténhet.

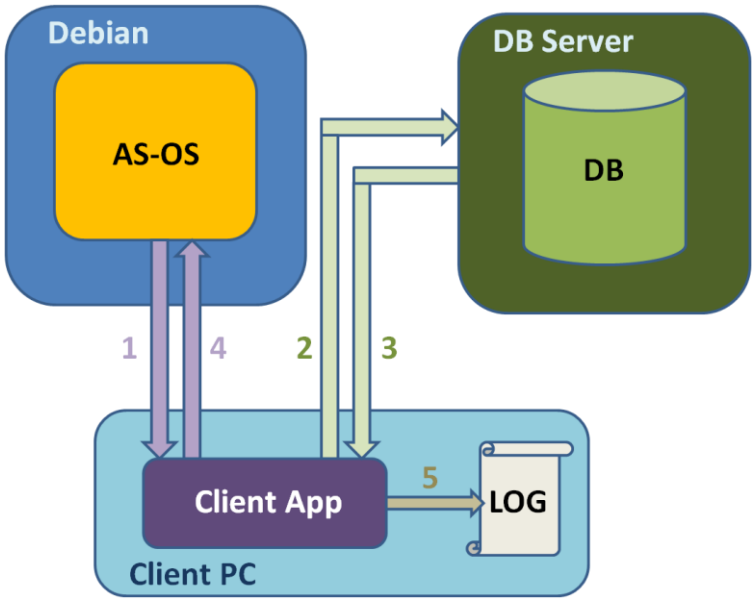


7.2.2. ábra : A gyorsaságra optimális szimuláció architektúra [saját forrás]

A sebesség azonban nem elsődleges szempont, főleg a tool fejlesztésének kezdeti szakaszában. A későbbiekben, amikor valóban a teljes SW ebben a környezetben fog futni, már lesz jelentősége a futási idők csökkentésének, hogy a folytonos integráció elveit betartva, ne tartson túl sok ideig a tesztelés. A meghozandó kompromisszumok azonban gyakorlatilag kizárják ezt az architektúrát. A vezérlés és minden konfiguráció megvalósítása az MCAL modulokban, és az AUTOSAR OS taszkjaiban nem összeegyeztethető a feladat specifikációjával. A tool használata során napi rendszerességgel jönnek új, módosított SW verziók, melyeket minimális módosításokkal alkalmassá kell tenni a PC-n való futtatásra, tesztelésre. Az MCAL API-k módosításában ennek ki kell merülnie, és minél kevesebb funkcionális módosításnak szabad csak történnie az ECU-n futó verzióhoz képest, egyéb esetben a mért futási paraméterek közel sem lesznek valósághűek. Továbbá fontos szempont a mérések könnyű konfigurálhatósága, melyet minden esetben az OS taszkokban kellene megvalósítani. A tool széleskörű alkalmazása során nem várható el, hogy minden felhasználónak le kelljen mennie az ütemezés szintjére, ha valamit módosítani akar.

Sokkal ésszerűbb, és esetünkben a legtöbb szempontból ideálisabb, ha ezek mind kliens szinten konfigurálhatóak. Az AUTOSAR OS szintjén csak a kérések átadását szabad lebonyolítani, azt is minél kevesebb ráfordítással. Tehát az adatbázis lekérdezések is a kliensben találhatóak, ezért nem szükséges, hogy az AUTOSAR OS és az adatbázis

szerver egy helyen fussanak. Az adatbázis egy központi szerveren fut, és minden adatforgalom a kliensen keresztül zajlik. Emiatt egy port lekérdezése jóval több lépésből áll. Az AUTOSAR OS a TCP/IP socketen keresztül átküldi a függvény nevét és az argumentumot a kliensnek. Ezután a kliens az adatbázistól kérdezi le az adott porthoz tartozó értéket. Miután megkapta az adatot, a socketen visszaüzeni a szervernek az értéket. Ez tehát két TCP/IP üzenettel hosszabb, mint az első architektúra esetén. A fájlba loggolás nem várakoztatja az AUTOSAR OS futását egyik esetben sem. Előny a második architektúra esetén, hogy mindegyik résztvevő függetlenül futhat, illetve, hogy a kliens gép erőforrásai is jobban kihasználhatóak a későbbi futás gyorsítás érdekében.



7.2.3. ábra : A kliensből konfigurálható szimuláció architektúra [saját forrás]

7.3. *Az Autosar OS bemutatása*

A hardver nélküli futtatáshoz szükségünk van az AUTOSAR-os ECU-n futó operációs rendszer specifikációs modul átkonfigurálására PC-n futtatható környezetre. Ez gyakorlatilag az ütemezés szimulációját jelenti, az applikációnak úgy kell érzékelni a jeleket, mintha az AUTOSAR OS ütemezője küldené.

A ThyssenKrupp Presta AUTOSAR fejlesztéssel foglalkozó osztályán merült fel egy, az AUTOSAR követelményeinek megfelelő saját OS megvalósításának ötlete. Ezáltal mélyebben optimalizálhatóak a taszkok ütemezései a cég igényeinek megfelelően,

emellett függetlenedni lehet a licencelt szolgáltatásoktól, így a support belsőleg is megoldható.

7.3.1. Az OS ütemezése

A taszkok ütemezése többféleképpen is megvalósulhat. Az ECU-kon futó SW külső hardveres interruptok hatására ütemezi a taszkokat, ez ilyen formában nem alkalmazható a PC-s futtatás esetén. Legegyszerűbb egy ütemező listát implementálunk, mely fix prioritással tartalmazza a taszkok sorrendjét. Az AUTOSAR támogat egy nehézsúlyú, több konfigurációs lehetőséggel rendelkező ütemezőt, mely ennek megfelelően viszonylag nehezen implementálható, funkcióinak azonban csak töredékét használnánk ki az alkalmazásunkban. Ezért elegendő egy könnyűsúlyú ütemező lista, mely tartalmazza az R8-as projektben futó taszkokat sorrendben, a megfelelő prioritással.

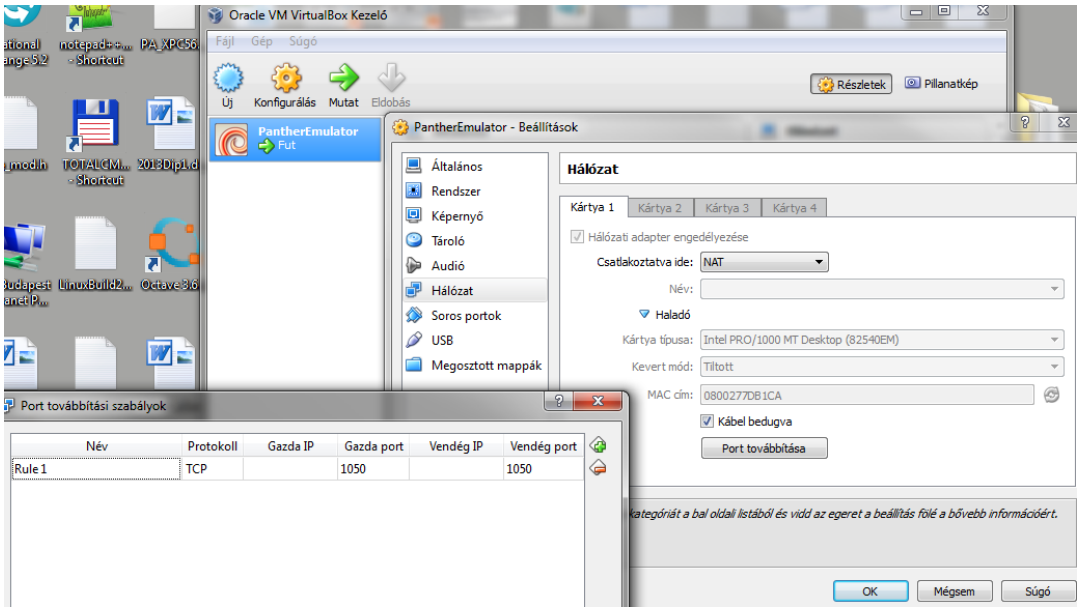
A leegyszerűsített, PC-n futtatott projektben található main függvény a `StartOS(OOAMI_EPSAPPMODE)` paranccsal indítja az operációs rendszert. Ez egy inicializáló *Startuphook* taszkban meghívott függvények egyszeri lefutása után az *Idle* taszkban folyamatosan triggerel egy megszakítást, ami a könnyűsúlyú ütemező szerint hívja a taszkokat. A rendszer funkcióit itt implementáljuk; pl. ha egy TCP/IP réteget definiálnánk az ECU projektben, akkor a *Startuphook* taszkban hozzuk létre a socketet, a kommunikációt pedig az ütemezésnek megfelelő taszkokban bonyolítjuk. PC-s futtatás esetén leállítható és újraindítható az ütemezés, ellentétben az ECU-kon futó beágyazott alkalmazásnál.

7.3.2. A fejlesztő környezet

Ez az OS egy virtuális gépen keresztül érhető el, a forrásában is ezen a felületen módosíthatunk, fordíthatunk, majd futtathatjuk. A virtuális gép operációs rendszere Debian, melyet az Oracle VirtualBox programjával indíthatjuk a host PC-n. Ehhez tartozik egy közel 4.5 GB-os fájl, melyet a VirtualBox telepítése után a megfelelő beállítások mellett elindíthatunk. Ez az egy fájl teljes egészében tartalmazza a virtuális

gép minden paraméterét, tartalmát, beállításait. Ennek segítségével egy jól beállított AUTOSAR szervert könnyen duplikálhatunk további gépeken.

Egy fontos konfigurációs beállítás szükséges a socket kommunikáció helyes működéséhez. Ez a port engedélyezése, melyet a VirtualBox hálózati beállításai közt kell keresni. Az itt megadott porton keresztül tud majd a kliens socket kapcsolatot kezdeményezni. A csatlakoztatás NAT (Network Address Translation) módját kiválasztva a TCP/IP csatlakozás során a host PC IP-címén érhetjük el az AUTOSAR OS-t. A host a korábban megadott szabad porton továbbítja a kéréseket a virtuális gép felé.



7.3.1. ábra : Az Oracle VirtualBox hálózati beállításai [saját forrás]

7.4. Kliens-Szerver kommunikáció

Első lépésként azt a kommunikációt valósítjuk meg, amivel a PC-n futó AUTOSAR-os projekteket távolról, egy kliens PC-vel tudjuk konfigurálni, adatokat lekérdezni, teszteket futtatni és ellenőrizni. Ezt egy TCP/IP kliens - szerver alkalmazással oldjuk meg.

Hasonlóan a szimuláció további módosításaihoz, itt sem kell magas szinten belenyúlni az AUTOSAR architektúrába. Az AUTOSAR tartalmaz *stack*eket a socketekkel megvalósítható adatforgalomra, a 4.1-es verzióban konkrét TCP/IP stack is

hozzáférhető, ez azonban elsősorban az ECU-k közötti kommunikáció támogatására hivatott, illetve HIL tesztek esetén diagnosztikai célokra, mely az Ethernet hálózat magasabb átviteli sebességét használja ki (Ethernet: 10-100Mbit/sec, CAN: 500Kbit/sec) [28].

7.4.1. Berkeley socket API

Alacsonyszinten valósítunk meg egy TCP/IP protokollt, melyet Unix környezetben a Berkeley (BSD) socket könyvtár és API segítségével implementálunk. A BSD socket az eredeti implementációja az internetes kommunikációs protokoll API-nak, melyet manapság a legelterjedtebb protokollként csak TCP/IP-ként emlegetünk.
Minden modern operációs rendszernek van egyfajta BSD socket interfész implementációja. Még a Winsock, az MS Windows implementációja is BSD socket API alapokon nyugszik [29].

7.4.2. Socketek létrehozása

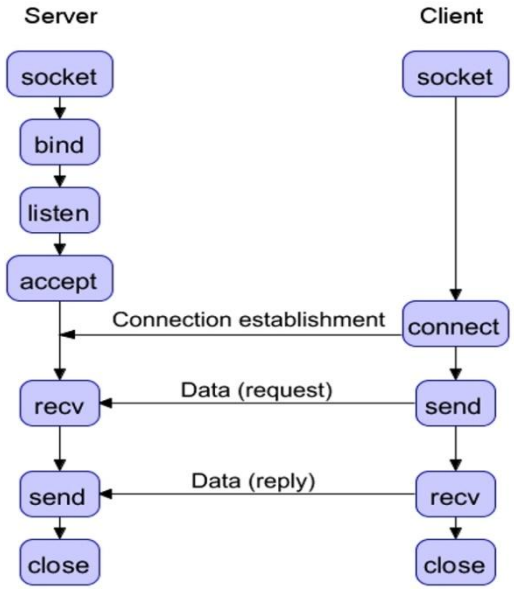
A socketek egy kétirányú kapcsolat végpontjai, melyek segítségével a küldendő adatcsomagok a megfelelő kommunikáló alkalmazáshoz lesznek kézbesítve. Az operációs rendszer socketeket rendel minden kommunikáló alkalmazás-szálhoz. Így ezen alkalmazások számára hozzáférhetővé válik a TCP/IP protokoll, a hoszt gép az IP címe, és az ott kommunikálni kívánó alkalmazás az egyedi portszám alapján egyértelműen azonosítható.

A BSD socketek létrehozása a `socket()` rendszerhívással történik, mely a Linuxos fájlabsztrakciós interfész implementációnak megfelelően egy állományleíróval tér vissza. Ennek segítségével kezelhetjük le a kommunikáció kiépülése során fellépő hibákat. A socket létrehozásakor 3 paraméter definiálja a használandó protokollt. Első a protokollcsaládot adja meg, mely esetünkben az IPv4-es protokollt jelző `PF_INET` érték. Következő a *type* paraméter, mely a protokollcsaládon belül a kommunikáció módját határozza meg. Ez TCP esetén a `SOCK_STREAM`, mely sorrendtartó, megbízható, kétirányú, kapcsolatalapú bájtfolyam kommunikációt valósít meg. A harmadik ezen

protokollcsaládon belül konkretizálja a protokollt, ez általában alapértelmezetten az egyetlen benne található protokollt jelenti.

7.4.3. Szerver és kliens socketek

A stream socket kapcsolatok esetén a socket kapcsolódása aszimmetrikus művelet, így a szerver és a kliens oldalon a megvalósítandó feladatok eltérnek egymástól.



7.4.1. ábra : A szerver és a kliens oldali socket feladatai [30]

A szerver oldalon először a létrejött socketet a szerverprocessz a `bind()` rendszerhívással hozzárendeli egy címhez, amelyen majd várja a kommunikálni kívánó klienseket. Ennek legfontosabb paramétere a címet leíró struktúra. Többek között ügyelni kell a kommunikáció bájtsorrendjére, ugyanis egyes processzorok különböző módon tárolják a különböző adattípusokat. A hálózati bájtsorrend a magasabb helyiértékű bájtot küldi először (big endian). Az Intel 8086 alapú processzorok ennek ellentettjét használják, így konverziós függvényekkel egyeztethetjük a közös bájtsorrendet.

Az IP címeknél az IPv4-es szabványnak megfelelően adjuk meg a címeket, míg a portszámot az IANA, az internet számhozzárendelő hatósága alapján különböző szolgáltatásokra (pl.: ssh, ftp, http) fenntartott portokon kívül érdemes megválasztani.

Ezután a processz egy `listen()` rendszerhívással állítja be a socketre, hogy figyelje a beérkező kapcsolódási igényeket. Még ekkor sem épül ki a kapcsolat, mert a szervernek az igényeket az `accept()` rendszerhívással kell elfogadnia. A `listen()` hívásánál egy paraméterben maximalizálható a kapcsolódó kliensek száma. További igényeket `accept()` hívással nem fogadunk el, ezeket pending connection-nek nevezzük. Az `accept()` visszatérési értéke az új kapcsolat leírója, egy kliens socket, paramétereiben a másik oldal címét kapjuk meg.
A kliens socket esetén a `connect()` rendszerhívásnak adjuk át a szerver címparamétereit, hasonlóan a másik oldal `bind()` hívásánál. Itt adjuk meg a portszám mellett a szerver kapcsolatot azonosító IP címet. Dinamikus IP címek esetén használható a `gethostbyname()` függvény, mellyel lekérdezhető az adott szerver géphez tartozó IP cím, a gép nevét argumentumként megadva a címet automatikusan frissíteni tudja a kliens oldali alkalmazás.

7.5. Az Autosar OS módosításai

A tervezett tool fejlesztése tehát két fő részből fog állni; a TCP/IP szerverként futó AUTOSAR OS módosításai, illetve a socket kliens oldalán a lekérdezések kezelése. Emellett létre kell hozni egy adatbázist, mely a Dio függvényei által lekérdezhető, módosítható GPIO portok értékeit tartalmazzák.
A simulation engine réteg, mely a kapcsolatot valósítja meg az AUTOSAR OS változói és a kliens között, legegyszerűbben magukban az OS taszkokban valósítható meg. Ehhez szükség lesz a TCP/IP szerver oldali implementációjára, a szükséges SW változókhoz pedig közvetlenül hozzáférünk.

7.5.1. TCP/IP szerver socket

A tool segítségével tehát az ECU SW futása során a hardverből érkező adatokat kérdezzük le a kliensoldali program segítségével. Ehhez először létre kell hozni a TCP/IP kommunikációt a szerver és a kliens között. A szerver oldalán egy megfelelő, a

Dio függvényeket hívó utasítások előtt futó taszkban hozzuk létre a szerver oldali socketet.

A korábban említett szerver socket eljárás került be az AUTOSAR OS függvényei közé. A *Startuphook* taszkban található socket kapcsolatot megvalósító függvények csak egyszer futnak le az ütemezés elején, ez a taszk nem kap többször futási jogot. Így ez a taszk megfelelő a szerver socket létrehozásához, hiszen elegendő egyszer, az ütemezés elején létrehozni a kapcsolatot.

7.5.2. MCAL API módosításai

Amint rendelkezésre áll az adatforgalom az AUTOSAR OS és a kliens program között, a Dio API függvényeinek módosítása következik.
Ehhez először a függvények deklarációját kell módosítani a Dio header fájlban. Ebben a fázisban a Dio modulnak nem kell további modulokkal együttműködnie, ezért a fájlban, a szükséges header inkludálások és típusdefiníciók mellett csak a függvények egy egyszerűsített meghatározása marad. A függvény deklaráció az eredeti szoftverben egy speciális, az AUTOSAR szabványnak megfelelő formátumot használ. Ezt később a klasszikus C nyelvi szintaktikára alakítja egy header fájl, a *compiler.h*. Az átalakítás, s ezzel egy további hibalehetőség megspórolható, ha a *Dio.h*-ban rögtön a megszokott szintaktikával hozzuk létre a függvényeket. Az eredeti `Dio_ReadChannel` függvény az alábbi módon került átalakításra. Hasonló módon deklaráltuk a *Dio.h* további függvényeit.

```
// Az eredeti Dio_ReadChannel függvény
FUNC(Dio_LevelType, DIO_CODE) Dio_ReadChannel
(
    CONST(Dio_ChannelType, DIO_CONST) ChannelId
);

// A C szintaktika szerint átalakított alak
Dio_LevelType Dio_ReadChannel(Dio_ChannelType ChannelId);
```

A *Dio.c* fájlban a headerben deklarált függvények definiálása következik. Ez a gyakorlati része a Dio API átírásának, a benne végrehajtott utasítások itt írhatóak át. A `Dio_ReadChannel` függvény esetében a paraméterben átadott `ChannelId`-hoz tartozó szintet szeretnénk megtudni. Ehhez először jelezzük a kliens oldalon, hogy milyen függvényt hívott meg az AUTOSAR OS. Ezt a socket kommunikációt minden modul elejére érdemes betenni, így lehet a kliens oldalon diagnosztizálni a szerver futását.

```
strcpy(out_buf, "FUNC_Dio_ReadChannel");
// a függvény nevének küldése a kliens felé
retcode = send(connect_s, out_buf, (strlen(out_buf) + 1), 0);
```

Ezután a függvény argumentumait küldjük el:

```
sprintf(arg1, "%d", ChannelId);
// a függvény argumentumainak küldése a kliens felé
retcode = send(connect_s, arg1, sizeof(arg1), 0);
```

Ezután azonnal várakozni kezd a kliens válaszára. Amennyiben a szerver lemarad a válaszban kapott értékről, ennél a lépésnél beragad, hiszen a `recv` függvény egy blokkoló utasítás.

```
retcode = recv(connect_s, in_buf, sizeof(in_buf), 0);
```

A válaszban megkapott szint értéket integer számmá alakítva adja vissza a függvény a visszatérési értékében.

```
value = atoi(in_buf);    // kasztolás integerre
return value;      // visszatérés a válasszal
```

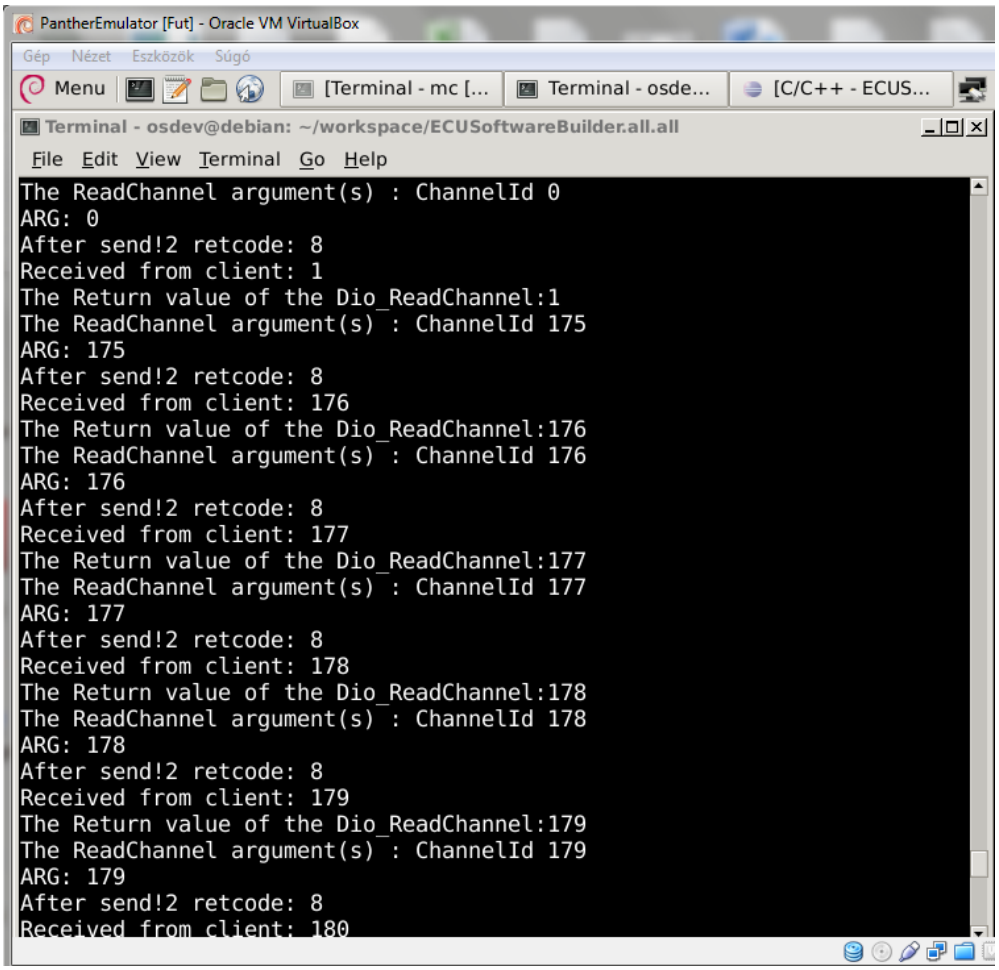
Hasonló logika szerint adjuk át a kliens oldal felé a függvényt és paramétereit, majd kezeljük a válaszban kapott értékeket.

A módosított API forrásfájlokat hozzá kell adni az AUTOSAR OS projekthez. A beinkludálandó header fájlok az *include* mappába mennek, a hozzájuk tartozó *.c* fájlok pedig az *src*, azaz forrás mappába. Ezután a megfelelő make fájlban is hozzáadjuk a módosított API fájlokat. Ez szintén az *src* mappában található *ecuswconf* makefile. Itt a

megfelelő szintaktikával az *src* mappa minden fájlja szerepel, így ide kerül egy új sor, mely a *Dio.c* fájlt illeszti be a projektbe.

```
ECUSWCONF_EXTRA_SRCS += src/Dio.c
```

A Dio API módosítása után a rendszer működését próbáljuk a függvényhívások szintjén modellezni. Különböző prioritású taszkokba illesztünk egyszerű kiíratót függvényeket, melyekkel könnyedén ellenőrizhetjük, hogy az adott taszk hívásai érvényre jutottak-e. Emellett a korábban módosított Dio API függvényeit is kellően gyakran meghívjuk, ezáltal vizsgálva a TCP/IP kommunikáció valamint az adatbázis elérés megbízhatóságát.
Minden ilyen módosítás, ahogy a TCP/IP szerver funkciói is a *tasks.c* fájlban található. A legtöbb Dio API függvényhívás a `TaskBackgroundImpl` taszkban kapott helyet, mely minden OS ütemezési ciklus elején fut le. Minden Dio API függvény hívása előtt egy `connected` változó magas szintje ellenőrzi, hogy létrejött-e már a socket kapcsolat. Ez könnyen ellenőrizhető, iterált lekérdezések után az ütemezés leállítása, és a socket lezárása is ebben a taszkban történik. Ezek mellett tehát minden olyan taszkba bekerült egy diagnosztikai függvényhívás, mely a kliens oldalon jelzi, hogy az adott taszk ütemezése helyesen zajlott-e. Így olyan, időhöz kötött taszkok futását ellenőrizhetjük, mint a 2 ms ciklusonként lefutó `Task2msTrustedImpl`, vagy a 250 µs intervallumonként lefutó `Task250usImpl`.
A szerver oldalon futó alkalmazás fordítását és futtatását egy shell script segítségével végezhetjük. Ez a *build.sh* script, mely az AUTOSAR OS kezelő funkcióit tartalmazza. Ebből nekünk a fordításhoz a *build.sh --all-build* és a futtatáshoz a *build.sh --all-run* parancsok szükségesek. Sikeres fordítás esetén keletkezik egy *.out* kimeneti fájl, melynek futtatásával kezdődik az ütemezés. A szerver oldal indítása tehát az első lépés az alkalmazás használata során. Ekkor a feláll a rendszer és a kliens socket csatlakozására várakozik. Az alábbi képen a szerver alkalmazás futtatását láthatjuk Oracle VirtualBox környezetben.



7.5.1. ábra : AUTOSAR OS szerver [saját forrás]

7.6. *Adatbázis megvalósítás*

Az adatbázis esetében is meg kellett választani az alapvető fejlesztési irányt. Minden TKP-s SW esetén, minden modulra egy külön tábla tárolja majd a porthoz tartozó adatokat. Tehát a táblák állhatnak akár kettő oszlopból is, az egyik a porthoz tartozó egyedi azonosító (primary key), az ehhez tartozó egyetlen adat pedig a port értéke. Mivel az adatstruktúra nagyon egyszerű, több adatbázis megvalósítás is felmerült. A tervezés során az alábbi szempontokat kellett figyelembe venni:

- Bővíthetőség
    - több tábla egy adatbázisban
    - egy cím több jelentésének lekezelése
    - további modulok adatai

- Inicializálhatóság
    - a különböző konfigurációkat egyszerűen adhassuk át az adatbázisnak
- Hozzáférhetőség, hordozhatóság
    - könnyen elérhető kliens oldali használat (driverek, jogosultságok)

A leggyorsabb prototípus megvalósítás az ún. flat-file adatbázissal a legegyszerűbb. Ehhez a megoldáshoz nem kell klasszikus adatbázis-kezelő, mivel minden rekord egy táblában tárolódik, egy fájlból (txt, xls, ...) érhetjük el az adatokat íráshoz, olvasáshoz. Ugyan gyakorlatilag semmit sem kell telepítenünk a használatához, azonban a lekérdezések logikáját nekünk kell implementálni; milyen fájlműveletek kellenek, a rekordokat mivel választjuk el, stb. Az adatbázis-kezelés gyakorlatilag fájlműveletek, esetenként szöveges adatfeldolgozás megvalósításába megy át. Ez könnyen a robosztusság rovására mehet, illetve a későbbi bővítés, fejlesztés is körülményes. Ez talán az első lépés lehet magában a kliens forrásfájlban tárolt port-adat értékpárok leváltására, viszont abszolút nem jövőbemutató, kezelhető vagy általánosítható megoldás.

Ezzel szemben a relációs adatbázisok használata sok előnnyel kecsegtet. Ugyan az adatbázis rendszer felállítása komplikáltabb, szerver kell hozzá, jogosultságok kezelése, adatbázis driverek a kliens alkalmazásokhoz, viszont felhasználói szempontból sokkal kezelhetőbb, megbízhatóbb, mint a flat-file adatbázis.

Az AUTOSAR gyakran xml formátumban tárol adatokat, a modellek is speciális AUTOSAR-os (arxml) fájlokat használnak. Az adatformátumok felesleges átalakítása miatt merült fel az xml alapú relációs adatbázis használata. Az xml adatbázis ugyanúgy igényel keretrendszert, mint a klasszikus relációs adatbázisok, csak az adatok xml struktúrában tárolódnak. Webes alkalmazások esetén javasolt használata, amikor ugyanolyan xml adatformátumot használ az alkalmazás, mint ahogy az adatbázis eltárolja, így felesleges egy további átalakítás. Az xml alapú adatbázis használata sok hasonló típusú adat esetén nem praktikus, a hosszú leíró fájl előnyei több különböző adattípus esetén jönnek elő. Az AUTOSAR-os adatstruktúrák esetén is egyszerűbb, ha az xml-es konfiguráció adatai is relációs adatbázisból származnak, ezek dinamikusabban módosíthatóak.

Nem tudjuk kihasználni az xml alapú adatbázisok előnyeit, tehát érdemes a klasszikus, könnyűsúlyú relációs adatbázisok közt megoldást keresni. Az egyik legelterjedtebb,

szabad felhasználású nyílt forráskódú adatbázis, a ThyssenKrupp Presta által is támogatott MySQL. A cégnél elegendő egy adatbázist igényelni a helyi szerveren a szükséges jogosultságokkal, melyet ezután a belső hálózaton belül bárhonnan lekérdezhetünk. Az adatbázisban tetszőleges számú táblát konfigurálhatunk igényeinknek megfelelően, akár kliens oldalról egy inicializáló script segítségével. Amennyiben több modul ugyanarról a port azonosítóról kérdezne le, a kliens alkalmazásban könnyen a megfelelő táblához tudjuk irányítani a kérést. A MySQL használata megfelelő támogatással rendelkezik, a legfontosabb fejlesztőrendszerek néhány driver telepítésével támogatják a kliens oldali adatbázis hozzáférést. A MySQL rendelkezik egy webes GUI hozzáféréssel is, ahol néhány gombnyomással hozhatunk létre táblákat, ellenőrizhetjük azok tartalmát. Ez a phpMyAdmin, mellyel a fejlesztés kezdeti szakaszában SQL scriptek nélkül is létrehozhatjuk a szerver oldali konfigurációt. Az adatbázis táblák létrehozása szorosan összefügg a kliens alkalmazás feladataival, úgyhogy ennek leírását is a következő részben teszem meg.

7.7. Kliens alkalmazás

A kliens oldali alkalmazás feladata az AUTOSAR OS-sel való kommunikáció, a konfigurálás lehetőségének biztosítása, az ennek megfelelő vezérlés és az adatbázis lekérdezések lebonyolítása. Az alkalmazás a fejlődése során több programnyelven is elkészült. C nyelven íródott az első verzió, mivel a TCP/IP kommunikáció a szerver oldalon így került be az OS taszkok közé. Ezután, a flat-file adatbázis használatának gyorsítása érdekében a Perl nyelvre tértem át, hiszen a script nyelvek segítségével nagy mennyiségű szöveges adat könnyebben kezelhető. Miután megállapítottam, hogy nem kerülhető ki a MySQL adatbázis használata, visszatértem a C nyelvre, mert előzetes értesüléseim szerint ehhez a nyelvhez könnyebben telepíthetőek a kliensoldali lekérdezésekhez szükséges driverek. Azonban kiderült, hogy külön MySQL kliens telepítése szükséges a C nyelven történő kliens-hozzáféréshez. Perl esetén két drivercsomag elegendő a MySQL kliens oldali lekérdezéseihez, így végül visszatértem a Perl scripthez, mely az aktuális környezete a kliens alkalmazásnak.

7.7.1. A kliens socket megvalósítása

Hasonlóan a szerverhez a kliens oldali alkalmazásba is bekerült a TCP/IP socket kezelése. Mivel a végső verzió Perlben készült el, a parancsok szintaktikája némileg eltérő, de a függvények logikája hasonló. A kapcsolódáshoz használt IP címet az első argumentumban beadandó AUTOSAR OS virtuális gépet futtató hoszt nevéből a `gethostbyname()` függvény segítségével kapjuk meg. A port címe a VirtualBox programban is megadott 1050-es, melyet szükség esetén a kliens kódban lehet módosítani. Az AUTOSAR OS indul el először, mely a szerver socket létrehozása után várja a kliens csatlakozását. A kliens alkalmazás indítása után először az adatbázis szerverhez csatlakozik, majd a szerver sockethez. A kapcsolat létrejöttéről egy-egy üzenet küldés és fogadás biztosít.

7.7.2. Az adatbázis lekérdezések megvalósítása, vezérlése

A kliens alkalmazás elején tehát először az adatbázis szerverhez csatlakozunk. Miután megadtuk a belépéshez szükséges adatokat (hoszt cím, adatbázis neve, felhasználónév, jelszó), a Perl adatbázis-kezelő szintaktikájának megfelelően egy egyszerű `connect` utasítással csatlakozhatunk az adatbázishoz. Ezután a kliens socket csatlakozik, majd az alkalmazás várja az utasításokat a szervertől.
A konfigurálhatóság szempontjából fontos, hogy különböző tesztesetek használata során más-más adathalmazzal dolgozhasson a lekérdezés. A kliens alkalmazás mindig ugyanazt a táblát használja a lekérdezések során, ezt az első megvalósított projekt alapján *panther_r8_gpio* táblának neveztem el. A konfiguráció határozza meg, hogy milyen adatok kerüljenek ebbe a táblába. Erre szolgál a kliens alkalmazás futtatásánál használt második argumentum. A script összeveti ezt az argumentumot a munkatábla *(panther_r8_gpio)* nevével, és amennyiben nem egyezik, lemásolja a hivatkozott tábla adatstruktúráját, és feltölti annak tartalmával a munkatáblát. Ez az alábbi Perl függvényekkel valósítható meg.

```
// csatlakozás az adatbázis szerverhez
DBI->connect("DBI:mysql:$db:$host", $user, $pass);
```

```
// munkatábla törlése
DROP TABLE `$table`

// munkatábla létrehozása a config tábla mintájára
CREATE TABLE `$table` LIKE `$config`

// config tábla tartalmának betöltése a munkatáblába
INSERT $table SELECT * FROM $config
```

Ahhoz, hogy a hivatkozott config táblák létrejöhessenek, egy másik script futtatása szükséges. Ez az alkalmazás hasonlóképpen csatlakozik az adatbázis szerverhez, mint a kliens alkalmazás. Az első argumentumban megadott nevű táblát tölti fel véletlenszerű adatokkal, ez a lépés a későbbiekben a konfigurációnak megfelelően módosítható. Amennyiben nem létezik ilyen tábla, a script futtatásánál a második argumentumban a *create* kulcsszó megadásával létre is hozhatjuk azt az ennél a modulnál használatos adatstruktúrának megfelelően. Ezt a scriptet tehát opcionálisan, a kliens alkalmazás futtatása előtt alkalmazhatjuk az ott használni kívánt config tábla létrehozására. Egy példa a script futtatására:

```
insert_default_values_to_given_db.pl test_table create
```

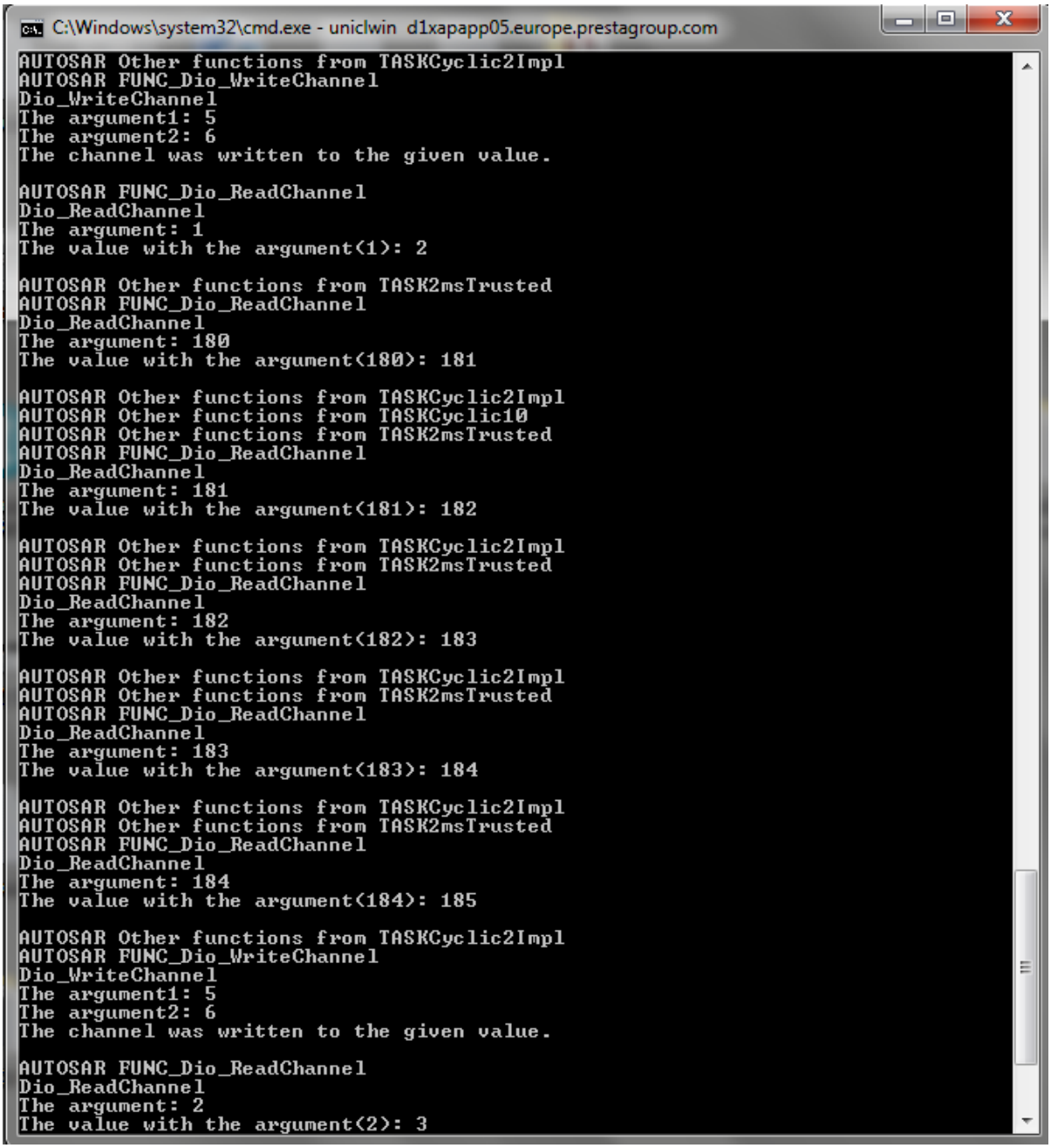
Az adatbázis függvények után csatlakozik a kliens socket a feljebb említett módon. A kapcsolat létrejötte után egy végtelen ciklus kezdődik. Ennek első lépéseként a kliens üzenetet vár az AUTOSAR OS-től. A végtelen ciklus egy lefutása egy ilyen szerver utasítást kezel le. Amennyiben olyan függvény hívódott meg az AUTOSAR OS-ben, mely kiszolgálást vár a klienstől, az egy TCP/IP üzenetben átküldésre kerül a hozzá tartozó argumentumokkal együtt.
Ezután egy egyszerű switch-case szerkezet elnavigál az utasítás lekezeléséhez, ezáltal a megfelelő adatbázis lekérdezésekhez.
A `Dio_Read_Channel()` függvény hívása esetén az argumentumban megadott porthoz tartozó jelszintet akarjuk kiolvasni. Ehhez az adatbázison egy `SELECT` lekérdezést kell végrehajtanunk. Amennyiben ilyen port nem található az adatbázisban, létre kell hozni az adott rekordot a default értékkel. Ezek az SQL lekérdezések az alábbiak:

```
SELECT Value FROM $table WHERE Port = $portid
INSERT INTO $table (Port, Value) VALUES ($portid, 0)
```

A visszakapott értéket a válaszban elküldi a socketen keresztül a szervernek. Emellett diagnosztikai célból a szerver és a kliens oldalon is kiíratásra kerül.
A `Dio_Write_Channel()` függvény esetén felül kell írnunk az adatbázisban a porthoz tartozó aktuális értéket. Ekkor tehát két argumentumot kell átküldenünk, a port azonosítóját és az új értéket. Amennyiben ez a változó nem is létezik, az előző függvényhez hasonlóan létre kell hoznunk a kezdőértékkel. A port írás lekezelésére egy speciális SQL lekérdezést használhatunk, mely csak abban az esetben update-eli az adott változót, ha már benne van az adattáblában. Egyéb esetben hozzáad egy új sort az `INSERT` utasítással.

```
INSERT INTO $table
    (Port, Value)
VALUES ($portid, $writevalue)
ON DUPLICATE KEY UPDATE Value = $writevalue;
```

Innen nem vár válaszértéket a szerver, a kliens azt küldi vissza, hogy sikeresen felülírta az adott értéket.
A kliens alkalmazás kiléptethető a végtelen ciklusból, amennyiben a szerver üzenetben az *'exit'* kulcsszót kapja meg. Ekkor lezárja a socket, illetve az adatbázis kapcsolatokat és leáll. Az alábbi ábrán a kliens alkalmazás futását láthatjuk Windows Terminal ablakban.



7.7.1. ábra : Kliens alkalmazás [saját forrás]

7.8. *Az ECU szimuláció használata*

Az elkészült tool-t egyre inkább a mindennapi felhasználóbarát alkalmazásra kell felkészíteni. Ezek a szempontok a fejlesztés során is figyelembe voltak véve, így készült el a kliens alkalmazás a könnyen hozzáférhető Perl adatbázis-kezelő függvények segítségével. A szerver valószínűleg egy központi helyen lesz elérhető és ennek konfigurációja ritkábban változik. A mindennapi használat során ezen az oldalon a SW

verziók folyamatos frissítése a megoldandó feladat. A folytonos integrációhoz tartozó automatikus tesztelések a fejlesztés korai szakaszában megvalósulhatnak. A szerver oldal konfigurációja a 7.3.2. alfejezetben leírtaknak megfelelően történik, a későbbiekben érdemes lehet egy külön szervert dedikálni erre a feladatra, így egyszerűsíteni a használatát.
A kliens oldal beállításához alapvetően két funkciót kell lehetővé tennünk, a Perl scriptek futtatását, illetve innen a MySQL adatbázis elérését. Egy megfelelő csomag telepítésével ezt egy lépésben megtehetjük.
A Perl scripthez az adatbázis kezelő parancsokat a megfelelő adatbázis kezelő driver telepítésével érhetjük el. A kliens tehát MySQL adatbázishoz szeretne hozzáférni, így meg kell keresnünk azt a drivert, mely ezt támogatja. A ThyssenKrupp Prestánál támogatott drivercsomag (Perl Activestate) aktuális verziója nem támogatja a MySQL csomagokat, korábbi verziókhoz pedig nehéz a hozzáférés. Egy hasonló telepítő, a Strawberry Perl 5.10.0.6 verziója viszont tartalmazza a szükséges drivereket, így az általános adatbázis-kezelő DBI modult, és a speciálisan MySQL-hez használható DBD-mysql modult. A Strawberry Perl telepítése után ellenőrizzük a Perl parancsokhoz tartozó elérési utat, hogy a további műveletek az új Perl verziót használják-e, véletlenül sem a korábbi, minden gépen telepített Activestate Perlt. Ezután a Perl Packet Manager (PPM) segítségével feltesszük a fent említett adatbázis-kezelő drivereket. Ehhez a konzolablakban az alábbi egyszerű parancsok tartoznak.

```
ppm install DBI
ppm install DBD-mysql
```

A sikeres telepítés után nem kaphatunk hibát a Perl scriptben implementált MySQL parancsok futtatása során. Ezzel a kliens gépen minden rendelkezésre áll a tool használatához.

A szerver oldalon az AUTOSAR OS indításához a 7.5.2. alfejezet alapján a *build.sh --all-run* parancsot használhatjuk. A kliens script indítása előtt érdemes meggyőződni az adatbázis megfelelő működéséről. Erre alkalmas a phpMyAdmin felület, ahol ellenőrizhető, hogy működik-e a szerver és a használni kívánt táblák megfelelő tartalommal vannak-e jelen. Ellenkező esetben a szerver elindítása után az

insert_default_values_to_given_db.pl script futtatásával létrehozhatjuk a szükséges táblákat.

Ezután már csak a kliens script futtatása maradt hátra. Egy opcionális harmadik argumentuma, melyről még nem esett szó, a konzol ablak helyett egy text fájlba írja ki a futás paramétereit. Egyszerűen a fájl nevét megadva a kliens script mappájában létrejön az adott fájl a konzol ablak tartalmával. Az első argumentum tehát az AUTOSAR OS-t futtató szerver hoszt neve, a második a lekérdezések során használt tábla neve, a harmadik pedig a kliens log fájl neve. Egy ilyen futtatásra példa az alábbi:

```
perl perl_client.pl d11388.europe.prestagroup.com default_data log.txt
```

7.9. Továbbfejlesztési lehetőségek

Az elkészült tool egy kiindulási alap egy, a folytonos integráció bevezetését teljes körben támogató eszköz létrehozásához. A legnagyobb nehézséget igyekszik megoldani, hogy a HW függő, alacsonyszintű fejlesztések a magasabb szintű programnyelvekhez hasonlóan azonnal integrálhatóak és tesztelhetőek legyenek. Ez valóshoz közeli eredményt csak úgy tud szolgáltatni, ha a hardverről minél pontosabb modell áll rendelkezésünkre. Ez nem kis feladat, főleg ilyen összetett rendszerek esetén, mint a TKP kormányvezérlői.

Első körben a megvalósítandó cél az, hogy minél előbb egy olyan eszköz álljon rendelkezésre, melyen egy teljes SW futtatható, tesztelhető. A fent leírt fejlesztés során a DIO modul függvényei készültek el. A teljes rendszer szimulációjához az összes, hardverrel kapcsolatos modul API-ját ennek megfelelően át kell alakítani. Ezek közül kardinális a CAN és a FlexRay modulok módosítása, az API megfelelő átalakítása után a protokoll használatával a rendszer rengeteg kommunikációs funkcióját el tudja látni a PC-n futó tool.

Amennyiben minden modul rendelkezésre áll, hogy egy teljes SW képes legyen lefutni PC-s környezetben, meg kell oldanunk, hogy az integráció automatikus legyen. Ehhez nem kell mást tenni, mint az eredeti API fájlokat a módosítottakra lecserélni a projektben. Ez egy alkalmas scripttel, akár a kliens eszköz futtatása elején elvégezhető.

A következő lépés, hogy minél több projekten lehetővé váljon a tool használata. Ennek érdekében az esetleges különbségek kezelésére fel kell készíteni az eszközt, az egyes projektek esetén több, vagy más API-kat kell lecserélni, ennek is automatikusan kell működnie.

A széles körben alkalmazott tool modelljét tovább lehet finomítani, ha a kliens oldalról átküldendő adatok valóságtartalmát fejlesztjük. Az olyan hardver modulok, melyek egy szenzor beolvasásánál bonyolultabb feladatokat is ellátnak, így a szűrők, ADC/DAC, PWM, PLL, léptetőmotorok, stb. működése szimulálható akár MATLAB Simulink segítségével is. A megfelelő paraméterekkel felkonfigurált elemek által létrejött adatok közelebb állhatnak a valóságos, folytonos működéshez, mint az egyes órajelekre beadott konstans értékek. A szimulált analóg modulokból származó adatok segítséget nyújthatnak különböző konfigurációk esetén az adatbázis valósághű paraméterekkel való feltöltéséhez is.

Egy megbízhatóan működő tool esetén megfontolandó a részegységek (kliens, szerver, adatbázis kezelő, integrátor script) egy egységes felületre való integrációja. Így egyetlen ablakból vezérelhető lenne a tesztelni kívánt SW betöltése, az API-k lecserélése, a kívánt konfigurációk kiválasztása, majd a futtatás (szerver és kliens oldal egyaránt). Egy ilyen, csak a legszükségesebb felhasználói beavatkozásokkal, kényelmesen működtethető eszköz valódi segítsége lehet a folytonos integráció széles körű bevezetésének a ThyssenKrupp Presta fejlesztési folyamataiban.

Irodalomjegyzék

[1] Liszbauer Tamás: Szoftverintegrációs tesztelés automatizálása (2012) Budapesti Műszaki Egyetem, Budapest

[2] Scherer B., 2010. Rendszertervezés (Embedded System Design). [óravázlat] Budapesti Műszaki Egyetem, Budapest

[3] http://www.extremeprogramming.org [megtekintve: 2013.05.11.].

[4] Hegedűs B. (ThyssenKrupp Presta): Feature Driven Development Process Training Material (TRA-03811-EN) 2012.

[5] Marossy K.,Dr. Charaf H. Komponens alapú programozás COM+ és .NET környezetben [online] http://nws.niif.hu/ncd2002/docs/ehu/28/index.html [megtekintve: 2013.05.11.].

[6] Ficsor L., Dr. Kovács L., Krizsán Z., Dr. Krusper G.: Szoftvertesztelés [online] http://www.inf.unideb.hu/kmitt/konvkmitt/szoftverteszteles/book.xml.html [megtekintve: 2013.05.11.].

[7] http://en.wikipedia.org/wiki/Test-driven_development#mediaviewer/File:Test-driven_development.PNG [megtekintve: 2013.05.11.].

[8] Wikipedia: Test-driven development http://en.wikipedia.org/wiki/Test-driven_development [megtekintve: 2013.05.11.].

[9] http://www.autosar.org/ [online - megtekintve: 2013.05.11.].

[10] www.martinfowler.com/articles/continuousIntegration.html [online - megtekintve: 2013.05.11.].

[11] http://agilemanifesto.org/iso/hu/principles.html [online - megtekintve: 2013.05.11.].

[12] Jez H., David F.: Continuous Delivery 2011 https://buildrelease.googlecode.com/hg-history/1998cd1d530b35b79740d7bf93f8915548136c25/Trunk/BreBooks/Continuous%2520Delivery.pdf [online - megtekintve: 2013.05.11.].

[13] Eric R. : Continuous deployment in 5 easy steps (2009) http://radar.oreilly.com/2009/03/continuous-deployment-5-eas.html [online - megtekintve: 2013.05.11.].

[14] Galambos R. (ThyssenKrupp Presta): Build Speed up Training Material 2013.

[15] Szikes Á. (ThyssenKrupp Presta): TKP Dev GUI Training Material 2013.

[16] Martin F. : FeatureBranch (2009) www.martinfowler.com/bliki/FeatureBranch.html [online - megtekintve: 2013.05.11.].

[17] EB Tresos Wincore http://automotive.elektrobit.com/ecu/eb-tresos-wincore [online - megtekintve: 2013.05.11.].

[18] Hupwiki: GCC http://wiki.hup.hu/index.php/Gcc [online - megtekintve: 2013.05.11.].

[19] Webopedia: Linker http://www.webopedia.com/TERM/L/linker.html [online - megtekintve: 2013.05.11.].

[20] Kitlei R.: Segédanyagok www.kitlei.web.elte.hu [online - megtekintve: 2013.05.11.].

[21] Wikipedia: Compiler https://en.wikipedia.org/wiki/Compiler [online - megtekintve: 2013.05.11.].

[22] Richard M. S., Roland McG, Paul D. S.: GNU Make (2013) http://www.gnu.org/software/make/manual/make.pdf [online - megtekintve: 2013.05.11.].

[23] Compiler Optimizations: Function Inlining http://www.compileroptimizations.com/category/function_inlining.htm [online - megtekintve: 2013.05.11.].

[24] Options That Control Optimization http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html [online - megtekintve: 2013.05.11.].

[25] Electric Cloud: ElectricAccelerator https://www.electric-cloud.com/products/electricaccelerator.php [online - megtekintve: 2013.05.11.].

[26] Faragó Dániel: AUTOSAR FlexRay kommunikációs modulok megvalósítása (2012) Budapesti Műszaki Egyetem, Budapest

[27] http://www.autosar.org/gfx/AUTOSAR_ECU_softw_arch_b.jpg [online - megtekintve: 2013.05.11.].

[28] http://www.autosar.org/download/conferencedocs/07_Elektrobit_Ethernet_for_Autosar.pdf [online - megtekintve: 2013.05.11.].

[29] Berkeley Sockets http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Berkeley_sockets.html [online - megtekintve: 2013.05.11.].

[30] TCP and UDP in Socket APIs http://liuj.fcu.edu.tw/net_pg/socket.html [online - megtekintve: 2013.05.11.].