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

Automatikus, modellalapú biztonsági analízis

Készítette
Ratkovszki István

Konzulensek
dr. Sujbert László, Monostori Dénes

2017

MŰEGYETEM 1782

DIPLOMATERV-FELADAT

Ratkovszki István (AUPYJJ)
szigorló villamosmérnök hallgató részére

Automatikus, modellalapú biztonsági analízis

A modern gépjárművek biztonságtechnikai és kényelmi funkcióinak megvalósításában, környezetvédelmi jellemzőinek javításában stb. egyre jelentősebb szerepet kapnak a számítástechnikai megoldások. Ma egy prémium személyautó gyártójának közel száz elektronikus vezérlőegységből (ECU) és számos fedélzeti kommunikációs sínből kell kialakítani egy megbízhatóan működő elosztott rendszert, amely komoly algoritmus- és kommunikációtervezési, illetve munkaszervezési kihívást jelent. Az így adódó komplexitás uralására alakultak ki különféle szabványok, mint például az AUTOSAR modellező nyelv, melynek segítségével elosztott szoftverrendszerek leírása válik lehetővé.

Az így létrejövő modelleket elsősorban implementációs célból használják, azaz a modell alapján a futtató platform konfigurációját (és részben implementációját) generálják. Mivel azonban a modellek részletesek, és mindig szinkronban vannak a megvalósítással, új felhasználási területeket is találhatunk számukra. Az egyik ilyen a szoftver FMEA (hibamód- és hibahatás-analízis) támogatása automatikus hibaterjedés-analízissel, az FMEA riportok generálásával.

A jelölt feladata egy meglévő modellező eszköz kiegészítése az FMEA támogatás megvalósítására. Konkrétan az alábbi célokat kell elérnie:

- Az AUTOSAR modellező nyelv megismerése
- Az FMEA módszertan megismerése
- Szerkesztőprogram készítése a komponensenkénti hibamodellek egyszerű megadásához
- A hibaterjedés számításának automatizálása a komponensek közötti összeköttetések figyelembevételével
- A hibaterjedés eredményeinek megjelenítése, riportok készítése

Tanszéki konzulens: Dr. Sujbert László, docens
Külső konzulens: Dr. Balogh András, Monostori Dénes (ThyssenKrupp Presta Hungary Kft.)

Budapest, 2016. március 19.

……
Dr. Dabóczi Tamás
tanszékvezető

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

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

TARTALOMJEGYZÉK

HALLGATÓI NYILATKOZAT

Alulírott Ratkovszki István, 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, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2017. 05. 19.

.....

Ratkovszki István

Összefoglaló

Napjainkban egyre több helyen alkalmazzák a modellvezérelt fejlesztés paradigmáját különféle – akár biztonság-kritikus – rendszerek fejlesztése során is. A mérnöki modellező nyelvek legfőbb előnyei a magas-szintű megközelítés, precíz szintaktika, és gyakran a gazdag szemantika.

A biztonság-kritikus rendszerek tervezése során a tervezési módszerek mellett több analízis módszer használata is szükséges, hogy a tervek és a végső rendszer helyességét belássuk. A modell-alapú megközelítés előnye, hogy a modell az analízis alapja is lehet, így nem kell minden információt újra (manuálisan) megadni, ezzel csökkentve az időráfordítást, és a hibázási lehetőségeket.

Az egyik gyakran használt analízis megközelítés a hibamód és hatás analízis (Fault Mode and Effect Analysis – FMEA), melynek során az elemi hibák a teljes rendszerre gyakorolt hatását vizsgáljuk.

A feladat egy olyan kiegészítés készítése egy Eclipse alapú, ipari modellező alkalmazáshoz, ami támogatja az FMEA analízis elvégzését a tervezői modellek alapján, segítve az analízist végző szakértő munkáját, biztosítva, hogy az analízis mindig szinkronban legyen a rendszer viselkedésével.

Abstract

Nowadays, the paradigm of model-driven development is being applied in more and more places in developing various systems, even security-critical ones. The main advantages of these modeling languages are the high-level approach, the precise syntax, and often the rich semantics.

In designing security-critical systems, - besides design methods - a number of analysis methods are necessary/required to understand the pertinence of the designs and the final system. The advantage of the model-based approach is that the model can be the basis of analysis, so it is not necessary to reenter all the information (manually) again, reducing time for implementation and analysis and also the possibility of making mistakes.

One of the frequently used analytical approaches is the Fault Mode and Effect Analysis (FMEA), which examines the effect of elemental errors on the whole system.

The task of this thesis is to make an addition to an Eclipse-based, industrial modeling application that supports FMEA analysis based on design templates, which is able to help the analytical engineers to ensure that the analysis is always synchronized with the behavior of the system.

1. Bevezetés

Jelen diplomaterv célja egy kiegészítés, bővítmény készítése egy olyan autóiparban használt szoftverhez, amely lehetővé teszi a fejlesztett autóipari komponensek biztonsági analízisének könnyebb végrehajtását.

Az autóiparban fejlesztett szoftverek gyakran biztonságkritikus rendszerek részei, ezért is nagyon fontos, hogy a komponensek minél többféle analízisét elvégezzük mielőtt azok egy autó elektronikus vezérlőegységeibe (ECU) kerülnének.

Az egyik ilyen analízis módszer a hibamód és hatás analízis (továbbiakban FMEA – Failure mode and effect analysis), melynek a lényege, hogy a komponensek lehetséges hibamódjait összegyűjtve azok hatását a rendszerre vizsgálva megkaphatjuk, mely szoftver összetevők javítása, revizionálása a leglényegesebb.

Az autóipari eszközök fejlesztése az utóbbi évtizedekben egyre komplexebb feladattá vált, amit nehezített, hogy a különböző gyártók, beszállítók nem használtak egy egységes szabványt az általuk készített komponenseknél, emiatt a különböző forrásból származó komponensek használatához különböző interfészek, implementációk voltak szükségesek.

A probléma megoldására a nagyobb autóipari gyártók létrehoztak egy konzorciumot, amelynek célja egy egységes szabvány kidolgozása volt. Ennek az eredménye lett az iparban egyre szélesebb körben használt AUTomotive Open System ARchitecture (AUTOSAR). Ez a szoftverek fejlesztésénél használt módszertanokat, alap szoftver (BSW) csomagot valamint egységes teszteket biztosít a felhasználók számára, amelyek alapján megfelelő termékek előállítására lehetnek képesek.

A hiba analizáló rendszert, egy autóiparban használt szoftver bővítményeként kell megírni, amely a szabvány szerinti fejlesztést teszi lehetővé. Ez a szoftver Eclipse alapú, a thyssenkrupp Presta Hungary által fejlesztett program, ami az AUTOSAR Architect nevet viseli.

A szoftverek növekvő komplexitásának eredménye, hogy az egyszerű, szöveg alapú fejlesztés módszertana nehézkessé, már-már teljesíthetetlenné teszi a komponensek karbantartását, implementálását. Az autóiparban használt szoftvereknél lényeges, hogy a teljesítményük a maximális legyen, a fejlesztésük pedig egyszerű, és átlátható. Ez a programozási nyelvek szempontjából közelítve azt jelenti, hogy a programoknak a determinisztikusan kell működniük, az erőforrások minél hatékonyabb kihasználásával, ami a C nyelvre jellemző, emellett pedig olyan nyelvi eszköztárral kell rendelkezzenek mint a Java vagy C++.

A probléma megoldására kidolgoztak egy új módszert, a modellalapú fejlesztést. A lényege, hogy a szoftvereket elemenként összerakva, az elkészített modellen különböző analízis módszereket végrehajtva, végül belőle kódot generálva egyszerűsíti le a fejlesztés folyamatát, ezzel csökkentve az implementálás időszükségletét, valamint a hibázási lehetőségek számát.

Az AUTOSAR Architect az autóipari szoftverek ilyen nemű fejlesztését teszi lehetővé. A program tartalmazza az AUTOSAR szabvány által definiált modell struktúrát, ennek felhasználásával a fejlesztők képesek grafikus felületen komplex szoftverek implementációjára.

Az AUTOSAR Architect-hez hasonló, Eclipse platform alapú programokhoz bővítményeket, plug-in-eket lehet csatolni, így különböző funkcionalitásokkal kibővítve őket. Az analízist megvalósító programrészt egy ilyen bővítmény formájában valósítottam meg.

A megoldás során az Eclipse, és a Java különböző technológiáival, a modellvezérelt fejlesztés metodikájával, és különböző hibaanalízis módszerekkel ismerkedtem meg, majd használtam fel őket.

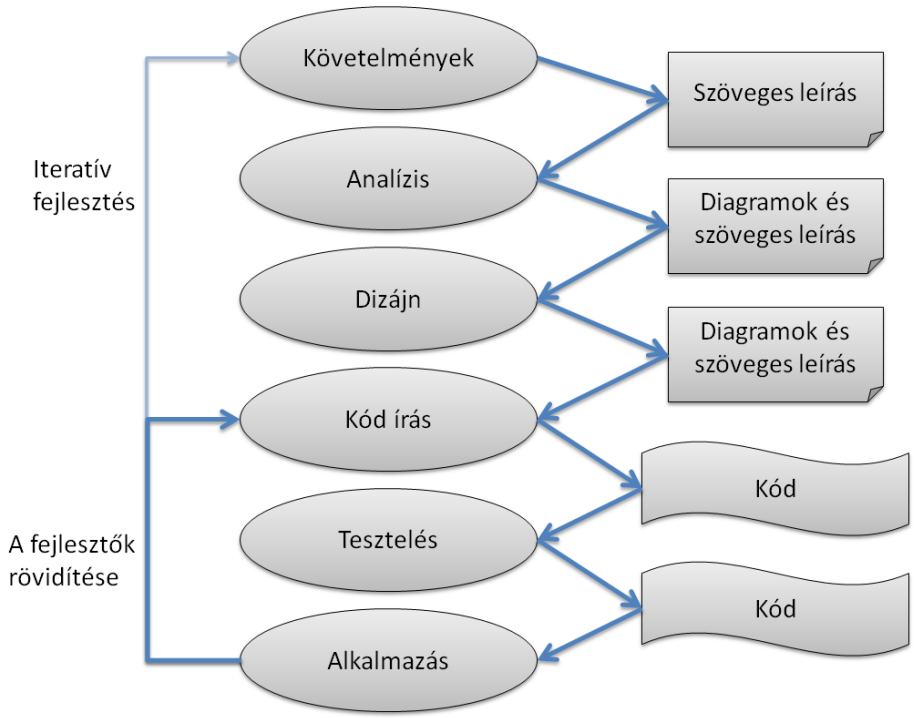
Ahhoz, hogy a diplomaterv érthető legyen minden olvasó számára, lényeges, hogy ezekről az olvasó is alapvető tudással rendelkezzen, emiatt a diplomaterv első felében egy átfogó bemutatást tartok róluk.

2. Modellvezérelt fejlesztés [1]

A szoftver fejlesztést gyakran hasonlítják össze a hardver fejlesztéssel érettségük szempontjából. Míg a hardver fejlesztésben nagy fejlődés történt, pl. a processzorok sebessége exponenciálisan nőtt az utóbbi évtizedekben, a szoftverek szempontjából a változás tulajdonképpen egészen minimális. A modellvezérelt paradigma a szoftverfejlesztés egyik relatíve új aspektusa.

Ahhoz, hogy megmutassam, a modellvezérelt fejlesztés miben nyújt előnyöket, először a hagyományos szoftverfejlesztés néhány problémáját vázolom, majd ismertetem, ezekre milyen alternatívát kínál.

2.1. Hagyományos szoftver fejlesztés



2-1Hagyományos szoftverfejlesztés életciklus modellje

A 2-1 ábra a hagyományos szoftverfejlesztés általános metodikáját, életciklusát írja le.

2.1.1. Fejlesztési problémák

A szoftverfejlesztés területén több komoly problémával is szembe kell néznie a fejlesztőknek. A szoftver megírása munkaigényes, minden új technológiával újabb és újabb problémákat kell megoldani, az implementációt átdolgozni. A rendszereket sosem egy technológiát használva építik meg, ezeknek pedig egymással kommunikálni kell. Ugyanígy problémát jelent, hogy a szoftvereknek meg kell felelniük a folytonosan változó követelményeknek mind funkcionalitás, mind felépítés szempontjából.

2.1.2. Dokumentációs problémák

A fejlesztés folyamata 6 fázisból épül fel, melynek jellemzően az első felében (2-1 ábra 1-3 lépése) készülnek a dokumentációk, és diagramok, attól függetlenül, hogy inkrementális, iteratív, vagy esetleg a klasszikus vízesés modell szerint készül el a szoftver. Ezek a jellemzően követelmény leírások szöveges formában, képekkel valamint gyakran Unified Modeling Language (UML) diagramokként vannak szemléltetve.

Ezek a dokumentációk gyorsan érvényüket veszthetik mikor a kódolás elkezdődik, és ahelyett, hogy a program pontos dokumentációját tartalmaznák, ahhoz többé-kevésbé nem kapcsolódó leírásokká változnak. A változtatások leggyakrabban csak a kódban érvényesülnek, a fejlesztésre szánt idő szűkössége miatt, emellett a fejlesztők megkérdőjelezhetik a dokumentáció frissítés projekthez adott értékét.

A dokumentáció készítése tehát egy – sajnálatos módon – nehézkes, és nem kívánt feladata a fejlesztőknek. Ezek ellenére a jó dokumentáció egy fejlesztési projektnél elengedhetetlen, és szükséges a program karbantarthatósága, továbbfejleszthetősége szempontjából.

2.1.3. Hordozhatósági problémák

A szoftveripar jellemzően különbözik fejlődés szempontjából a többi iparágtól. Évente, vagy akár annál gyorsabban is jelennek meg új technológiák, és válnak kedveltté a felhasználók körében.

Sok cég kénytelen követni az új trendeket több okból:

- A felhasználóik/vásárlóik igénylik ezek használatát
- A problémáik megoldására alkalmas az új technológia
- Az eszköz gyártók nem támogatják tovább a korábbi megoldásokat

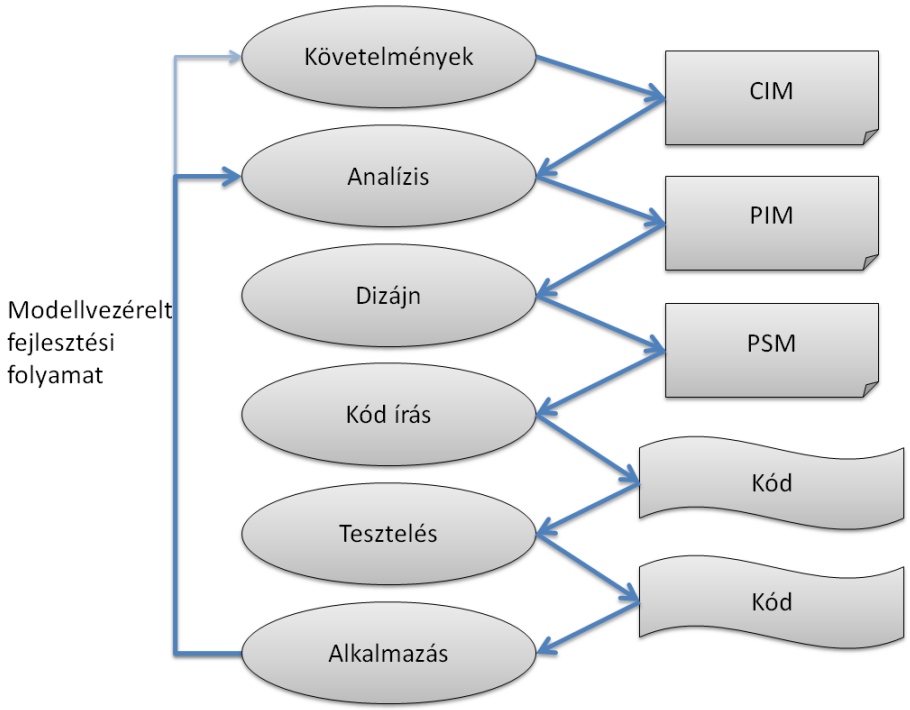
Ezek az új megvalósítások gyakran szolgálnak előnyökkel a cégek számára, és sokan nem tehetik meg, hogy lemaradjanak. Emiatt gyorsan adaptálni kell az új megoldásokat, aminek következményeképp a korábbi technológiákba befektetett energia és pénz értékét veszti.

Ennél még komplexebb a probléma, mert a technológiák maguk is változnak, különböző verziók jelennek meg, és nem garantálják a kompatibilitást a korábbiakkal. A következmény, hogy a már létező megoldásokat, szoftvereket ennek megfelelően írják át, portolják, vagy annak új verziójára frissítik.

Természetesen a szoftver megmaradhat a régi formájában, ez esetben viszont biztosítani kell azt, hogy az újabb technológiákkal készült egyéb rendszerekkel kommunikálni tudjon.

2.2. Modellvezérelt fejlesztés

2.2.1. Alapok



2-2 Modellvezérelt szoftverfejlesztés életciklus modellje

A modellvezérelt fejlesztés életciklusa, amit a 2-2 kép mutat, ránézésre nem tér el túlságosan a hagyományos fejlesztés modelljétől (2-1 ábra). Az egyik legnagyobb különbség a fejlesztés során készített állományokban található. Olyan formális modell kimenetei vannak ennek a metódusnak, amiket például a számítógépek értelmezni tudnak. A következő három modell az modellvezérelt fejlesztés magját képzi.

2.2.1.1. Számítástól független modell

A követelmények meghatározása során a számítás független modell (Computation Independent Model - CIM) elkészítése a cél. Ez egy olyan, egész rendszert leíró reprezentáció, ami a megvalósítástól teljesen eltekint. A célterület szókincsét használva megadja, hogy a rendszernek pontosan milyen funkciókra kell képesnek lennie, ezzel igyekszik áthidalni a célterület szakértői, és a funkciók implementálói közötti értelmezési különbségeket.

2.2.1.2. Platform független modell

A második modell egy olyan magas szintű absztrakció, ami független mindenféle megvalósítási technológiától. Ezt Platform független modellnek (Platform Indepentend Model – PIM) nevezik. Ebben van specifikálva, hogy egy rendszer hogyan képes legjobban ellátni a neki szánt feladatokat, függetlenül attól, hogy a végső megvalósítás milyen technológiával történik.

2.2.1.3. Platform specifikus modell

A PIM-et az modellvezérelt szoftverfejlesztés következő lépésében át kell transzformálni egy, a megvalósítás platformjától függő modellre. Ezt nevezik platform specifikus modellnek (PSM). A PSM kialakítása szerint információkat tartalmaz arról, hogy egy implementációs technológia esetén milyen építőelemekből épülhet fel maga a modell, miként lehet annak egyes részeit a technológia „alkatrészeiből" megalkotni. A PIM-et egy, vagy több platformra is áttranszformálhatják, ezekhez mind külön-külön PSM tartozik. Manapság a legtöbb rendszer több technológiát is felsorakoztat, így egy-egy PIM-ből több PSM transzformálása is szükséges.

2.2.1.4. Kód

Az utolsó lépés a fejlesztési folyamatban a PSM átalakítása kóddá. Ez a lépés tulajdonképpen elég egyértelműen végrehajtható, ugyanis a PSM a megvalósítás technológiájához elég közeli modellt eredményez.

2.2.1.5. Kiegészítés

A modellvezérelt fejlesztés még nem egy teljesen kiforrott módszer, nem is tudja teljesen felcserélni a hagyományos fejlesztést, legalábbis olyan szempontból, hogy a transzformációk során nem feltétlenül képes minden metódust és elemet legenerálni. Ezért a PSM és kód absztrakciós szintjein is szükség van kód megírására, revizionálására, de ez jóval kevesebb munkát igényel, mint a hagyományos fejlesztésnél.

2.2.2. Modellvezérelt fejlesztés előnyei

A modellvezérelt fejlesztés elméleti folyamatának megismerése után, az ezekből származó előnyöket analizálom.

2.2.2.1. Fejlesztés

A fejlesztők feladatának fókusza eltolódik a PIM elkészítésére, a platform specifikus részeket a transzformálás folyamata adja a modellhez. Természetesen a PIM-PSM transzformációt meg kell írni minden platformra külön-külön, ami igen specifikus, és nehéz feladat lehet, ám ha egyszer elkészül, ugyanazt a megoldást több rendszer elkészítésénél is lehet alkalmazni, tehát újrahasználható, így a befektetett energia hosszú távon megtérül.

Így tehát a fejlesztők nagy részének feladata a PIM megalkotása, ami esetén nem kell törődniük a platformok különbözőségéből adódó követelményekkel, részletekkel, azok a transzformáció során hozzáadódnak a modellhez. Ez azt eredményezi, hogy a fejlesztőknek kevesebb munkája van a modellel, valamint a PSM és kód absztrakciós szintjein kevesebb kód megírása szükséges.

Másik előnye, hogy a fejlesztők több időt tudnak így fordítani az aktuális probléma megoldására, mint a kóddal való foglalkozásra. Ez olyan szoftvert eredményez, amely sokkal jobban illik a felhasználók, megrendelők igényeihez, jobb funkcionalitással ellátott szoftvert kapnak rövidebb idő alatt.

2.2.2.2. Dokumentáció készítés

A PIM fejlesztése során a fejlesztők elkészítik a szoftver felépítését, kapcsolatait, működésének alapjait. Ez tulajdonképp azt jelenti, hogy a PIM egy magas szintű dokumentációnak felel meg. A legnagyobb különbség a hagyományos dokumentációk, és e között, hogy ezt nem hagyják magára miután elkészítették azt. Mivel a szoftver fejlesztésének ez az absztrakciós szint az alapja, a fejlesztéssel együtt a dokumentációt is elkészítik egyszerre.

Gyakorlatban a változtatások a szoftverben leggyakrabban a PSM szintjén történnek meg, és innen generálják a kódot, azonban megfelelő eszközök képesek a kapcsolatokat felismerni a PIM és PSM közt, és azonosítani a változásokat, majd azokat megjeleníteni a PIM-ben, hogy konzisztens maradhasson a kóddal.

2.2.2.3. Hordozhatóság

A hordozhatóság megvalósítása tulajdonképpen evidensen következik magából a modellvezérelt fejlesztési folyamatból. Mivel az első lépés egy platform független modell meghatározása, ezért csak egy megoldáson kell változtatni, és ezt több különböző platformra letranszformálni. Ez azt jelenti, hogy minden, ami a PIM absztrakciós szinten definiált, teljesen hordozható.

A hordozhatóság korlátait ez alapján egyértelműen a transzformációt végző eszközök képességei jelentik. A népszerű platformokhoz nagy valószínűséggel több megfelelő eszköz is található, azonban az új, vagy kevésbé ismert technológiájú platformokhoz elképzelhető, hogy a fejlesztés során meg kell valósítani egy saját eszközt.

2.3. Az AUTOSAR Architect rövid bemutatása modellvezérelt fejlesztés szempontjából

Mint azt a bevezetőben írtam, a diplomatervezés feladat egy – a thyssenkrupp Presta Hungary által fejlesztett – szoftverhez egy bővítmény készítése. Ez a szoftver az imént bemutatott modellvezérelt fejlesztés paradigmáját támogatja, ebből a szempontból mutatom be röviden a rendszert.

2.3.1. Absztrakciós szint

A programot vizsgálva az absztrakciós szintjének kérdésére nem egyértelmű a válasz, annak két megközelítése van.

Ahogy a program nevéből látszik, maga a szoftver a későbbiekben bemutatott AUTOSAR autóipari szabvány alapján való fejlesztést támogatja, ebből a szempontból tehát nem teljesen platform független a megvalósítás, hiszen a szabvány elemeiből építkezik, teljesen általános megfogalmazást szoftverekre nem enged.

Másik megközelítésből vizsgálva a szoftvert elfogadható a nézet, mely szerint PIM készíthető vele. A modellezés során a mérnököknek csak az elemek kapcsolatát, a komponensek felépítését kell megadniuk, a tényleges megvalósítással nem kell foglalkozniuk. Más szóval kódot nem kell írni, tehát a kódnyelv szempontjából vett platformmal nem foglalkoznak a mérnökök, az csak a kódgenerálás után feladat.

2.3.2. Miért van rá szükség

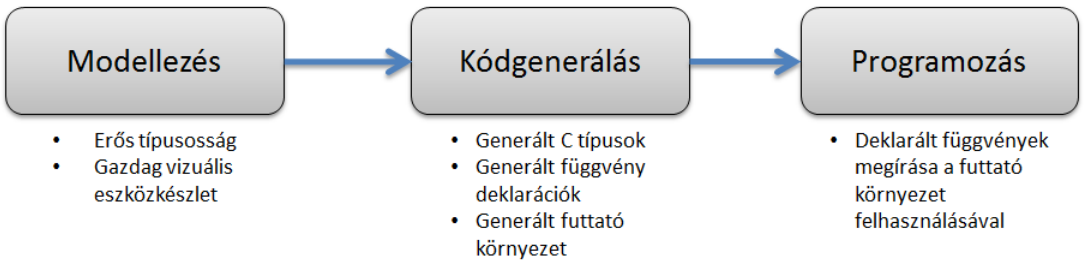
A beágyazott szoftverek fejlesztése során egyre bonyolultabb követelményeket határoz meg az ipar, ez pedig egyre komplexebb feladatok megoldását jelenti. Ahhoz hogy ezeket megfelelően lehessen teljesíteni szükséges a megfelelő programnyelv meghatározása, amely minden szempontból kielégíti az elvárásokat.

Amennyiben a szoftver teljesítményével szemben támasztott elvárásokat figyeljük, egyértelmű, hogy olyan kódnyelv szükséges, amely képes megfelelően kihasználni a hardvert, natívan fut, gyors. Ezeket a követelményeket a mai napig a C nyelv teljesíti a legjobban, így a beágyazott rendszerek fejlesztésénél ennek különböző verzióit használják az iparban, így a thyssenkruppnál is.

Ezzel szemben a fejlesztés és karbantarthatóság szempontjából a C nem a legalkalmasabb nyelv, hiszen nem objektum orientált, az újrahasználhatóságát nehéz biztosítani, valamint a szoftverek bonyolultságából adódóan az egyes részek átláthatósága is kérdésessé válik. A fejlesztéshez kéne tehát egy olyan nyelv, amely rendelkezik megfelelő hierarchiával, könyvtárakkal, amelyekkel a szoftver könnyen megvalósítható. Ilyen nyelv a C++, ami még natívan fut, azonban bonyolultsága miatt nem túl népszerű, valamint a Java, ami pedig nem natívan futtatható programot eredményez, külön virtuális környezetre van szüksége, azaz beágyazott környezetben nem megfelelő.

Beágyazott rendszerek esetén tehát megkerülhetetlen a C használata, azonban szükséges a fejlesztéshez egy magasabb absztrakciós szint. Az AUTOSAR és az AUTSOSAR Architect teszi lehetővé ezt az absztrahálást.

2.3.3. AUTOSAR Architect fejlesztés



2-3 Az AUTOSAR Architect által megvalósított komponens vezérelt fejlesztés

A modellezést megvalósító felületet egy későbbi fejezetben mutatom be, jelenleg elegendő annyit ismerni róla, hogy az elemek grafikus megvalósítását teszi lehetővé, így a kódírás helyett a tényleges implementációra engedi koncentrálni a fejlesztőket.

A modellezés után kódgenerálás útján lehet a modellből futtatható kódot készíteni. A kódgenerálás, ahogy azt írtam, még nem teljesen kiforrott. Ebből következően a modellben megalkotott komponensek esetén csak egy csontvázat (skeletont) ad, a valós működésüket nem készíti el, ezt a fejlesztők feladata kiegészíteni a valós funkcionalitás törzsével. Ezzel szemben a komponensek együttműködését megvalósító RTE generálása teljes funkcionalitású rendszert ad vissza.

3. Hiba analízis

A beágyazott rendszerek fejlesztése gyakran jelenti olyan szoftver készítését, amely a későbbiekben biztonságkritikus rendszerek részét képezi majd. Emiatt ezek megvalósítása, működésének helyessége a rendszer használójára kritikus jelentőséggel bírhatnak, ellenőrzésük több irányból is szükséges.

Az ilyen ellenőrzések egyik fajtája a rendszerek hiba analízise. Ezek a rendszerek és folyamatok biztonságát hivatottak javítani. Ez különösen lényeges azokban a hibakritikus rendszerekben, ahol egy-egy hiba végzetes következményekkel járhat, ezzel magát a rendszert, vagy akár emberéleteket is veszélyeztetve.

A rendszerek ilyen elemzésének célja a bennük rejlő potenciális hibák azonosítása, valamint a veszélyek minősítése, így elősegítve a biztonságot. A cél eléréséhez többféle módszer használható, ezeket pedig rendszerezni lehet.

3.1. Módszerek rendszerezése

Jelen diplomatervben, csak kettő lényeges, alapvető eltérés szempontjából különböztetem meg őket. Az első az ok-okozat, a második pedig a rendszerhierarchia kérdése.

A módszerek közti különbség ok-okozat szempontjából azt jelenti, hogy a módszer hatást, vagy okot keres. Azaz kétféle csoport létezik. Az első az előrelépő, ezek a módszerek azt vizsgálják, hogy amennyiben egy esemény bekövetkezik, annak milyen hatásai lesznek az egész vizsgált rendszerre nézve. A második a visszalépő, az előző csoporttal ellentétes irányban működik, tehát egy adott esemény kiváltó okait keresi.

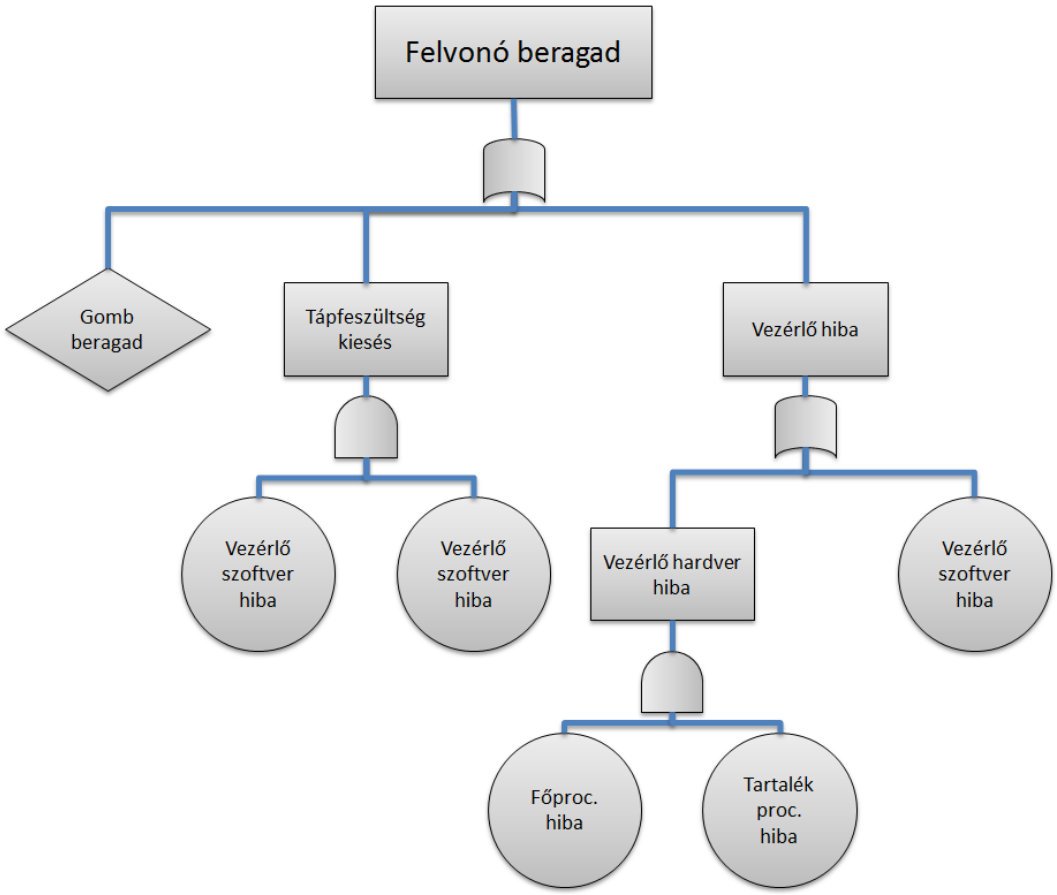
A rendszerhierarchia szempontjából vett megkülönböztetés, a nevéből adódóan a komponensek „bejárásának” irányából fakad. Ezt meg lehet tenni alulról felfele, azaz az alrendszerek, elemi komponensek felől a rendszerszintig haladva, valamint fordítva a legmagasabb szintről elindulva, és lebontva az egész rendszert az elemi komponensekig.

3.2. Hiba analízis módszerek

Több jól ismert, és széles körben használatos veszély analízis technika létezik. Ilyenek az ellenőrző lista, a hibafa, az eseményfa, az ok-következmény analízis, valamint a hibamód és hatás analízis. Ezek közül kettőt mutatok be, a hibafát és a hibamód és hatás analízist.

3.2.1. Hibafa analízis

A hibafa analízis egy visszalépő technika, azaz a veszélyek, meghibásodások okait vizsgálja. Hierarchia szempontjából felülről lefelé haladó, egy rendszerszintű probléma, veszély okait bontja le egészen az alapszintig, az egyszerű rendszerkomponensekig. Ezáltal az analízis a hiba okokat, valamint azoknak a kombinációit deríti fel.



3-1 Hibafa példa

Az elemzés eredménye egy grafikusan megjelenített fa struktúra. A 3-1 ábra egy ilyen analízis eredményére mutat példát. Legtetején a vizsgált veszély, probléma áll. Ehhez csatlakoznak hozzá különböző Boole-algebrai kapcsolatokkal az őt kiváltó köztes okok, valamint alapszintű események, amik tovább nem bonthatók, azaz elemi események a vizsgált rendszer szempontjából.

Az ábrán látható fát különböző elemek építik fel. Négyzetekben láthatók a közbenső, valamint a legfelső szintű (vizsgált) események. Rombuszban olyan események, amelyeket nem vizsgálunk, körökben pedig az elemi események. Ezek összekapcsolásánál logikai kapuk (ÉS illetve VAGY) láthatók.

Az analízisnek két fajtája van, a kvantitatív, valamint a kvalitatív. Utóbbi a hibák jelentőségét vizsgálja, olyan szempontból, hogy amennyiben eltüntetjük a köztes állapotokat, és azok okozóit úgynevezett vágatokba rendezve egy VAGY kapun keresztül a vizsgált veszélyhez kötjük, akkor megállapítható, hogy egy-egy vágatban szereplő elemi esemény milyen jelentős. A vágatok állhatnak egy eseményből, vagy több esemény ÉS kapcsolatából is. Ezek közül azok jelzik a nagyobb sérülékenységet, ahol kevesebb az elemi esemény, azaz a legnagyobb sérülékenységet az egyedül álló események jelenthetik.

A kvantitatív vizsgálat esetén a rendszerszintű veszélyek valószínűségét elemezzük. Az elemi eseményekhez hozzárendeljük az előfordulásuk esélyének valószínűségét, majd ezeket az értékeket az ÉS kapuk mentén szorozva, valamint a vagy kapuk mentén összeadva megkapjuk a rendszerszintű hiba esélyét.

3.2.2. Hibamód és hatás analízis

A hibamód és hatás analízis (Failure Modes and Effects Analysis – a továbbiakban FMEA), az első szisztematikus hiba analízis módszerek egyike volt. Az 1950-es években fejlesztették ki katonai eszközök megbízhatóságának vizsgálatára. Az adott rendszer lehető legtöbb komponensének, alrendszerének hiba lehetőségeit, valamint ezek egész rendszerre gyakorolt hatását vizsgálja.

Az FMEA módszere a potenciális megbízhatósági problémák felfedezését a fejlesztés korai szakaszában teszi lehetővé, amikor még könnyebb lépéseket tenni a hibák kiküszöbölésére, ezáltal növelve a megbízhatóságot.

A metódus kritikus lépése meghatározni, hogy egy eszközben, termékben milyen hiba keletkezhet. Az összes hibamód ugyan nem meghatározható, vagy csak az esetek nagyon kis részében van erre lehetőség, de a fejlesztés során minél több lehetséges meghibásodást azonosítani kell.

3.2.2.1. Létjogosultsága

A múltban a gyártási folyamatoknál a gyakran különböző biztonsági tényezőkre hagyatkoztak, ezek segítségével biztosították, hogy a design megfelelően fog működni, ezzel megóvva a végfelhasználót az esetleges hibáktól.

Ez például azt jelentette, hogy az eszközök előrelátható terhelését megbecsülték, és ennek többszörösét kibíróra méretezték őket. Ebben az esetben a biztonsági tényező az volt, hogy hányszor nagyobb terhelést kell kibírnia az adott eszköznek. Ilyen megoldásokat természetesen ma is alkalmaznak, mert hatékonyak a váratlan nagy terheléssel szemben, azonban nem mindig vezetnek megfelelő eredményre. Előfordul, hogy a nagy biztonsági tényező egy túltervezett, megbízhatósági problémákkal rendelkező terméket ad. Emellett a költségeket is növeli, nagyobb teherbíráshoz például egy acélszerkezetnél több acélra van szükség, ami több anyag- és szállítási költséget jelent.

Az FMEA egy olyan eszköz, aminek segítségével a potenciális veszélyek egy termékben felderíthetővé válnak, ezáltal több esetben is használható [2]:

- A termék, folyamat követelményeinek formálása, a hibák valószínűségének csökkentése érdekében
- Az ügyfelektől érkező követelmények kiértékelése, hogy azok esetleg hibákhoz vezetnek-e
- Azon tervezési jellemzők kiiktatása, hatásaik minimalizálása, amik a hibák előfordulásához hozzájárulnak
- Módszerek, folyamatok fejlesztése, amik a hibák kivédésének sikerességét vizsgálják

- A potenciális hibák nyomon követése, kezelése
- Annak biztosítása, hogy a hibák, amik előfordulhatnak, nem okoznak sérülést a felhasználó személyben/termékben/folyamatban

3.2.2.2. FMEA típusok

Az FMEA több fajtája különböztethető meg, az alapján, hogy az adott típus a rendszer mely részének analízisére fókuszál. Az alábbi felsorolás ezt adja meg [2]:

- System – a globális rendszerszintű funkciók
- Design – alrendszerek és a komponensek
- Process – gyártási és összeszerelési folyamatok
- Service – szolgáltatási funkciók
- Software – szoftver funkciók

A feladat során Design típusú FMEA támogatás megvalósítás a cél, egy adott rendszer komponenseinek bemeneteit vizsgálom, hogy ezek miként romolhatnak el, és a komponensre ez milyen hatást gyakorol.

3.2.2.3. FMEA használata

Az FMEA az egész fejlesztést végig kíséri, így mindig az aktuális helyzetet kell mutatnia. Mivel változik a termék, folyamat, a változtatások pedig új hibák lehetőségét vetik fel, ezért az FMEA ellenőrzése, frissítése az alábbi esetekben szükséges [2]:

- Új termék, folyamat bevezetése, tervezése (a tervezés elején)
- Az üzemeltetési körülmények (ahol elvárt a helyes működés) változása
- A dizájnban történt változás
- Új szabályozás bevezetése
- Felhasználói visszajelzések feldolgozása

3.2.2.4. FMEA folyamata

Az FMEA folyamata alapvetően jól meghatározott lépésekből áll [2]:

1. A termék funkcióinak, folyamatainak meghatározása
2. Blokk diagram készítése, ami a komponensek közti összeköttetéseket adja meg
3. FMEA adatlap készítése (a rendszer felépítését írja le, komponensek, tulajdonságaik)
4. Ezek hibamódjainak meghatározása, a hibamódokat priorizálni kell az alapján, hogy mennyire valószínű az előfordulásuk
5. A hibamódok hatásainak leírása, mindegyikhez a végső hatást is meg kell határozni (pl. sérülést okoz), valamint hogy mit lehet tapasztalni a hiba bekövetkezésekor

6. A hatásokhoz súlyosságot jelző rang rendelése. Az iparban elfogadott standard skála 1-től 10-ig terjed. 1 jelzi azt, hogy nincs hatás, 10 pedig a nagyon komoly, rendszert károsító hatás, figyelmeztetés nélkül. A rangsor segítségével lehet priorizálni, így elsőként a nagy problémákra fókuszálva
7. A hiba előfordulás okának azonosítása, ezekhez valószínűséget kell rendelni. Elfogadott skála az 1 (nem fordul elő) -10 (elkerülhetetlen)
8. A hibák megelőzésére szolgáló megoldások azonosítása, a hiba észrevételének valószínűségét is meg kell adni
9. Az előző három skála értékéből adódó eredmény segítségével lehet a hibákat priorizálni. A Risk Priority Numbers(RPN) a hiba végső prioritása
   RPN = (hatás súlyosság) x (előfordulás valószínűség) x (észlelés valószínűség)
10. A nagy RPN-nel rendelkező hibák kijavítására ajánlott intézkedések meghatározása, ezek lehetnek különböző megfigyelések, tesztelések, stb.
11. Hibák javítása, majd az FMEA frissítése

Az FMEA folyamata a korábban leírt csoportosítások alapján tehát egy előrelépő, alulról felfele haladó módszer. A legkisebb részegységek hibáinak azonosításával kezdődik, majd ezek hatását vizsgálja az egész rendszerre vetítve.

3.2.2.5. Változtatás az FMEA folyamatán

A thyssenkruppnál dolgozó biztonsági analízissel foglalkozó mérnökök a bővítmény első, még korai szakaszban tartott demó bemutató után kérték, hogy a hibamódok lehetőség szerint ne 0-10 skálán megadott súlyosság értékekkel és előfordulási valószínűséggel legyenek jellemezve, hanem az autóiparban szokásosan használt Automotive Safety Integrity Level [3] [4] (ASIL) értékekkel.

Integrált biztonsági szint	Hiba valószínűsége óránként	Legrosszabb esetben bekövetkező esemény
QM	-	Csak dokumentálás miatt szükséges
ASIL A	$< 10^{-6}$	Néhány ember kisebb sérülést szerezhet
ASIL B	$< 10^{-7}$	Néhány ember súlyosan sérülhet vagy egy ember halála
ASIL C	$< 10^{-7}$	Több ember halála
ASIL D	$< 10^{-8}$	Számos ember halála

3-2 ASIL értékek besorolása

„Egy biztonsági rendszer 100%-osan funkcionálisan biztonságos, ha a véletlen meghibásodás, a közös meghibásodás és a szisztematikus meghibásodás nem vezet el a biztonsági rendszer hibás működéséhez, és nem eredményez emberi sérülést, vagy halált, környezetszennyezést, illetve anyagi károkat. Teljes mértékű funkcionális biztonság nem létezik, ugyanakkor az ilyen jellegű események bekövetkezésének várható/megengedhető gyakoriságát az úgynevezett SIL és ASIL értékekkel jellemezhetjük.

A veszélyes szituációk szisztematikus elemzéséből meghatározhatóak a biztonsági célok és az azokhoz kapcsolódó autóipari biztonsági integritási szintek (ASIL). Az ASIL szintek meghatározásához a következő faktorok hatását kell mérlegelni: súlyosság a baleset bekövetkezése esetén, a kitettség mértéke és szabályozhatóság (irányíthatóság).” [4]

- ASIL = Súlyosság × (Kitettség × Szabályozhatóság)

A hibának kitettség mértéke, és annak szabályozhatósága a relatív valószínűséget határozzák meg, ezt kombinálva a súlyosságának mértékével adódik az ASIL érték. Ahogy a 3-2 ábra mutatja az értékek ASIL A–tól ASIL D–ig terjednek, valamint van egy ötödik típus is, a Quality Management (QM). Ez utóbbi az olyan hibás működéseket jellemzi, amelyik az ASIL értékeket definiáló szabvány szerint nem igényel biztonsági beavatkozást.

Az értékek definíciója alapján látható, hogy a megvalósítandó analízis technikába ez a hiba jellemzési mód megfelelően beilleszthető.

4. AUTomotive Open System ARchitecture – AUTOSAR

A feladat alapkoncepciója tehát az előbbiekben ismertetett FMEA támogatásának megvalósítása. Magát a módszert autóipari környezetben kell alkalmazni, ehhez pedig az itt használt szabványt kell megismerni közelebbről.

4.1. Történelem [5]

A járműipar fejlődésével a modern elektronikus eszközök elértek egy olyan szintre, hogy ahhoz, hogy a kliensek elvárásai és az eszközökkel szemben támasztott jogszabályi követelmények betarthatók legyenek, komoly technikai újításokat kell bevezetni. Ezek az újítások különösen fontosak a járműgyártók, és fő beszállítóik számára, akik gyakran egymásnak ellentmondó igényeket kapnak különböző forrásokból:

- Jogszabályok, főként környezeti, és biztonsági tényezőkre fókuszálva
- Végfelhasználók/utasok, a kényelem és élvezeti értékre helyezve hangsúlyt

Az akadályok könnyebb leküzdése érdekében a gyártók, és beszállítóik összeálltak, hogy biztosítsák a fejlesztési alapfunkciók együttműködését, elősegítve az innovatív megoldások megvalósítását. Létrejött az AUTOSAR fejlesztési partnerkapcsolat, hogy létrehozzanak egy szabványt az alábbi célokkal:

- Az elektronikus vezérlő egységek (ECU) szoftver funkcióinak szabványosítása
- Különböző jármű típusokra méretezhetőség, használhatóság
- A szoftverek újrahasználhatóságának, hordozhatóságának támogatása
- Nyílt architektúra létrehozása
- Nagy megbízhatóságú rendszerek készítése
- A nemzetközi autóipari technológiák, szabványok támogatása
- A partnerek közti együttműködés elősegítése

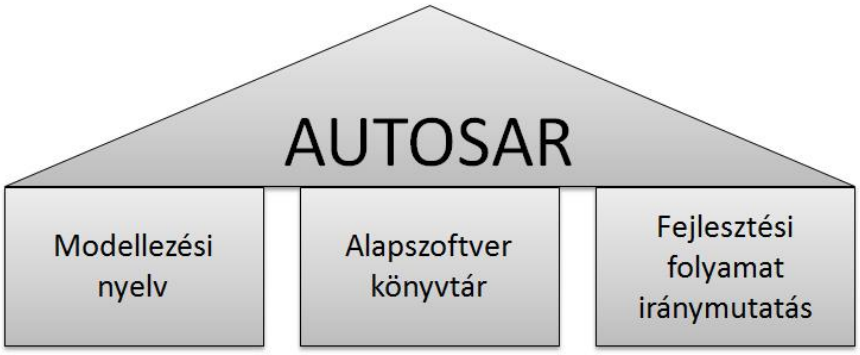
Megalkották az AUTOSAR szabványt, ami az autógyártás minden területét lefedi. Az első szabványt 2004-ben adták ki, azóta folyamatosan fejlesztik, javítják. Jelenleg a 4.3-as kiadásnál járnak. A diplomaterv készítése során a 4.0.3-al dolgoztam.

NoC	3.1.1	3.1.2	3.1.3	3.1.4	3.1.5	3.2.1	3.2.2	3.2.3	4.0.1	4.0.2	4.0.3	4.1.1	4.1.2	4.1.3	4.2.1
3.1.1	0	66	66	317	582	7119	8384	9223	31251	35485	39606	52322	53138	53827	71287
3.1.2	0	0	20	274	539	7084	8349	9188	31265	35499	39620	52336	53152	53841	71301
3.1.3	0	0	0	274	539	7084	8349	9188	31265	35499	39620	52336	53152	53841	71301
3.1.4	0	0	0	0	317	6865	8134	8986	31357	35596	39722	52441	53257	53946	71405
3.1.5	0	0	0	0	0	6675	7944	8801	31567	35696	39822	52551	53367	54056	71555
3.2.1	0	0	0	0	0	0	1422	2454	34922	38905	42913	55821	56649	57342	75228
3.2.2	0	0	0	0	0	0	0	1109	35842	39805	43774	56637	57470	58162	76148
3.2.3	0	0	0	0	0	0	0	0	36326	40156	44125	56925	57742	58398	76359
4.0.1	0	0	0	0	0	0	0	0	0	5783	12235	29690	30696	31570	55690
4.0.2	0	0	0	0	0	0	0	0	0	0	7026	24900	25927	26817	51706
4.0.3	0	0	0	0	0	0	0	0	0	0	0	18662	19765	20743	47619
4.1.1	0	0	0	0	0	0	0	0	0	0	0	0	1372	2472	35067
4.1.2	0	0	0	0	0	0	0	0	0	0	0	0	0	1104	33979
4.1.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	33241
4.2.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

4-1 Az AUTOSAR Szabvány iterációi közti különbségek hőtérképe [6]

A verziók átdolgozásának mértékét jól mutatja a 4-1 ábra, láthatóan egy-egy minor verzió váltáskor is ezres nagyságrendű változtatásról van szó, a major verziók esetében ennek sokszorosa. Ezzel magyarázható, hogy a cégek nem feltétlenül tudják azonnal lekövetni a szabvány változásait, és használnak korábbi szabvány megvalósításokat.

4.2. Szabvány



4-2 Az AUTOSAR szabvány által nyújtott fejlesztési irányelvek

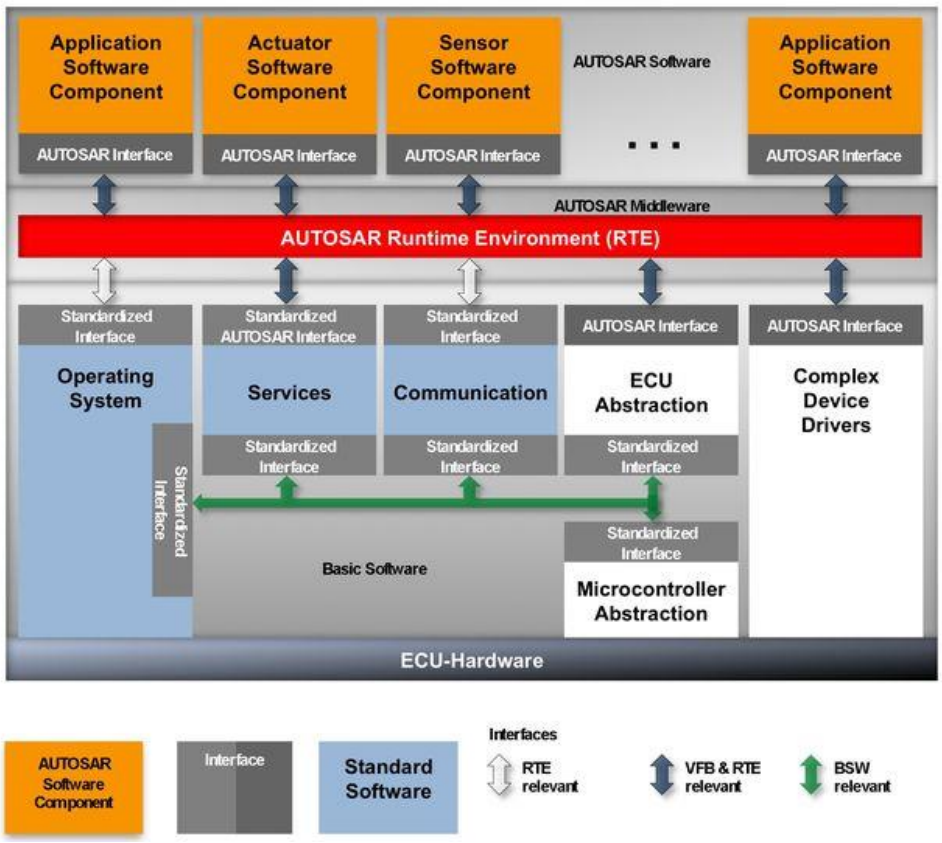
Az AUTOSAR tehát egy szabványcsalád [7], mely három fő részre osztható, ezt mutatja a 4-2 ábra.

Az első rész egy komponens orientált modellezési nyelvet határoz meg, ez elsősorban a később bemutatott Software Component Template-et tartalmazza. Ennek segítségével lehet meghatározni az első fejezetben tárgyalt modellt, amelyből a szoftver generálás következik.

A második rész egy olyan alapszoftver könyvtár, amely segítségével az autóipari alkalmazások programozása válik egyszerűbbé, gyorsabbá.

A harmadik rész pedig az autóipari fejlesztési folyamat leírását tartalmazza, benne a megfelelőség vizsgálatára vonatkozó elfogadási tesztekkel. Javaslatokat tesz modellezési, fejlesztési és integrációs lépések folyamatba szervezésére.

4.3. Architektúra



4-3 Az AUTOSAR ECU architektúrája [8]

A 4-3 ábra a szabvány elektromos szabályzó egység szoftvereinek (Electrical Control Unit – a továbbiakban ECU) architektúráját mutatja. Jól láthatóan több elkülöníthető rétegből épül fel.

A legalsó réteg maga az ECU. A közvetlenül hozzá csatlakozó mikrokontroller absztrakciós réteg a hardver vezérlés elérését biztosítja, ezzel a magasabb szintű szoftverekből a mikrokontroller regisztereinek elérését indirektté téve elrejti az alacsony szintű részleteket [9].

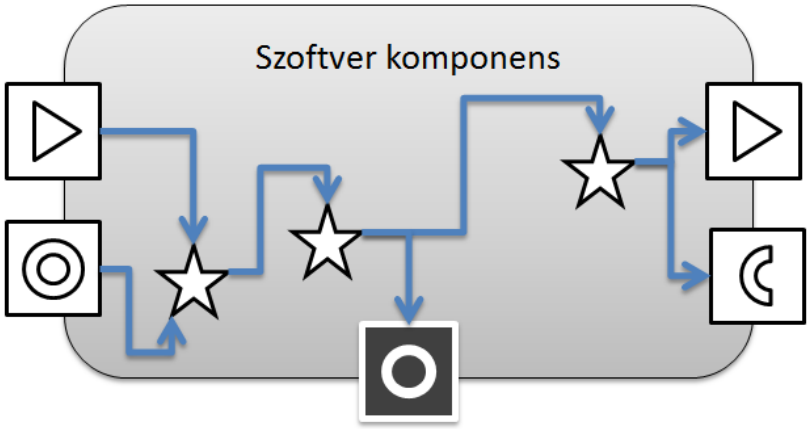
Az alap szoftver (Basic Software) [9] az a szabványosított réteg, ami az AUTOSAR komponenseknek biztosít olyan szolgáltatásokat, amik szükségesek a funkcióik megfelelő működéséhez. Ilyen például a kommunikációs rétegek – pl. CAN, LIN –, vagy a rendszer szolgáltatások támogatása – pl. diagnosztikák, protokollok – is.

A középen lévő (pirossal jelölt) AUTOSAR Real Time Environment [10] felelős az ECU-n belüli, és kívüli – pl. CAN, LIN buszon megvalósított – kommunikáció összekötéséért.

Legfelül látható (sárgával jelölt) a Software Component réteg, itt találhatóak meg az alkalmazásokat magas szinten megvalósító komponensek. A feladat során az ebben található elemek analízise a cél, ezért ezeket a következő alfejezetben részletesen bemutatom.

4.4. Software Component Template

Az architektúra legfelső szintjéhez tartozó szoftver komponensek jelentik az ECU működésének magas szintű absztrakciós rétegét. Sok fajtája van, a munkám során az AtomicSwComponentType típusúakkal foglalkoztam. Ezek a réteg legkisebb egységei.



4-4 A szoftver komponensek felépítése

Réteges felépítésűek, mindent tartalmaznak a saját működésükről, tartalmazott elemeikről és a külvilág felé nyújtott kapcsolási lehetőségekről. A 4-4 ábra egy ilyen komponens felépítését mutatja, az AUTOSAR szabvány jelöléseit használva.

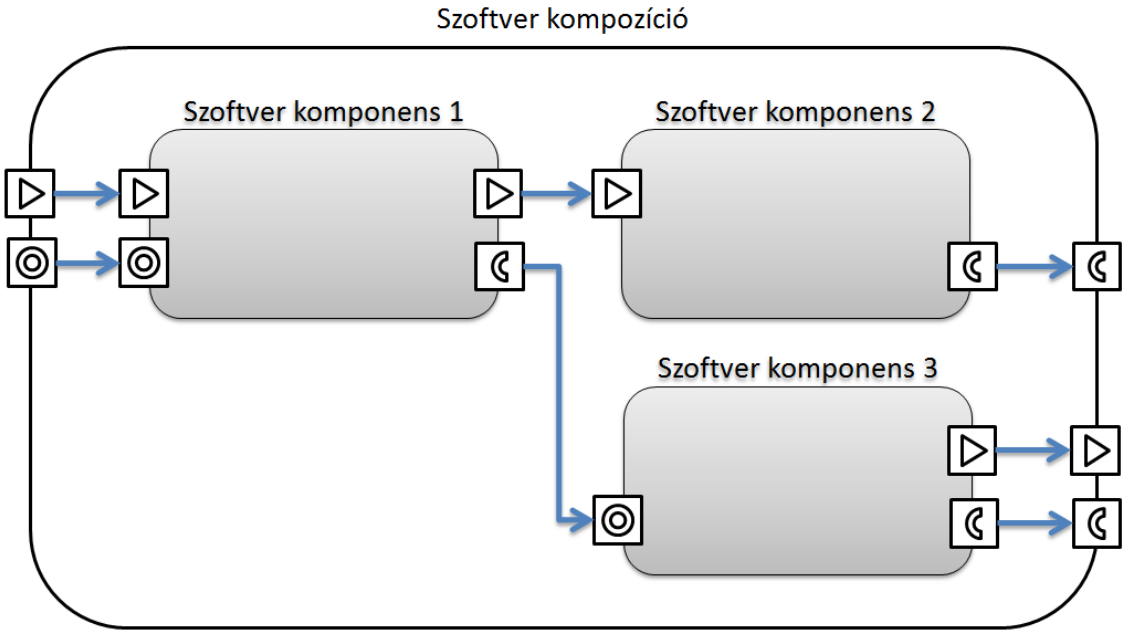
A külvilág felé nyújtott kapcsolódási pontok a portok. Ezek a szabványban a PortPrototype-ok. A működés iránya szerint vannak igényelt (required), valamint felkínált (provided) portok. Azt, hogy ezeken milyen adatok haladhatnak keresztül, azok interfésze adja meg, ami a PortInterface nevet viseli. A feladat során ezek közül kettővel foglalkoztam:

- Adatfolyam típusú (az ábrán a háromszög alakú portok)
- Vezérlési folyam típusú (az ábrán a kör és félkör alakú portok)

Az adatfolyam két komponens közti adattovábbításra szolgál, küldő-fogadó kapcsolatot valósít meg. A vezérlési folyam pedig különböző szolgáltatások elérését támogatja, rajta argumentumokat lehet átküldeni, kliens-szerver kapcsolatot létesít.

Ebből fakadóan az adatfolyam típusú portok esetén a kínált (provided) porton a komponens által szolgáltatott kimeneti adatok érhetőek el, a vezérlési folyam típusún pedig a komponens által megvalósított műveletek, metódusok. Az igényelt (required) portokon értelemszerűen a szükséges bemeneti adatok, valamint az elérni kívánt függvények biztosítása szükséges a megfelelő működés eléréséhez.

A portokon továbbítható adatokat a fejlesztő szabadon meghatározhatja, tulajdonképpen bármilyen konstrukció továbbítása lehetséges. A szabványban erre az implementációs adat típus (ImplementationDataType) ad lehetőséget, aminek a segítségével definiálhatók egyszerű értékek, tömbök, struktúrák stb. adattípusként.

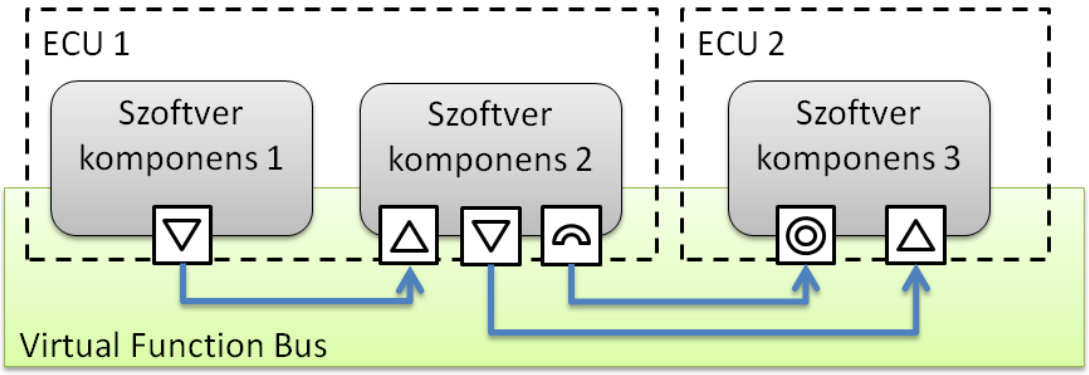


4-5 Kompozícióban összekapcsolt szoftver komponensek

A komponensek a portjaikon keresztül kapcsolódhatnak, ez szoftver kompozíciókban (CompositionSwComponentType) valósul meg, a 4-5 kép egy ilyen összekapcsolást mutat be. Ennek segítségével a komplexebb részegységek működése írható le.

Amint már említettem, a komponensek egy-egy funkcionalitást zárnak egységbe. Ez azt jelenti, hogy a komponens tartalmaz függvényeket, amik futásának eredményét a portjain közli kimenetként. Ezeket a függvényeket hívja a szabvány futtatható entitásnak (RunnableEntity), a 4-4 kép csillag elemei jelzik ezeket. A komponensek réteges felépítésének megfelelően az ilyen függvényeket belső viselkedésekben (InternalBehavior) tárolja.

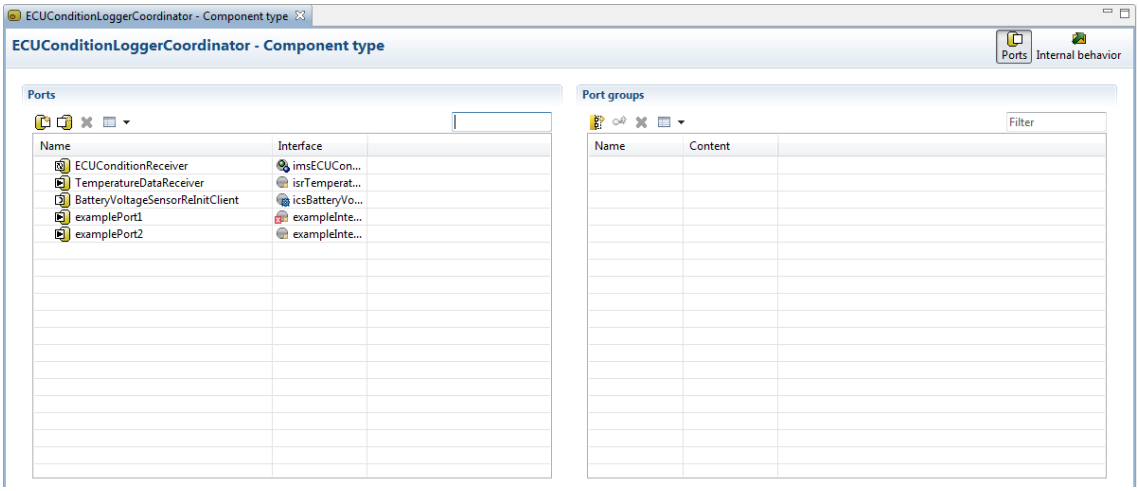
A futtatható entitások az adott elemen belül akkor indulnak el, és futnak le, amennyiben egy a fejlesztők által definiált trigger elindítja őket. A triggerek a modellben események (eventek) formájában jelennek meg. Az eseményeknek sok fajtája van definiálva, a feladat megértéséhez elegendő kettő ismerete. Az első ilyen a timing evenet, amely bizonyos időközönként automatikusan jelez, és elindítja a hozzá kapcsolódó futtatható entitást. A második az operation invoked event. Ez az esemény, ahogy a neve is mutatja akkor triggerel, ha egy függvényhívás érkezik.



4-6 Különböző ECU-kon megvalósított szoftver komponensek összekötése Virtual Function Bus segítségével

Mivel a szoftver komponensek az ECU-k csak egy funkcionalitását zárják egységbe, egy ECU egyszerre több ilyet is tartalmazhat, ezt mutatja a 4-6 ábra.

4.5. AUTOSAR a gyakorlatban



4-7 Komponens modellezéshez használt szerkesztő felület

A 4-6 ábrán látható, ECU-kat összekötő virtuális funkció busz (Virtual Function Bus - VFB) a modellezés alatt olyan szempontból érdektelen, hogy a hardveren melyik komponenst melyik ECU valósítja meg. Így a munka során tényleg csak a komponensek funkcióinak megvalósítására kell koncentrálni, a fejlesztés egyszerűbb.

A4-7 kép az AUTOSAR Architect szoftver komponens modellezésére használt szerkesztő felületet szemlélteti. Látható, hogy a portok, valamint a belső működés megadására külön szerkesztő oldalak érhetőek el, így könnyen átláthatóvá téve a modellezés folyamatát.

5. Eclipse technológiák

Amint már említettem a bevezetőben az AUTOSAR Architect szoftver, amelyhez az analízist támogató bővítményt kellett elkészítenem, egy Eclipse platform alapú megoldás. Emiatt elengedhetetlen volt, hogy megismerkedjek az Eclipse által nyújtott különböző fejlesztői eszközökkel, technológiákkal. A feladat során felhasznált főbb technológiák, amelyeket részletesen is bemutatok:

- Eclipse architectúra
- Eclipse plug-in architektúra
- Eclipse Modeling Framework
- JFace
- Futtatási konfiguráció (Run configuration)

5.1. Eclipse architektúra

Az Eclipse fejlesztői környezet segítségével lehetőség van úgynevezett Rich Client Platform (a továbbiakban RCP) alkalmazások létrehozására. Ezek önálló asztali alkalmazásokat jelentenek, a thyssenkrupp által fejlesztett AUTOSAR Architect is egy ilyen RCP alkalmazás, amely alap szoftvere több mint 600 bővítményt tartalmaz.

5-1 Az Eclipse architektúra

A 5-1 ábra az Eclipse architektúrájának felépítését és az RCP ezen belül elfoglalt helyét mutatja be. Az összes Eclipse alkalmazáshoz szükséges az Eclipse Runtime Platform, ez tartalmazza a bővítmények futtatásához szükséges információkat, funkciókat, többek között a frissítésekért, valamint a felhasználóval történő kommunikációért felelős.

Erre épül rá az RCP. Maga az Eclipse IDE (Integrated Developement Environment) is egy specifikus RCP alkalmazás.

Az IDE-re lehet ráépíteni a nyelvfüggő fejlesztői környezeteket, mint a Java Developement Toolkit (JDT), valamint a C/C++ Development Tools (CDT), amik Java és C programok készítésére használhatók, de kiterjeszthetőek nagyjából bármivel.

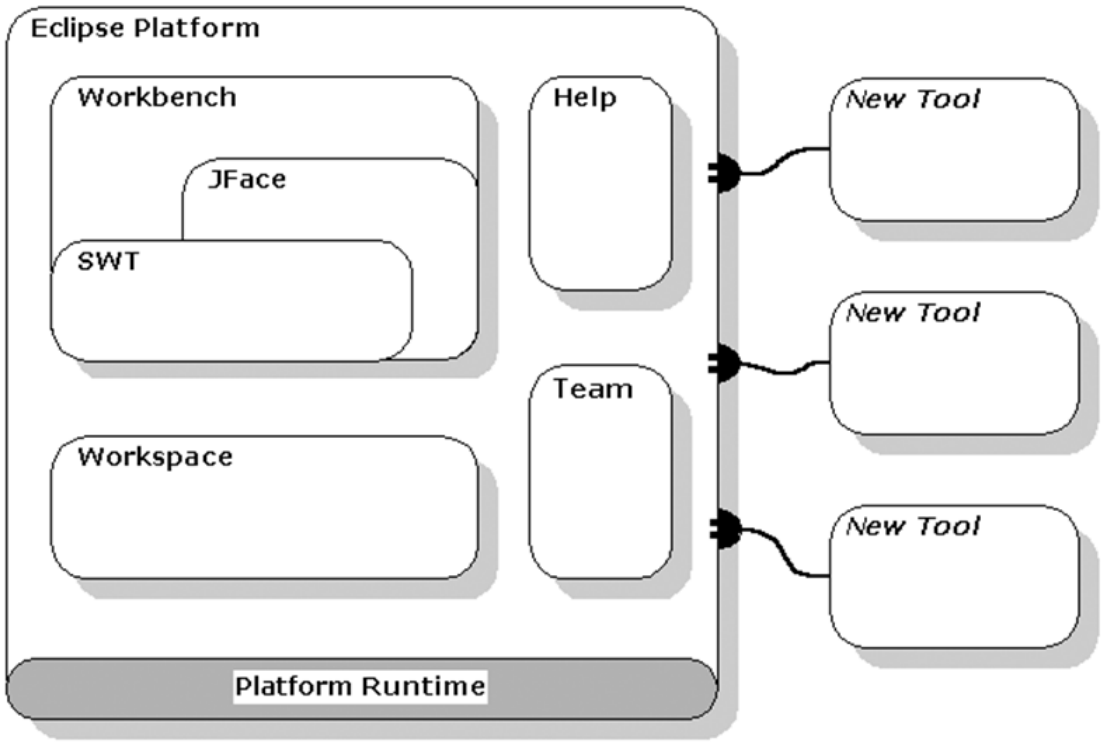
Szintén az IDE-re épül a Plugin Development Environment (PDE), amivel további bővítményeket készíthetünk RCP alkalmazásokhoz – ezeket reprezentálja az ábrán szürkével jelölt, Contributed plug-ins cimkéjű halmaz.

Ez tehát azt jelenti, hogy a bővítményeket az Eclipse platformhoz csatlakoztatva, az erre az alapra készített alkalmazások funkcionalitása kibővíthető.

5.2. Eclipse plug-in

A beépülő modulok, vagy bővítmények (plug-in) tehát az alap szoftverünk funkcionalitásának kibővítésére szolgálnak. Alapvetően a PDE, a JDT, és a CDT is több plug-in-ból épül fel.

Egy plug-in a lehető legkisebb installálható egység egy-egy alkalmazáshoz.



5-2 Eclipse plug-in architektúra [11]

Az 5-2 ábra illusztrálja, hogy az Eclipse platformhoz miképpen csatlakoznak az új plug-in-ok. Ezek a New Tool névre hallgató objektumok, amik „villásdugókkal” csatlakoznak a platformhoz. A villásdugók végeit (ahol az aljzat van) Extension point-nak [12] hívják.

Ezekhez csatlakoznak az Extension-ök, amik az Extension point-ot definiáló plug-in-ok működését egy meghatározott interfészen keresztül kibővítik, együttműködnek velük. Egy plug-in tetszőleges számú Extension-t, valamint Extension point-ot definiálhat, és a saját maga által definiált extension point-ot is kiterjesztheti, például egy default viselkedés implementálásával.

Az Extension point-oknak több fajtája van. Egyik például, amelyik az Extension-től nem vár semmiféle kódot, nincs szüksége rá, csak adatokat szolgáltat.

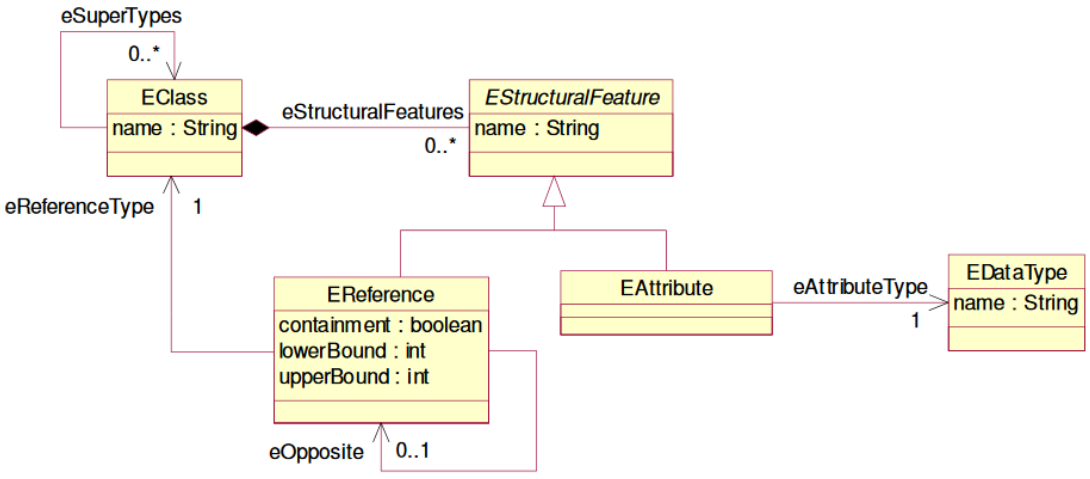
Egy másik fajta segítségével az adott plug-in valamely részének viselkedését lehet felülírni, ilyen megoldás például az, hogy a JDT esetén a kódszínezőt/kódformázót egy külső plug-in hozzáadásával le lehet cserélni.

A feladat megoldásához egy olyan bővítményt kellett megírnom, ami egy, az AUTOSAR Architect egyik szerkesztő bővítményében definiált Extension Point-hoz csatlakozik. Ez lehetővé teszi, hogy adott típusú elemekhez – esetemben AtomicSwComponentType-hoz – editorokat regisztráljak, ezáltal ezek az elemek megnyithatók és szerkeszthetők a készített editor plug-in segítségével.

5.3. Eclipse Modeling Framework – EMF

Az EMF projekt egy modellezési keretrendszer és kódgenerálási eszköz, aminek segítségével az elkészített modellekből használható Java kód generálható. Ez az Eclipse modellvezérelt szoftverfejlesztésének alapja [13].

A fejlesztés során létre kell hozni ún. metamodelleket, amik leírják a valós modellek statikus struktúráját: milyen elemek alkothatják, ezeknek milyen tulajdonságaik vannak, milyen kapcsolatok lehetnek közöttük, azok milyen számossággal stb. Ez ahhoz hasonlítható, mintha C-ben létrehoznánk egy struktúrát, aminek megadnánk, hogy milyen tulajdonságai vannak, milyen egyéb struktúrákra mutat, stb., majd példányosítanánk.



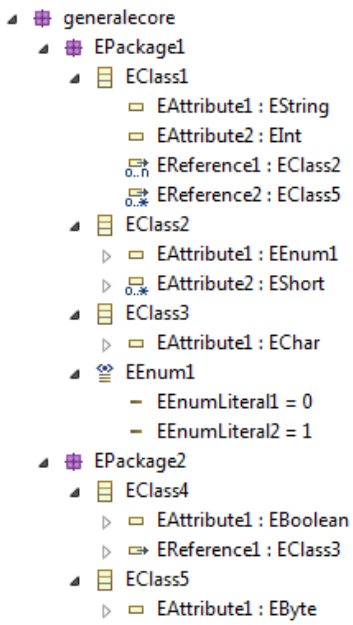
5-3 EMF metamodell általános felépítése [14]

A 5-3 ábra egy EMF osztály általános felépítését mutatja. A baloldalon maga az osztály, aminek meg lehet adni attribútumokat és referenciákat, melyek egy közös ősből, az EStucturalFeature-ből származnak. Az attribútumok különböző adattípusokra mutathatnak. A metamodellben lehetőség van saját típus létrehozására is. A nyilakon elhelyezett jelölések a számosságra utalnak. A 0..* a 0 vagy annál több elemre utal, * esetén nincs megszabva felső korlát.

Az elemekben azok lényegesebb tulajdonságai vannak feltüntetve az ábrán. A name egyértelmű, az elem nevét jelenti, a referenciánál látható lower- és upperBound az osztályhoz tartozó referenciák minimális, és maximális számát adják meg, a containment pedig, azt adja meg, hogy a hivatkozott elem a hivatkozó gyereke-e. Az eOpposite kétirányú kapcsolat esetén adja meg a referencia a hivatkozó oldalát, ez eReferenceType pedig azt, hogy a referencia milyen típusra mutat.

Az osztálynál az eSuperType-pal ősosztály határozható meg az ecore modellben létrehozott, vagy más ecore modellből importált osztályokból. A Java nem engedi több ősosztály megadását, abban az esetben, ha több helyről származtatott osztály létrehozása a cél, interfészek megadására van lehetőség. Ezt az EMF modellben eClassifier-ek megadásával lehet elérni. Alapvetően magában a modellben az ősosztály is meg fog jelenni mint eClassifier, de – mivel az eSuperType-ként van definiálva, – a generált kódban egyértelműen elkülönül az ősosztály és interfészek.

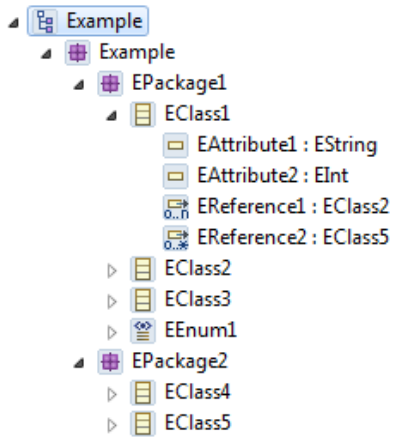
Az EMF metamodellek alapvetően ilyen elemekből épülnek fel, a 5-4 kép egy leegyszerűsített ecore modellt mutat az Eclipse EMF szerkesztőjében megjelenítve.



5-4 EMF metamodell megvalósíttás az EMF szerkesztő felületén

Egy modell több package-et tartalmazhat, ezeken belül is lehet további package-eket definiálni. Ezt követően lehet létrehozni osztályokat, Enum-okat.

A képen az EAttributeX, és EReferenceX néven jelennek meg az osztályok tulajdonságai, referenciái, a mellettük levő megnevezés a típus-, és osztálydefiníciójukra vonatkozik. Az EClass2 EAttribute2-nél látszik, hogy a package-en belül létrehozott Enum is definiálható típusként.



5-5 A példa modell generátor modellje

A 5-5 ábra a metamodellből készített generátor modellt ábrázolja. Láthatóan ez is tartalmazza az ecore-ban létrehozott összes elemet. Ez a modell a kód generálásának beállításaihoz nyújt opciókat, mint például:

- RCP, vagy IDE applikáció készüljön
- Az interfészek EMF, vagy egyszerű Java típusúak legyenek
- A generált editorban a modell elérése hogyan történjen
- A modellelemek szerkesztése hogyan valósul meg

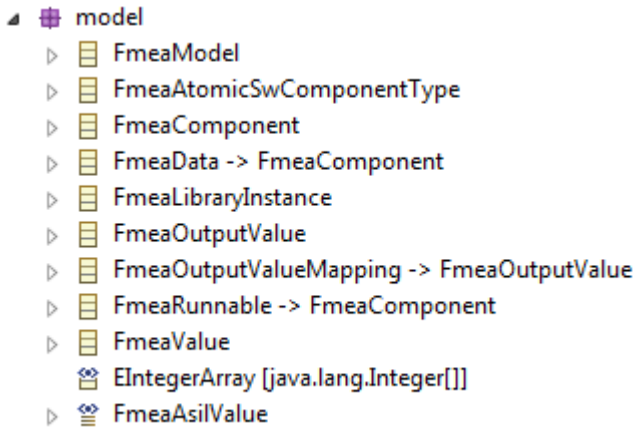
Ez tulajdonképp az ecore modellt megvalósító kód generálását leíró modell.

Az EMF kódgenerálás alatt alapvetően három féle plug-in létrehozását szokták érteni. Az első a modell Java kódját tartalmazza, ami lehetővé teszi elemeinek létrehozását, módosítását. A másik kettő a modell szerkesztését támogató plug-in. Ezek nagyon egyszerű szerkesztőt nyújtanak a felhasználóknak, a valós alkalmazásokban azonban jóval felhasználóbarátabb, szélesebb funkcionalitást biztosító felületekre van szükség, ezért inkább a saját megoldást választja mindenki. A jobb automatikusan generált felületek kidolgozásán folyamatosan dolgoznak, jó példa erre az Eclipse EMF client platform [15].

A plug-in megvalósítása során csak a modell generálást használtam, szerkesztéshez saját plug-in készítését választottam. Az EMF segítségével generált modell kód három package-t tartalmaz, benne a generált java osztályokkal:

- Az osztályok interfészeit tartalmazó package
- Az osztályok implementációit tartalmazó package
- Az AdapterFactory-k, funkciójuk, hogy a létrehozott osztályokhoz adaptereket készítsenek, amikhez különböző értesítéseket lehet kapcsolni. Ezt a feladat során nem hasznosítottam.

5.3.1. Használt ecore modell



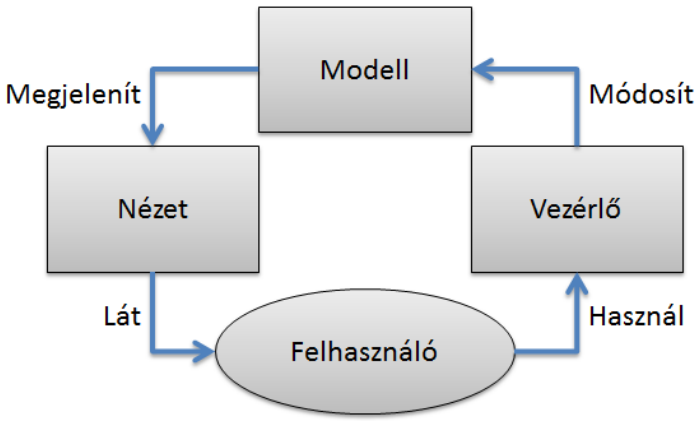
5-6 A feladat megvalósítása során megvalósított metamodell

Az EMF fejlesztés első lépése egy ilyen metamodell megvalósítása. A metamodell a későbbiekben az igények, feladat változása esetén változtatható. A feladat során definiált metamodellt mutatja az 5-6 kép, ezt a későbbiekben részletesen bemutatom.

5.4. Eclipse User Interface fejlesztés – Eclipse JFace

Egy jól használható szerkesztő létrehozásához szükséges volt a felhasználói felületek különböző elemeinek megismerése, használata. Ehhez az Eclipse JFace API-val ismerkedtem meg, ami az SWT (Standard Widget Toolkit) felett álló réteg. Az SWT használható alapvetően operációsrendszer függetlenül Java felhasználói felületek létrehozására. A JFace ugyanezt támogatja magasabb szinten, az SWT elemeivel együttműködve [16].

Ez az API elősegíti a Model-View-Controller minta [17] követését a felhasználói felületek fejlesztésben. Ez egy olyan megoldást jelent, amelynél a felhasználó számára a valós modell tulajdonképpen rejtve marad, annak csak egy része jelenik meg számára egy adott nézetben (View), a modell szerkesztését pedig egy vezérlő (Controller) teszi lehetővé. Az 5-7 ábra ennek az architektúra mintának a szemléltetése.



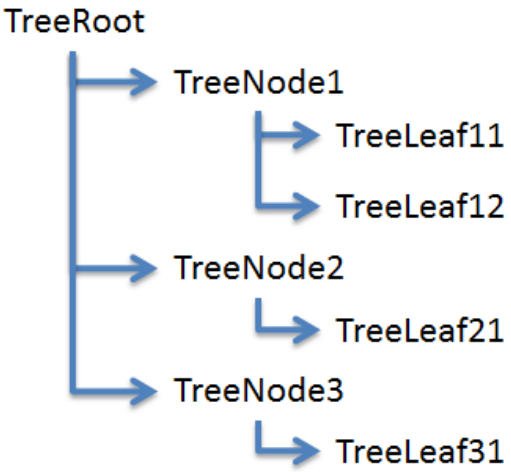
5-7 A modell - nézet - vezérlő architektúra vizualizációja

Az API-ból legfőképp a TableViewer, valamint a TreeViewer elemeket, valamint az ezekhez szükséges egyéb funkcionalitásokat használtam, ezek segítségével valósítottam meg a később részletesen bemutatott szerkesztőt. E két osztály segítségével létrehozhatók táblázatok, valamint fák, a bemenetüknek pedig egy objektumot adva a megjelenítést és a szerkesztést is megoldják.

A JFace Viewer segítségével modell megjelenítés jobb megértése érdekében a egy TableViewer működését magyarázom el. Ez négy osztály bemutatását jelenti:

- Viewer
- ContentProvider
- LabelProvider
- EditingSupport

5.4.1. Példamodell



5-8 A példamodell

A könnyebb átláthatóság, és egyszerűség kedvéért egy egyszerűsített fa alapú modellen mutatom be a JFace táblázatban történő megjelenítés mechanizmusát, ezt a modellt szemlélteti az 5-8 ábra. Ez egy olyan objektum (TreeRoot), amely három gyermekkel rendelkezik, ezek a TreeNode-ok. Mindegyik gyermek tartalmaz legalább egy levél elemet, egy TreeLeaf-et. Minden fában szereplő elem rendelkezik egy name attribútummal, ami az adott elem nevét tartalmazza, a példában ezt a tulajdonságukat fogom megjeleníteni.

Ez a Model-Viewer-Controller minta Model része, amit a felhasználó számára elérhetővé kell tenni.

5.4.2. A viewer működése

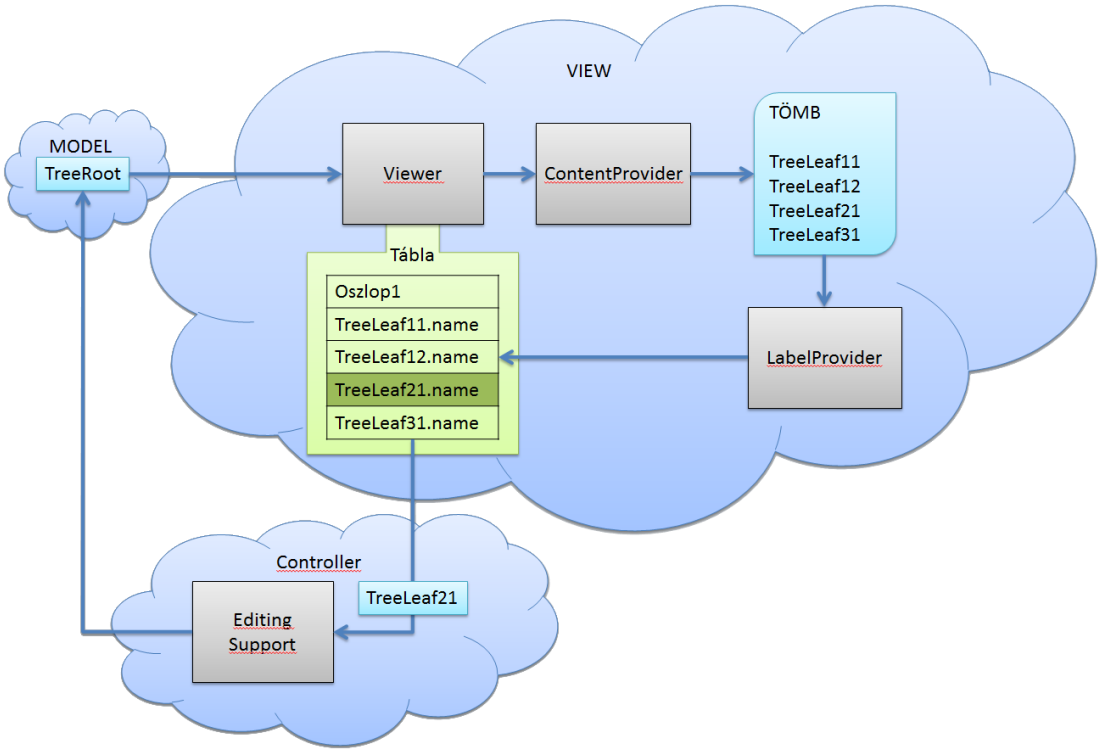
A JFace megjelenítésének alapja maga a Viewer osztály. A korábban említett Model-View-Controller architektúrából rajta keresztül kerül megvalósításra mind a modell megjelenítése és szerkesztése.

A megjelenítésre szánt modellt a Viewer bemeneteként lehet megadni. Ez a bemenet lehet bármilyen Object típusú elem, ami Java-ban azt jelenti hogy bármi, hisz az Object minden osztály ősosztálya, minden az Object-ből származik le.

Ez az osztály tartalmaz egy, a megjelenítésért felelős táblázatot. Azért hogy ebben a modell milyen elemei, és azoknak milyen tulajdonságai jelennek meg, kettő kisegítő osztály a felelős. Az első a ContentProvider, ez a bemenetként megadott objektumon a specifikáció szerint végighalad, és egy tömböt ad vissza. Ennek a tömbnek az elemeit kapja meg a másik kisegítő osztály, a LabelProvider, amely feladata, hogy kapott elemekől a specifikált tulajdonságokat egy darab String objektumként visszaadva a táblázat számára létrehozza a kimenetet. Ez tehát az architektúrának a View részét valósítja meg.

A harmadik rész a Controller, ami a modellen végzett változtatásokat hajtja végre. Ezt az EditingSupport kiegészítő osztály valósítja meg. A táblázat minden egyes sora a ContentProvider által szolgáltatott egy-egy modellelemre mutat. Az EditingSupport elkéri az elemet, majd a specifikáció alapján attól függően, hogy a táblázat hányadik oszlopában történt a kijelölés, megváltoztatja az adott tulajdonságát a bemeneti modellelemnek.

Az 5-9 kép a leírt működést szemlélteti. A korábban vázolt példamodellt egy Viewer bemenetére adva azt a ContentProvider feldolgozza, a kimenete a modelben szereplő TreeLeaf elemekből álló tömb. Ez átadja a LabelProvidernek, ami az elemektől elkéri a name attribútum értékét, ezt adja vissza a viewernek, amit az megjelenít a táblázatban. Felhasználói interakcióval kiválasztva a TreeLeaf21 elem sorát, az objektum az EditingSupport osztályhoz kerül. Itt lehet frissíteni az elem attribútumát, amely változtatás a modellben történik meg.



5-9 A JFace Viewer Model-Viewer-Controller megvalósítása

5.5. Eclipse run configuration

A bővítmény megvalósítása során az Eclipse run configuration megvalósítását is felhasználtam, ezért indokoltnak érzem, hogy a lényegesebb osztályok felelősségéről röviden említést tegyek.

Az Eclipse fejlesztő környezet támogatja más programok futtatását az Eclipse Launching Framework segítségével. Ezt egy LaunchConfigurationDelegate osztály definiálja, ami megszabja, hogy mi induljon el, milyen bemenetekkel, paraméterekkel.

Ahhoz, hogy a futtatás paraméterezhető lehessen egy LaunchConfigurationTab osztály létrehozása szükséges, ami az adott futtatáshoz a felhasználói felületet nyújtja. Egy ilyen általánosan használt, az Eclipse által definiált tab a Common tab, ami a különböző run configuration-ök workspace-be mentését, ezáltal pedig azok megosztását teszi lehetővé. Ennek használatával nem szükséges mindig létrehozni egy új konfigurációt másik workspace használatakor, csak importálni kell a már meglévőt [18]. A feladat során megvalósított run configuration a 8-17 ábrán látható.

6. Constraint Satisfaction Problem – CSP

A feladat egyes részei, mint később bemutatom levetíthetőek a CSP megoldás keresési módszerre, ezért az alapjainak megértése elengedhetetlen.

6.1. CSP alapjai [19]

Egy CSP alapvetően egy olyan probléma, amit változók (V1;V2;..;Vn) és ezekre vonatkoztatott megkötések (constraints – C1;C2;...;Cn) határoznak meg. Minden változóhoz tartozik egy, a lehetséges értékeit tartalmazó domain (D1;D2;...;Dn). Minden megkötés a változók egy részére vonatkozik, és megszabja, hogy azok milyen kombinációban vehetik fel domain értékeiket [22].

A domaineknek két fajtája van, a véges számú értéket tartalmazó (finite domain), valamint a végtelent tartalmazó (infinite domain). A probléma amit a diplomaterv során meg kell oldanom az előbbi esetre vetíthető le, így ezt ismertetem mélyebben.

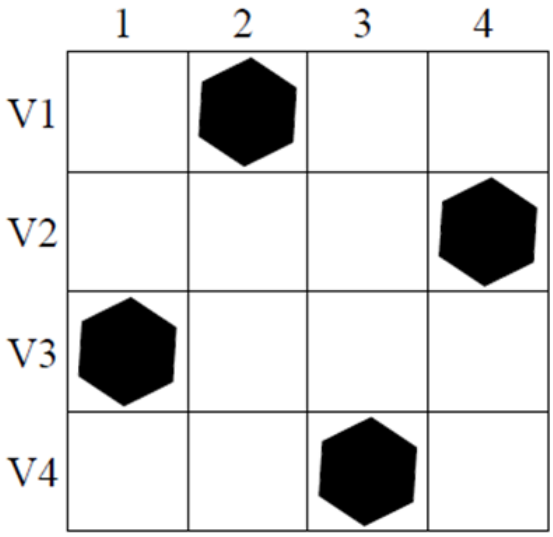
A domain értékek változókhoz rendelése (assignment) határozza meg egy probléma egy állapotát, ez azt jelenti, hogy a változók értékétől függően a probléma megoldott vagy nem. Az olyan hozzárendelést, amelyik egyik megkötést sem sérti következetes hozzárendelésnek (consistent/legal assignment) hívják. Teljes hozzárendelés az, amelyikben az összes változó említésre kerül, egy megoldás pedig az előző kettő együttese, azaz minden változó meg van említve, és nem sérti egyik megkötést sem.

A feladat ilyen problémák esetén, olyan hozzárendelés találása a változók és domainjeik között, ami teljes hozzárendelésnek felel meg. Egyes esetekben, amilyen ez is – mint például a diplomaterv során megoldott feladatnál is –, a cél az összes ilyen hozzárendelés megtalálása.

Számos való életből származó problémát lehet találni, ilyen például az oktatásban a termek összeszervezése a vizsgákra, vagy egy reptéren a kapuk elosztása a járatokhoz. A módszert azonban egyszerűbb bemutatni egy nem igazán valós eseten, ezért az N királynő problémáját reprezentálom.

6.2. N királynő probléma

A megoldandó feladat itt egy NxN-es táblán az N darab királynőt úgy elhelyezni, hogy azok egymást ne tudják leütni, azaz – a sakk szabályai alapján a királynő lépéseinek lehetőségeit figyelembe véve – nem állhatnak egy sorban és oszlopban, valamint nem lehetnek ugyanazon az átlón sem. A 6-1 ábra ennek a problémának az egyik megoldását mutatja 4 királynőre.



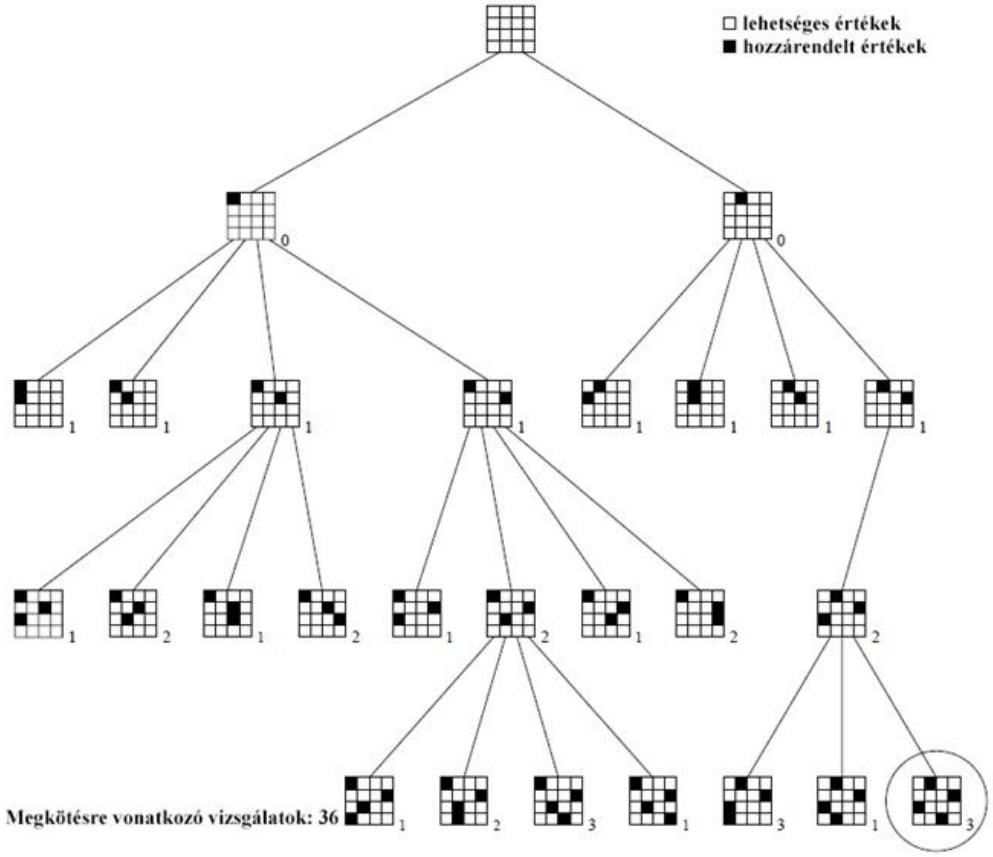
6-1 A négy királynő probléma egy megoldása [20]

A probléma megoldásához a következő lépések elvégzése szükséges:

- változók hozzárendelés a sorokhoz V = {V1,V2,V3,V4}
- minden sorhoz domain rendelése D = {D1,D2,D3,D4}, minden domain ugyanabból a négy értékből áll, D1 = D2 = D3 = D4 = {1,2,3,4}
- megkötéseket meghatározása a változókra:
    - a királynők nem lehetnek ugyanabban a sorban, a megkötés triviálisan teljesül, mert a változóink a különálló sorokat reprezentálják, ezáltal a rajtuk álló királynők biztosan külön sorokba kerülnek.
    - a királynők nem lehetnek ugyanabban az oszlopban, $Xi \neq Xj$, azaz az i sor domainből felvett értéke nem lehet egyenlő a j sor értékével
    - a királynők nem lehetnek ugyanabban az átlóban, $|i-j| \neq |Xi-Xj|$, vagyis sorszámok különbségének abszolút értéke különböző a domain értékek abszolút értékétől

A feladat ezután a változókhoz olyan domain értékek rendelése, amik teljesítik a korábbi megkötéseket. Amennyiben ez sikerült, megoldást találtunk.

Ez esetben az egyszerű keresés az ábrán látható megoldást, a {2,4,1,3} értékhalmazt adja, azaz ezeken a helyeken állhatnak az egyes sorokban a királynők. A keresés menetét a 6-2 ábra mutatja.



6-2 A négy királynő problémájának megoldása CSP segítségével [21]

Ez a megoldás keresés a backtrack (BT) algoritmus, amely a változókhoz egészen addig rendeli hozzá az értéket amíg hibás eredményre nem jut. Hiba esetén, azaz ha a keresés során megszegi az egyik megkötést, „visszalépked" odáig, ahol az utolsó, legális hozzárendelést végezte, majd a következő domain érték vizsgálatával folytatja a keresést.

A 6-2 ábra csupán az első helyes megoldás keresését mutatja be. A keresési algoritmus ezalatt pedig 26 hozzárendelést, valamint – abban az esetben, ha minden hibát 1 vizsgálattal ki képes szűrni – 36 darab, a megkötésekre vonatkozó vizsgálatot végez el.

Azonban, ha a vizsgálatokat az első sor esetén a 2. domain értéktől, azaz 2-től kezdte volna, ezek a számok lecsökkentek volna 9-re és 17-re. Ez a különbség egy ekkora probléma esetén nem számottevő, azonban a lépések száma exponenciálisan nőhet a probléma méretének növelésével, ezáltal pedig az az idő is, míg rátalálunk a megoldásra.

Ilyen esetekben érdemes különböző heurisztikákkal kísérletezni a megoldás keresésénél. A heurisztikák többek között a változókra, valamint azok domain értékeire vonatkozó vizsgálati sorrendet határozzák meg.

A legegyszerűbb ilyen heurisztikák:

- előre vizsgálat (forward checking, FC) – egy változóhoz domain érték rendelése a többi domain-ből „kitörli" a megkötésekkel hozzá kapcsolható értéket, megszüntetve ezzel az n királynő probléma esetén az ugyanabban az oszlopban lévő királynők hibát, valamint az átló hibákat
- minimum megmaradt érték (minimum remaining values, MRV) – a következő hozzárendelést azon a változón végezi, amelyiknek a legkevesebb lehetséges értéke maradt
- fokszám (degree) – ahhoz a változóhoz rendel legközelebb értéket, amelyik a legtöbb megkötésben szerepel
- minimum konfliktus (minimum-conflict, MC) – hibás hozzárendelés felfedezése után olyanra változtatja a változót, ami a legkevesebb konfliktust okozza

A heurisztikák alkalmazásának hatékonyságát, indokoltságát hivatott az 1. Táblázat megmutatni.

Probléma	**BT**	**BT+MRV**	**FC**	**FC + MRV**	**MC**
USA	(>1.000K)	(>1000K)	2K	60	64
N-királynő	(>40.000K)	13.500K	(>40.000K)	817K	4K

1. Táblázat Átlagos vizsgálati lépésszáám különböző heurisztikákra [23]

A táblázatban megjelenített két probléma közül az első az USA 50 államának színezése 4 színnel (ezesetben a feltétel, hogy semelyik két szomszédos állam nem kaphatja ugyanazt a színt). A másik a 2-50 királynő problémák közül az összes megoldásának megtalálása.

A zárójelben lévő értékek azt jelképezik, hogy adott lépésszám alatt nem sikerült megoldást találni a problémára, a K pedig az ezres szorzót jelenti.

Ezek alapján láthatóan számottevő különbséget jelent a különböző heurisztikák alkalmazása egy-egy problémára, azonban nem egyértelmű, hogy melyik segítségével érhető el javulást, és melyikkel nem (lsd. pl. FC oszlop).

6.3. Constraint Logic Programming – CLP

A CLP a logikai programozás kiterjesztése CSP problémák megoldásához. A logikai programozás során alapvetően szabályokból és tényekből álló kikötéseket teszek. Logikai programozási nyelv például a Prolog.

Egy egyszerű, ám remek példa a Prolog nyelv használatára:

- tudjuk, hogy István ember
- tudjuk, hogy az ember gyarló
- tehát István gyarló

Ez az igazán egyszerű logikai kapcsolat, tények és kikötések segítségével logikai programban a következőképpen néz ki:

```
gyarló(X):- ember(X).
ember(István).
```

Ez által felépítettem a logikai programban egy tudásbázist. Ezt felhasználva egy logikai program a gyarló(István) felvetésre igazzal fog visszatérni. A programkódban a gyarló(X):- a szabály fejrésze, az ember(X) a teste, az ember(István) pedig egy tény. A tények olyan szabályok, amik nem rendelkeznek testtel.

A CLP az ilyen nyelvekhez adja hozzá a CSP-t, azaz a szabályok testében lehetőség nyílik megkötéseket adni változókra, és ez alapján keresni megoldásokat.

Egy egyszerű példa CLP-re:

```
A(X,Y):- X+Y>0,B(X),C(Y).
```

Ebben az esetben az X+Y>0 a CSP megkötés, B(X), C(Y) pedig atomi formulák, műveletek (literals), vagy tények. Az egész szabály A(X,Y) akkor lesz igaz, azaz akkor adódik megoldást a problémára X és Y formájában, ha a szabály mindegyik része, azaz a megkötések, és az atomi műveletek is igazzal térnek vissza.

A korábban bemutatott 4 királynő probléma megoldása GNU Prolog segítségével:

```
queens(L):-
	length(N),
	N#>2,
	fd_domain(L,1,N),
	fd_labeling(L),
	safe(L).

safe([]).
safe([X| Y]) :-
	noattack(X, Y),
	safe(Y).

noattack(X, Xs) :-
	noattack(X, Xs, 1).

noattack(_, [], _) :- !.
noattack(X, [Y | Ys], Nb) :-
	X#\=Y,
	X#\=Y+Nb,
	X#\=Y-Nb,
	Nb1 is Nb+1,
	noattack(X, Ys, Nb1).
```

A kódban a korábban leírt megkötések szerepelnek. Ez a megoldás működik nagy számú királynőre is, ám a megoldási idő elég hosszúra nyúlik. 4 királynő esetén a program futtatásának eredménye:

```
queens(4,[Q1,Q2,Q3,Q4]).
Q1 = 2
Q2 = 4
Q3 = 1
Q4 = 3.
```

A program segítségével tehát megtaláltam a probléma korábban bemutatott megoldását.

7. Visszajelzés lehetőségei

A feladat végcélja egy olyan, korábban mutatott hibafa megjelenítése, amely a felhasználók által definiált specifikációk alapján az AUTOSAR szoftver komponensének bemeneti meghibásodásai alapján a működés meghibásodását jellemzi. Ebben a fejezetben a megjelenítés lehetőségeit taglalom.

7.1. Grafikus visszajelzés

Ahhoz, hogy a visszajelzés értékelhető kimenetet jelentsen a Java több lehetőséget is nyújt a felhasználók számára. Az első ilyen lehetőség egy saját megvalósítás készítése, melynek előnye, hogy olyan megoldást lehet implementálni, ami teljes mértékben a feladatnak megfelelő, azonban ipari, és mérnöki szempontból nem a legmegfelelőbb, hiszen rengeteg grafikus megjelenítésre tervezett Java API található meg a világhálón.

Emiatt a feladat megvalósítása során én is amellett döntöttem, hogy egy már meglévő API-t használva készítek hibafa kimenetet. Több különböző megoldást vizsgáltam meg, ezek közül hármat találtam bemutatásra alkalmasnak:

- GraphStream
- GraphViz
- JGraph

A megoldás keresése során három szempontot vettem figyelembe:

- Az API egyszerűen használható legyen (könnyen algoritmizálható gráf készítés)
- Lehessen a gráf pontjait konfigurálni (különböző alakzatok)
- Permanens kimenet készítésére alkalmas legyen (pl. PDF)

Az általam vizsgált lehetőségek közül mindegyik ingyenes licenc alatt használható.

7.1.1. GraphStream [24]

A Graphstream egy gráfok készítését, és kezelését lehetővé tévő Java könyvtár csomag, ami a gráfok dinamikus módosítására összpontosít. A fókusza a dinamikus kapcsolatok kialakítása és karbantartása különböző méretű gráfokban.

A könyvtárak célja, hogy egy könnyen használható eszközt nyújtsanak gráfok megjelenítéséhez, és a velük való munkához. Többféle gráf osztály készíthető vele, a gráf csúcsaihoz hozzárendelhetők különféle információk, számok, sztringek, vagy objektumok.

A leírás alapján tehát egy teljesen megfelelő API a hibafa megjelenítéséhez. A gráfok létrehozása egyszerűen végrehajtható vele, azonban a gráf csúcsai nem módosíthatóak, valamint a permanens kimenet készítésére sem a legalkalmasabb program, ezt természetesen jól magyarázza az, hogy a dinamikus gráf kezelés a fókusza.

A kimenet készítéséhez nem ezt a megoldást választottam az említett okok miatt, de egy igen hasznos eszközt ismertem meg, amivel percek alatt látványos, és hasznos gráf analízisek végezhetőek el.

7.1.2. GraphViz [25]

Az előző API-hoz hasonlóan ez a grafikus kijelzést megvalósító eszköz is egy gráfok megjelenítését lehetővé tevő applikációt nyújt a fejlesztők számára. Amit lényeges tudni erről a megoldásról, hogy a három kritériumból kettőt maradéktalanul teljesít.

A gráfok pontjai, ahogy az a GraphViz honlapján található galériában is látszik, nagy szabadsággal módosíthatók, azaz el lehet érni bármilyen formát. A permanens eredmény kijelzése szintén egyszerűen megvalósítható vele, a program eredményeként rengeteg típusú fájl generálható, többek között BMP, JPEG fájlok is.

Ami miatt nem ezt a megoldást választottam, a gráfok létrehozásának komplikáltsága. A program egy szöveges fájlt vár bemenetként, amely egy DOT [26] nyelvű gráf leírást tartalmaz magában, majd ezt értelmezve készíti el a kimenetet. Ez az általam választott megoldáshoz képest kissé nehézkesebb lett volna, emellett egy újabb fájl generálását kellett volna megoldani a feladatban, amit nem érzek kellően sokrétű feladatnak.

7.1.3. JGraph [27]

A feladathoz általam választott API a JGraph, egy a GitHub oldalon ingyenesen elérhető grafikon készítő megoldás, mely egy Java, C# és Javascript nyelvekhez alkalmazható diagram készítésre írt applikációs könyvtár.

Segítségével a gráfok elkészítése igen egyszerű feladat, létre kell hozni egy mxGraph osztályt, majd erre insertVertex() és insertEdge() függvényhívásokkal a gráf csúcsait és vonalait „megrajzolni”.

Mind permanens, mind pedig szerkeszthető kimenet készítését lehetővé teszi a program. Az API tartalmaz egy PDFWriter nevű osztályt, ami az elkészített gráfot egy pdf fájlba exportálja.

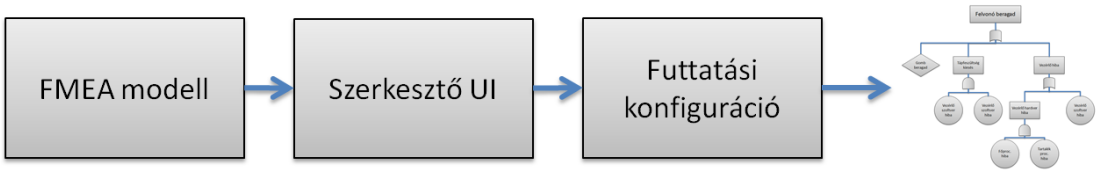
A gráf csúcsainak szerkesztéséhez egy olyan lehetőséget biztosít, amely szerint a felhasználó által kívánt alakzatokat egy XML formátumú fájlban kell meghatározni. Ebben a fájlban az alakzatokat a Hypertext Markup Language 5 (HTML 5) canvas [28] rajzolásához hasonlóan kell megadni.

Az XML fájlban valósítottam meg a hibafára jellemző AND és OR kapukat, amiket ezután az insertVertex() csúcs beszúrásnál megnevezve az adott alakzat lett a gráfhoz hozzáadva.

Mind az előbbi GraphViz, mind pedig a JGraph alkalmas a gráfhoz adott elemek automatikus elrendezésére. A megoldásomban a gráf elrendezésének hierarchikussá tételével értem el a végső kimenet felépítését. A módszer az elemek összeköttetései mentén egy szintekre bontott gráf struktúrába rendezi a már létrehozott csúcsokat.

8. Megvalósítás

Ebben a fejezetben a diplomaterv céljaként kitűzött analízist megvalósító bővítmény fejlesztésének lépéseit tárgyalom. Az előző fejezetekben mindent, ami a téma megvalósításának megértéséhez szükséges kielemeztem, értelmeztem.

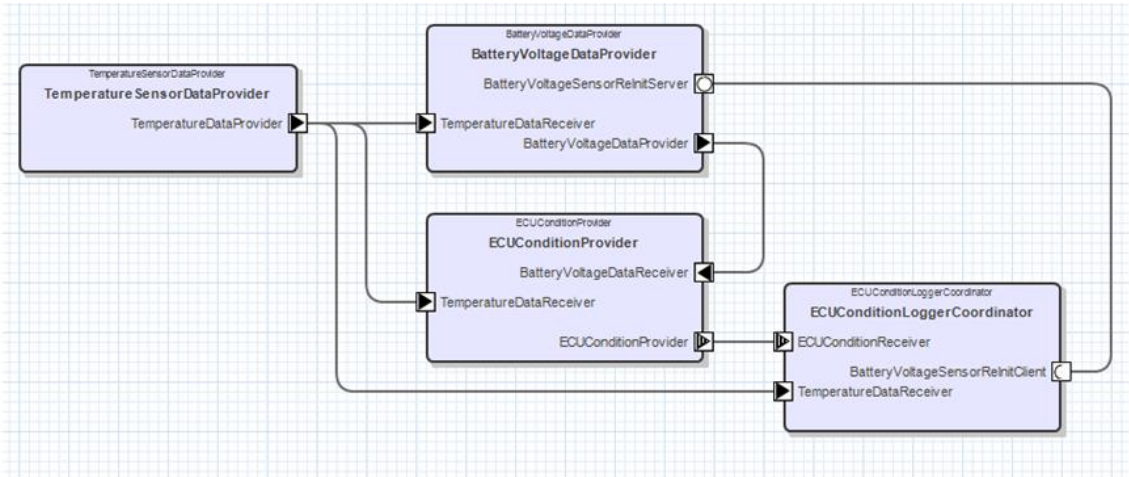


8-1 A három plug-in-ből álló szoftver blokkvázlata

A feladat megoldása egy három Eclipse plug-in alkalmazásból álló szoftver csomag lett, ezek kapcsolatát mutatja a 8-1 ábra.

Az első blokkban szereplő FMEA modell a feladathoz EMF segítségével készített metamodell bővítményét reprezentálja. Ez a modell karbantartásáért felelős. A második a szerkesztő felület, ami az első blokk elemeit jeleníti meg és teszi lehetővé szerkesztésüket. A harmadik az analízis futtatását valósítja meg és kimenetet készít az eredmény alapján.

8.1. Modell fejlesztéshez



8-2 A fejlesztéshez használt AUTOSAR Architect-ben megvalósított modell

A feladat megoldásához először egy, az AUTOSAR Architect-ben készített modellt ismertem meg, amely segítségével a szerkesztő különböző funkcióit megvalósítottam. A 8-2 ábra ezt a modellt szemlélteti. Ez egy valós ECU leegyszerűsített egyik funkciójának működési modelljét mutatja be. A komponensek feladata egy akkumulátor kondícióinak felmérése, naplózása, valamint beavatkozás ha szükséges.

A képen egy kompozíció látható, ami négy szoftver komponenst tartalmaz:

- TemperatureSensorDataProvider
- BatteryVoltageDataProvider
- EcuConditionProvider
- EcuConditionLoggerCoordinator

Ezek működését, és felépítését mutatom be röviden a feladat szempontjából releváns információkat összeszedve.

8.1.1. TemperatureSensorDataProvider

A mért hőmérsékletet – egy egyszerű értéket tartalmazó implementációs adat típusban – szolgáltatja a többi AUTOSAR komponens felé. A komponens egy kimeneti porttal rendelkezik, amelyiken a hőmérséklet értékét küldi. Működés szempontjából egy futtatható entitása van, ami bizonyos időközönként, azaz időzítési eseményre triggerelve lekérdezi a szenzortól a mért hőmérsékletet, és azt a kimenetén elérhetővé teszi.

8.1.2. BatteryVoltageDataProvider

A BatteryVoltageDataProvider méri a jármű akkumulátorának feszültségét (VoltageSensorData) és teszi azt elérhetővé az ECUConditionProvider számára a komponens nevével egyező porton keresztül. A TemperatureDataReceiver port a mért hőmérsékletet kapja, a BatteryVoltageSensorReInitServer porton keresztül az elem újraindítása válik elérhetővé más komponensek számára. Nem megfelelő hőmérséklet esetén hibát jelez, újra kell indítani, amint ismét megfelelő a hőmérséklet.

A működésének megfelelően két futtatható entitással rendelkezik, az első a feszültség lekérését végzi, ennek megfelelően egy időzítési esemény triggereli. A másik az újraindításért felelős, és egy külső függvényhívás alapján indul el.

8.1.3. EcuConditionProvider

A mért feszültséget az ECUConditionerProvider kapja. Ez a komponens akadályozza meg az ECU túl magas feszültség vagy hőmérséklet miatti sérülését. A működéshez egy futtatható entitással rendelkezik, amit egy eddig nem említett esemény típus két példánya indíthat el, ami abban az esetben jelez, ha adat érkezik. A kimeneti portján azt jelzi, hogy a kapott adatok alapján az ECU megfelelő kondíciókkal rendelkezik a működéshez.

8.1.4. EcuConditionLoggerCoordinator

A negyedik komponens az ECUConditionLoggerCoordinator, ami az előző kondíció adatát dolgozza fel, naplózza az ECU által tapasztalt környezeti hatásokat, és korrekciós műveleteket hajt végre, a feszültségérzékelő szenzor újraindítását végzi, ha az szükséges, valamint megfelelőek hozzá a körülmények.

A modellben ez az egyetlen komponens, ami vezérlési folyam típusú porttal rendelkezik, ez valósítja meg a feszültség mérő komponens újraindítását, amennyiben az EcuCondtitionProvider-től kapott adat alapján a működéshez ez szükséges.

8.2. EMF modell

A feladat következő lépése egy olyan modell meghatározása, aminek a segítségével megvalósíthatóvá válik a hiba analízis módszer. A bemutatott projekt, valamint a szoftver komponensek felépítése alapján meghatározható, hogy ehhez milyen modell komponensekre van szükség.

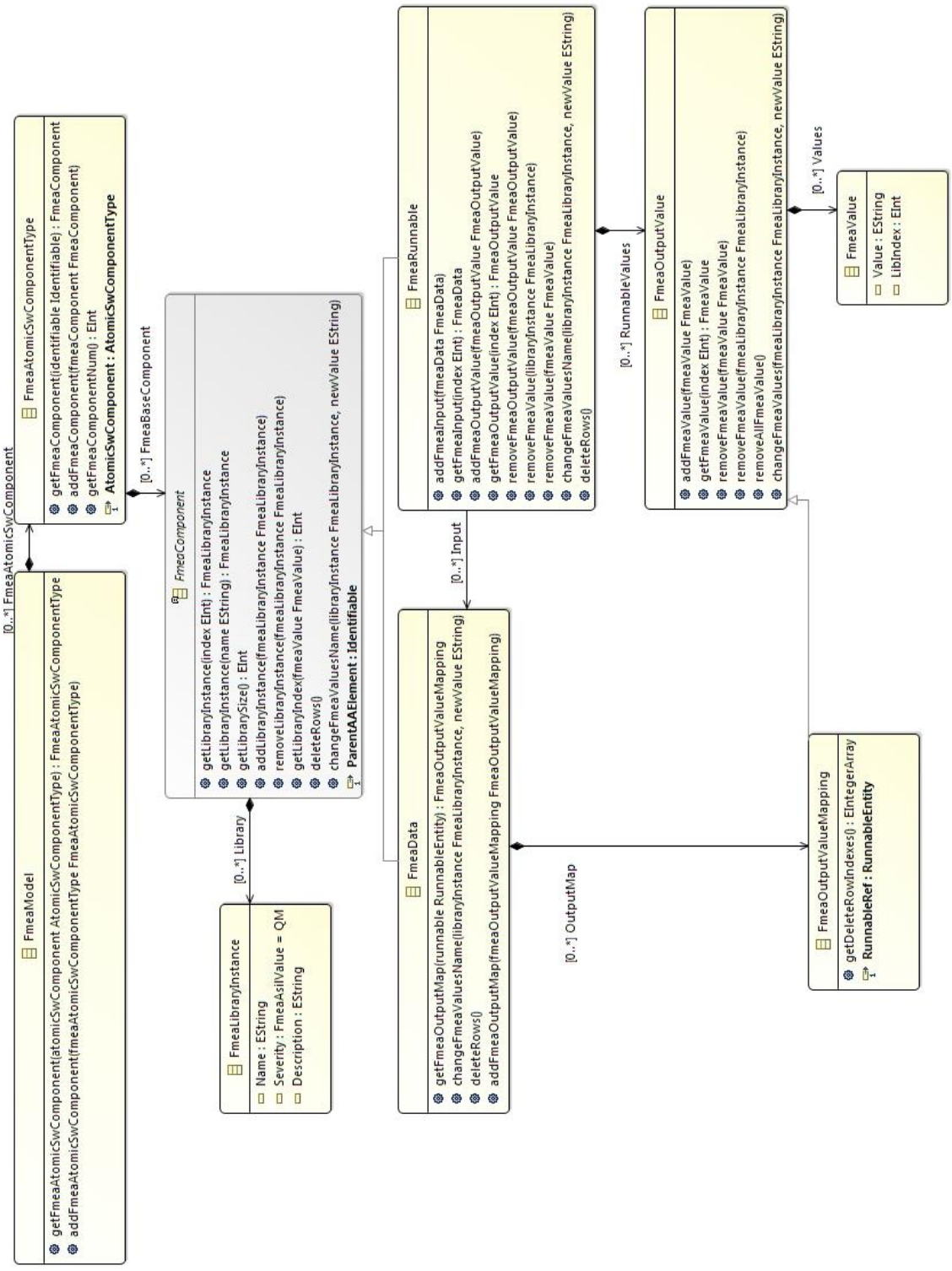
Első lépésben megállapítottam, hogy a szoftver komponensek melyik részeinek meghibásodása érdekes a vizsgálat szempontjából. Ezek a komponensek működését jellemző futtatható entitások, azaz egy elem nem várt meghibásodása egy, vagy több ilyen entitásának hibás működését jelenti.

A második lépés a meghibásodást okozó elemek meghatározása, ez egyértelműen a futtatható entitások bemenetére csatlakoztatott adat típusokat jelenti.

Miután a vizsgálandó elemeket meghatároztam, az EMF metamodellt kellett felépíteni. Ahhoz, hogy az analízis végrehajtható legyen szükséges minden elemhez egy-egy a lehetséges hibákat tartalmazó könyvtár hozzárendelése, valamint egy olyan megoldás, amely a különböző komponensek hibáinak kombinációit összevonja. Ez utóbbi egy igazságtábla a szerkesztőben, ezt a későbbiekben bemutatom.

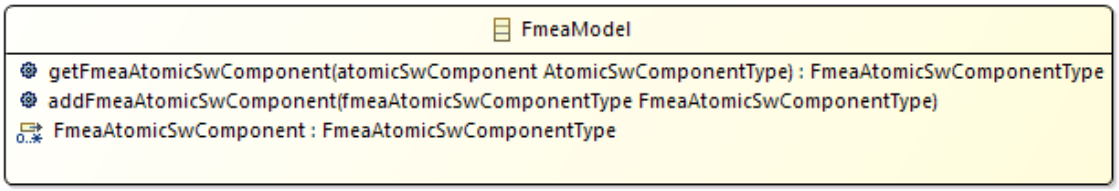
A 8-3 kép a létrehozott metamodellt mutatja (az ecore editor szerinti megjelenítése az 5-6 ábrán látható). Láthatóan többé kevésbé leköveti az AUTOSAR szabvány szerint definiált AtomicSwComponentType felépítését, abból csak az analízishez lényeges komponens elemeket megtartva.

A teljesség igénye nélkül a fontosabb elemeket részletesebben is bemutatom.



8-3 A feladat során megvalósított metamodell UML diagramja

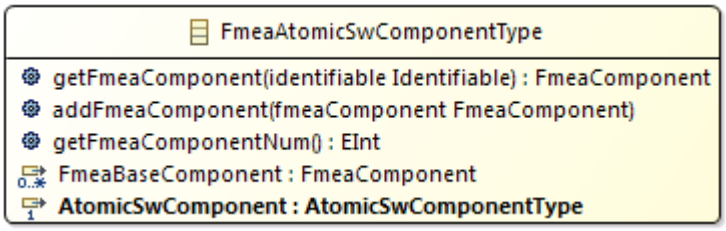
8.2.1. FmeaModel



8-4 A metamodell legfelső szintje

A legfelső szintet egy FmeaModel osztály jelenti, ez alatt van összegyűjtve az összes modellhez tartozó szoftver komponens. A két létrehozott függvény új komponens hozzáadását, valamint egy már létező lekérését valósítja meg.

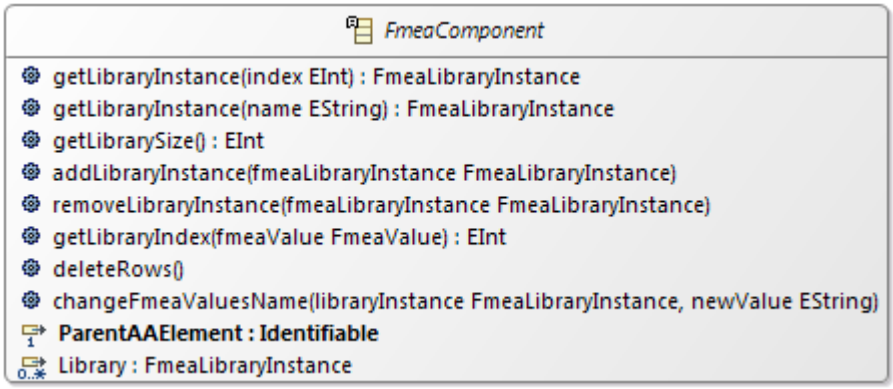
8.2.2. FmeaAtomicSwComponentType



8-5 Az AtomicSwComponentType komponenseket jellemző FMEA elem

A szoftverkomponensek tartalmazzák az összes futtatható entitásukat, valamint az összes bemenetüket elkülönítve. Erre azért van szükség, mert egy bemenet akár több futtatható entitás bemeneti értékét is jelentheti, viszont a hiba típusuk minden esetben ugyanaz.

8.2.3. FmeaComponent

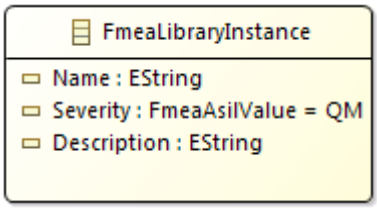


8-6 Az analízis magját adó komponensek ősosztálya

Az FmeaComponent nevű absztakt osztály az előbb említett adat és futtatható entitások tárolására szánt osztályok őseként szolgál, valamint a hozzájuk tartozó hiba könyvtárak tárolását, módosítását valósítja meg.

A belőle leszármazó osztályok a hibakönyvtár értékeiből megvalósított igazságtábla adott komponenshez tartozó leképzését tartalmazzák az FmeaOutputValue értékeikben. Ebből az osztályból egy-egy FmeaData esetén több is lehet, hiszen egy adat több futtatható entitás bemenete is lehet.

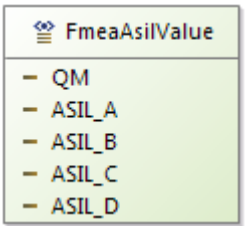
8.2.4. FmeaLibraryInstance



8-7 A hibamódokat ellemző könyvár értékek megvalósítása

Ez jellemzi a modellben a hibamódokat. Három attribútummal rendelkezik, egy névvel, egy súlyossági értékkel, valamint egy leírással, hogy miért is van szükség a hibamódra, hogyan jöhet létre.

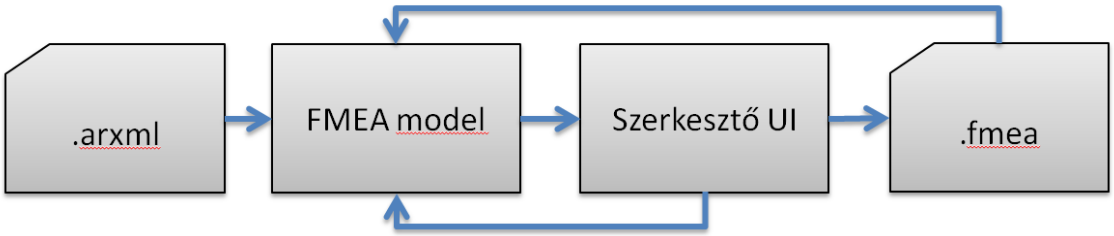
8.2.5. FmeaAsilValue



8-8 A hibamódok súlyosságát jellemző értékek Enum típusa

Ezek mint azt korábban leírtam a hibák súlyosságát jellemzik az autóiparban. A megvalósításban valós értéküket nem vettem figyelembe, azt az analízist végző mérnökök értelmezik, és használják e szerint.

8.3. Szerkesztő felület



8-9 A szerkesztő felület működése

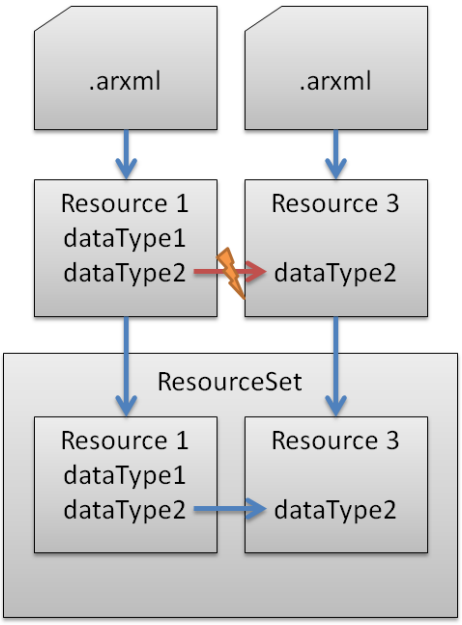
A metamodell megalkotása után a szerkesztését, létrehozását biztosító felület elékszítése volt a cél. Amint azt korábban említettem, az analízis cél komponensei az AtomicSwComponentType elemek voltak, ezért a szerkesztő használatának lehetősége csak az ilyen komponensekre kell adott legyen.

A szerkesztő megnyitásakor először az adott AtomicSwComponentType-hoz tartozó EMF modellt tölti be, ami egy Extensible Markup Language – XML fájlban van lementve, vagy amennyiben még nincs ilyen modell létrehozva megalkotja azt. A modellt egy EMF Resource-ban tárolja, a modellen végzett változtatások ezen hajtódnak végre.

Egy XML fájl tartalmát egy ilyen Resource-ba tölti be a program. Amennyiben egy XML fájl tartalmaz minden elemet az adott komponensről, a Resource-on belül hivatkozni képesek egymásra. Abban az esetben, ha az XML fájlon belül hivatkozott elem egy másik XML-ben, és ezáltal Resource-ban van definiálva, nem találja meg automatikusan. Ehhez szükséges, hogy a két Resource-ot egy ResourceSet-be töltsük be, így feloldhatóvá téve a hivatkozásokat. Ez az eset a feladatom során tipikusan akkor eshet meg, ha egy adattípus, amelyet az egyik szoftverkomponens hivatkozik, egy másik komponens típusai közt van definiálva.

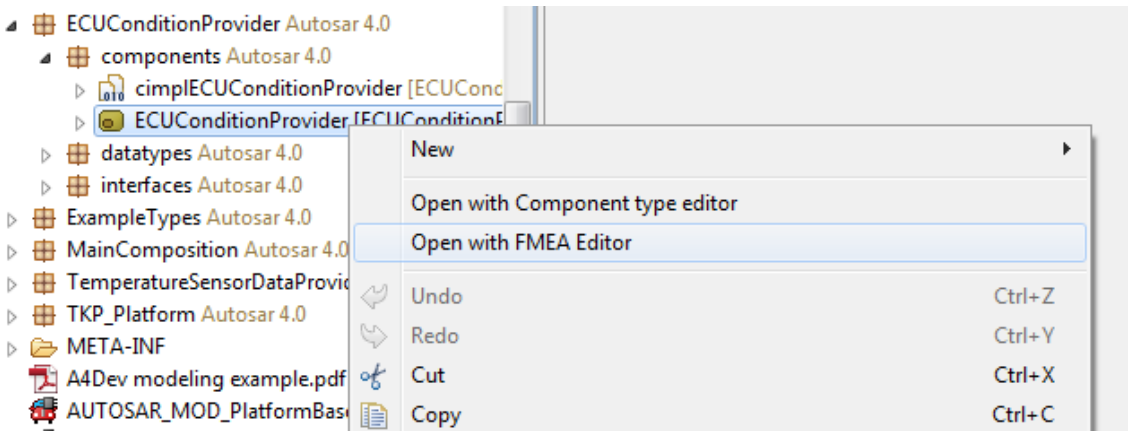
Az XML fájlok betöltését, és az egymás közötti hivatkozást mutatja a 8-10 ábra. A példában kettő XML fájl van, az első definiálja a dataType1, valamint referálja a másodikban definiált dataType2 típust. Amennyiben a Resource-ok külön-külön betöltésre kerülnek, a hivatkozás feloldása problémákba ütközik, egy ResourceSet-be töltve viszont a hivatkozás megfelelően működik.

A betöltés módjának megértése azért lényeges, mert a – később bemutatott – analízis futtatást a program az AUTOSAR Architect arxml fájljai, valamint a bővítményem által létrehozott fmea fájlok alapján végzi. Ehhez az elemeket a fent említett módon adja a modelljéhez.



8-10 Resource-ok betöltése XML-ből

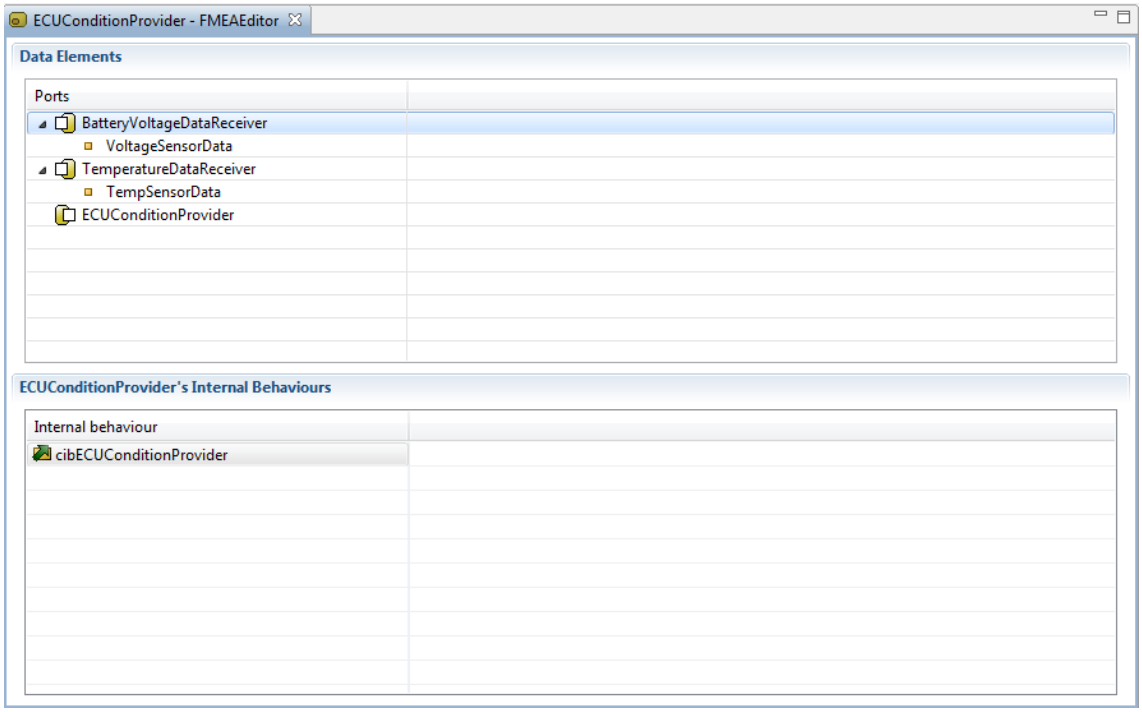
A program a fájlok betöltése után a megnyitott AtomicSwComponentType elemeknek megfelelő FmeaAtomicSwComponent elemet kísérli megkeresni az fmea fájlból betöltött Resource-ban, majd ha megtalálta, összehasonlítja vele a tartalmazott elemeit. Amennyiben valamilyen különbséget talál – új elem, már nem létező elem, vagy akár az egész szoftver komponens hiányzik – létrehozza, vagy törli azt a modellből.



8-11 Az FMEA szerkesztő elérése kontextus menü segítségével

A felület az Eclipse RCP kontextus menüje segítségével nyitható meg, ezt mutatja a 8-11 ábra. Annak a biztosítását, hogy csak az adott típusú elemekre lehessen megnyitni az AUTOSAR Architect által definiált Extension point beállításainál lehetőségem volt kiválasztani, így ezzel a problémával a továbbiakban nem kellett foglalkoznom.

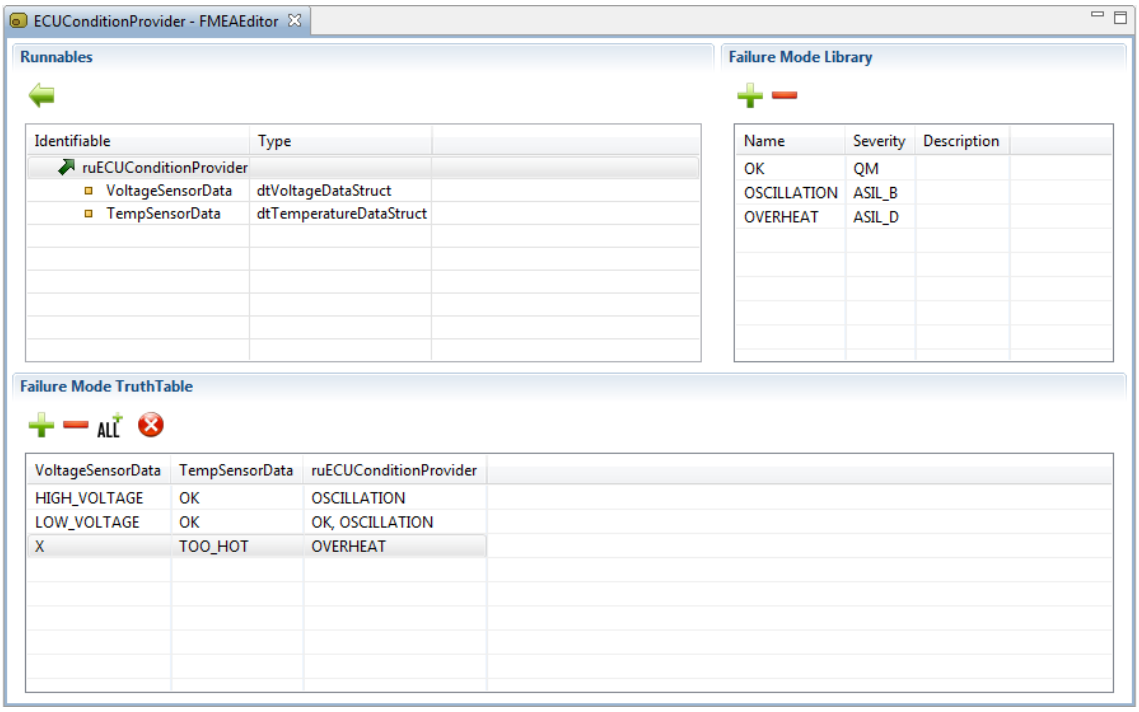
A nézetekben a tartalom szolgáltatását, és szerkesztését mindenhol az adott nézet ContentProvider, LabelProvider és EditingSupport implementációja hajtja végre.



8-12 A szerkesztő első oldala

A szerkesztő kiválasztása után az első oldala látható. Itt az AUTOSAR Architect komponensek már létező tulajdonságai vannak felsorolva. A felső nézetben láthatóak a komponens portjai, valamint rajtuk az interfészüknek megfelelő továbbított adatelemek. Ennek a nézetnek nincs különösebb funkciója jelenleg, csak összegyűjti a portokat, és adat típusokat, hogy a felhasználó számára információt szolgáltasson.

Az alsó részben a komponens belső viselkedései vannak összeszedve. Az egyik ilyenre duplán kattintva jelenik meg a szerkesztő második oldala, ahol a lényeges szerkesztés valósítható meg.



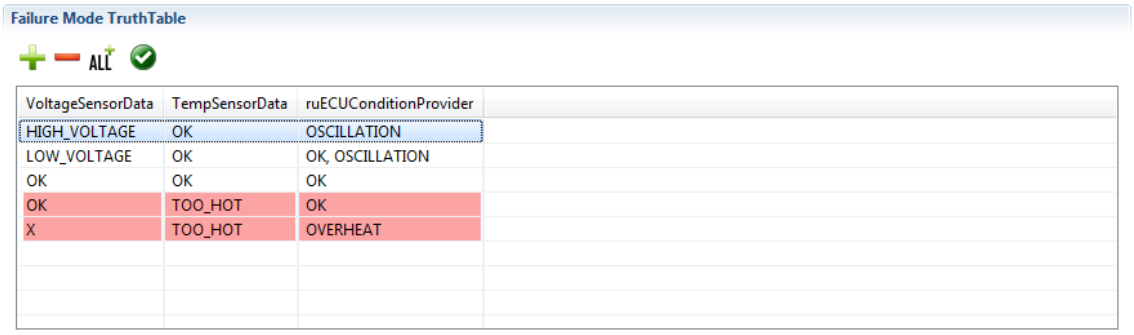
8-13 A szerkesztő második oldala

A második oldal három nézetet tartalmaz. A legelső az előző oldalon kiválasztott belső viselkedés által tartalmazott futtatható entitásokat, valamint azok bemeneteit jeleníti meg. Ezek közül egyet kiválasztva frissül a másik két nézet bemenete.

A felső jobb oldali nézet a kiválasztott elem hibakönyvtárát mutatja meg, az alsó pedig a kiválasztott elemhez tartozó aktuális futtatható entitásra jellemző igazságtáblát. A hibakönyvtár megvalósítása igen egyszerű, az első nézet bemenetére nem a RunnableEntity-k vannak megadva, hanem a korábban ismertetett EMF metamodell FmeaRunnableEntity, valamint FmeaData típusú elemei. Ezek közül a kiválasztottat nyújtja bemenetként a könyvtár szerkesztő nézetnek a program, amiből a ContentProvider lekérdezi az elemeket, a Label provider pedig a megfelelő értékekeket a megfelelő oszlopban megjeleníti.

A szerkesztéshez három féle cella szerkesztőt alkalmazok ebben a nézetben, mivel különböző típusú, és hosszúságú elemeket tartalmaz egy-egy hibakönyvtár elem. A nevet egy egyszerű szöveges cella szerkesztő segítségével lehet váltani. A súlyosság érték, mint korábban említettem egy Enum típusú érték, emiatt itt egy legördülő lista elemeiből lehet választani, a leírás pedig – mivel ez jellemzően egész mondatokat kell tartalmazzon – egy külön szöveg szerkesztő ablakot jelenít meg.

Az alsó igazságtábla rész bemenetének mindig egy FmeaRunnableEntity elemet nyújt a program. Ennek a bemeneteit sorolja fel a szerkesztő nézet, és az értékeket egy legördülő lista elemei közül lehet kiválasztani. Ez a lista a hibakönyvtár elemeit, valamint egy don’t care értéket tartalmaz, aminek a jelentése, hogy az adott kombináció során mindegy milyen értéket vesz fel az adott bemenet. Ezzel a megoldással csökkentettem a létrehozni szükséges igazságtábla sorokat, amennyiben minden bemenetet létre kíván hozni a felhasználó, de vannak elemek amik értéke nem érdekes egy-egy kombináció esetén.



8-14 Az igazságtábla funkcióinak bemutatása

Fontosnak tartom még elmondani az igazságtábla néhány kényelmi funkcióját. Az első a visszajelző ikon, ami abban az esetben, ha minden kombinációt sikerült legenerálni, zöld pipával jelzi a felhasználó felé. A második a generálást segítő, minden kombinációt előállító gomb, ennek segítségével csak a kimenetek értékeit kell szerkeszteni a felhasználónak. A harmadik pedig egy olyan funkció, amely ha egy kombináció kétszer, vagy annál többször szerepel a táblázatban, a sorok beszínezésével jelzi a duplikációt.

8.4. CSP

A feladat azon része, hogy a rendszer modelljét, valamint az igazságtáblák értékeit feldolgozzuk, levetíthető egy CSP-re. A cél, hogy a definiált igazságtáblák alapján meghatározható legyen a vizsgált komponens bemeneteinek meghibásodásainak hatásai.

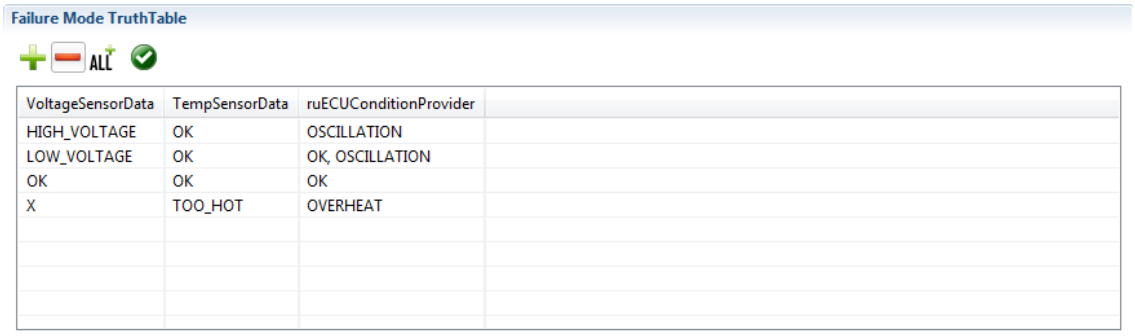
A megoldás kereséséhez változókat rendeltem a be- és kimenetekhez, helyesebben szólva a modellben használt FmeaData és FmeaRunnableEntity elemekhez. Az összes FmeaAtomicSwComponent objektumban szereplő ilyen tehát a CSP egy-egy változója.

A domain-jeik a hibakönyvtár elemei 0-(N-1) – ig kifejezve számmal, ezek értékeit veheti fel minden egyes elem. Ebből egyértelműen látszik, hogy a probléma, amire a megoldásokat keresem, egy véges domain probléma.

A CSP megkötései pedig elsősorban, mint már említettem az igazságtáblák bejegyzései, és másodsorban egy kiegészítő logika, ami segítségével a hibaszámra, valamint a hibák súlyosságára lehet szűrni.

A kiegészítő logika segítségével megoldható, hogy a felhasználók szűrni tudjanak a hibákra. Ennek segítségével a kimenetet áttekinthetőbbé tudják tenni, így használhatóbb, és könnyebben értelmezhető eredményt kapnak.

A probléma megoldásához GNU Prolog-ot használok, elsősorban a beépített véges domain megoldó képessége miatt, emellett mert publikus licenc alatt elérhető szoftver [29]. A program számára biztosítani kell egy bemeneti fájlt, amely tartalmazza az összes direktívát, ami alapján az elemzés eredményre vezethet. Ezt a fájlt a következő alfejezetben bemutatott run configuration egyik osztálya készíti el.



8-15 Példa igazságtábla

A 8-15 ábra igazságtábláját a program a run0 GNU Prolog kódjává alakítja. A kódban láthatóak a domain hozzárendelő (fd_domain), valamint az fd_labeling direktívák.

```
run0([I0, I1],[O0]):-
      fd_domain(I0,0,1),fd_labeling(I0),
      fd_domain(I1,0,2),fd_labeling(I1),
      fd_domain(O0,0,2),fd_labeling(O0),
((I0#=#0)#/\(I1#=#0)#/\((O0#=#1)))#\/
((I0#=#0)#/\(I1#=#1)#/\((O0#=#0)#\/(O0#=#1)))#\/
((I0#=#1)#/\((O0#=#2)))#\/
((I0#=#0)#/\(I1#=#2)#/\((O0#=#0))).
```

Az fd_labeling direktíva alapvetően ahhoz szükséges, hogy a kimenetben ne intervallumokat adjon a program válaszként, hanem minden változót megemlítve adjon megoldásokat, azaz solution-öket kapjunk. Ezzel a direktívával lehet még hozzárendelni a CLP programunkhoz különböző heurisztikákat, ezzel gyorsítva a megoldás keresést. Ezt a feladat során nem használtam, mert nem eredményezett volna érezhető különbséget a modellen, amin teszteltem a plug-int.

```
%A hibaszamszureshez hasznalt kodresz

hibaFilter(Is,MinHibaNum,MaxHibaNum,MinHibaSeverity):-
MinHibaNum #=< MaxHibaNum,
length(Is,N),
summazo(Is,N,MinHibaSeverity,O),
O #>= MinHibaNum,
O #=< MaxHibaNum.

summazo(_,0,_,0).

summazo(Is,N,MinHibaSeverity,O):-
N #>= 0,
N1 #=# N-1,
summazo(Is,N1,MinHibaSeverity,O1),
nth1(N,Is,Elem),
(Elem #>= MinHibaSeverity -> Faulty #= 1; Faulty #= 0),
O #= O1 + Faulty.
```

Ez a kódrészlet a kiegészítő logikát tartalmazza, amit a hibaszűréshez generál a bővítmény. A szűrés minimum és maximum hibaszám értékeket kap. Amennyiben a számolt érték ezek között van, a direktíva igazzal tér vissza. A hibaszámot a kapott MinHibaSeverity alapján határozza meg, összehasonlítja a kapott Is tömb elemeit ezzel a számmal, és ha nagyobb, növeli az értéket. Amennyiben a direktíva egyik része hamissal tér vissza, például a MinHibaNum > MaxHibaNum, a direktíva maga is hamissal fog visszatérni.

```
system0([I0,I1],[O0],[InputHibaMin,InputHibaMax,OutputHibaMin,OutputHibaMax,M
inHibaSeverity]):-
HI0List=[0, 0],nth0(I0,HI0List,HI0),
HI1List=[4, 1, 0],nth0(I1,HI1List,HI1),
hibaFilter([HI0, HI1],InputHibaMin,InputHibaMax,MinHibaSeverity),
HO0List=[0, 2, 4],nth0(O0,HO0List,HO0),
hibaFilter([HO0],OutputHibaMin,OutputHibaMax,MinHibaSeverity),
comp0([I0, I1, O0]).
```

Ezek az egész rendszer összeköttetésére vonatkozó megkötéseket. A fejrészben meg van adva az összes bemenet, kimenet, és a hibaszámok, valamint a minimális hibasúly értéke.

Ezután a be- és a kimenetek hibakönyvtárai alapján létrehozza a HIiList-eket, amik a hibamódok súlyosságértékeit tartalmazzák. Ezeknek az értékeit adja át ezután a hibaFilter direktívának, ami megadja, hogy megfelelő mennyiségű hibaszámot tartalmaz-e az adott bemeneti kombináció által meghatározott kimenet.

Miután ezt igazolta, csak ekkor halad tovább a program az igazságtáblák ellenőrzésére. Az igazságtáblák a comp0-3 rendszerkomponensek szabályai közt vannak. Ezesetben ezek a rendszerkomponensek a korábban bemutatott AUTOSAR kompozíciós diagramján látható AtomicSwComponentType modellelemek.

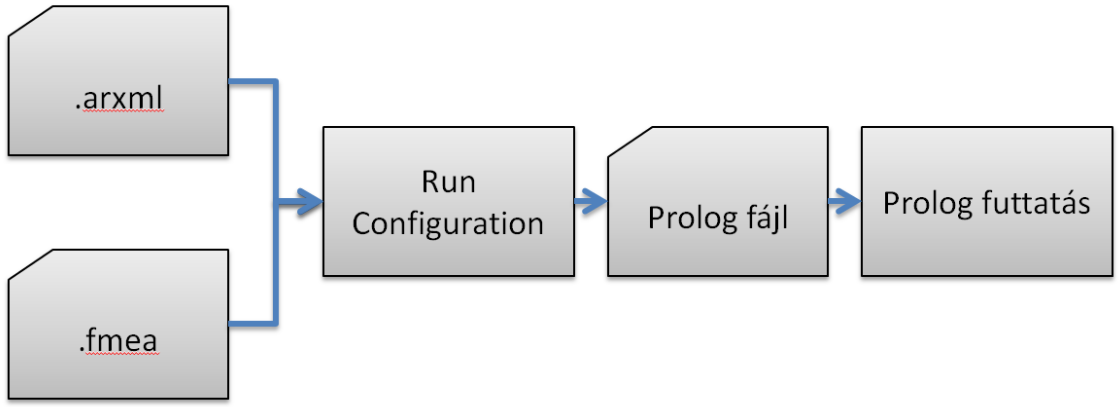
```
main ([InputHibaMin,InputHibaMax,OutputHibaMin,OutputHibaMax,MinHibaSeverity]):-
findall([[I0, I1],[O0]],system0([I0, I1], [O0],
[InputHibaMin,InputHibaMax,OutputHibaMin,OutputHibaMax,MinHibaSeverity]),X),
write(X).
```

A program futtatásához létrehoztam még egy direktívát, ami a generált fájl szabályai alapján megtalálja az összes megoldást. Ezeket a beépített findall direktívával lehet megkeresni, ennek használata során az inputok és outputok olyan módon kerülnek a kimenetre értékhalmaz formájában, ahogy itt megadtuk őket.

A program teljesen alapértékekkel – main([0,2,0,1,0]) – való futtatásának hatására a lejjebb látható értékhalmazt kapom kimenetként. A teljesen alapértékek ebben az esetben azt jelentik, hogy akármennyi hibát elfogadhatónak tartok, valamint bármilyen súlyosságúakat, tulajdonképpen az összes igazságtáblában megadott információra kíváncsi vagyok. Láthatóan az eredmény [[bemeneti értékek],[kimeneti értékek]] formájában jelenik meg, ezzel a beolvasást és értlmezést egyszerűvé téve.

```
[[[0,0],[1]], [[0,1],[0]], [[0,1],[1]], [[0,2],[0]], [[1,0],[2]],
[[1,1],[2]], [[1,2],[2]]]
```

8.5. Run configuration



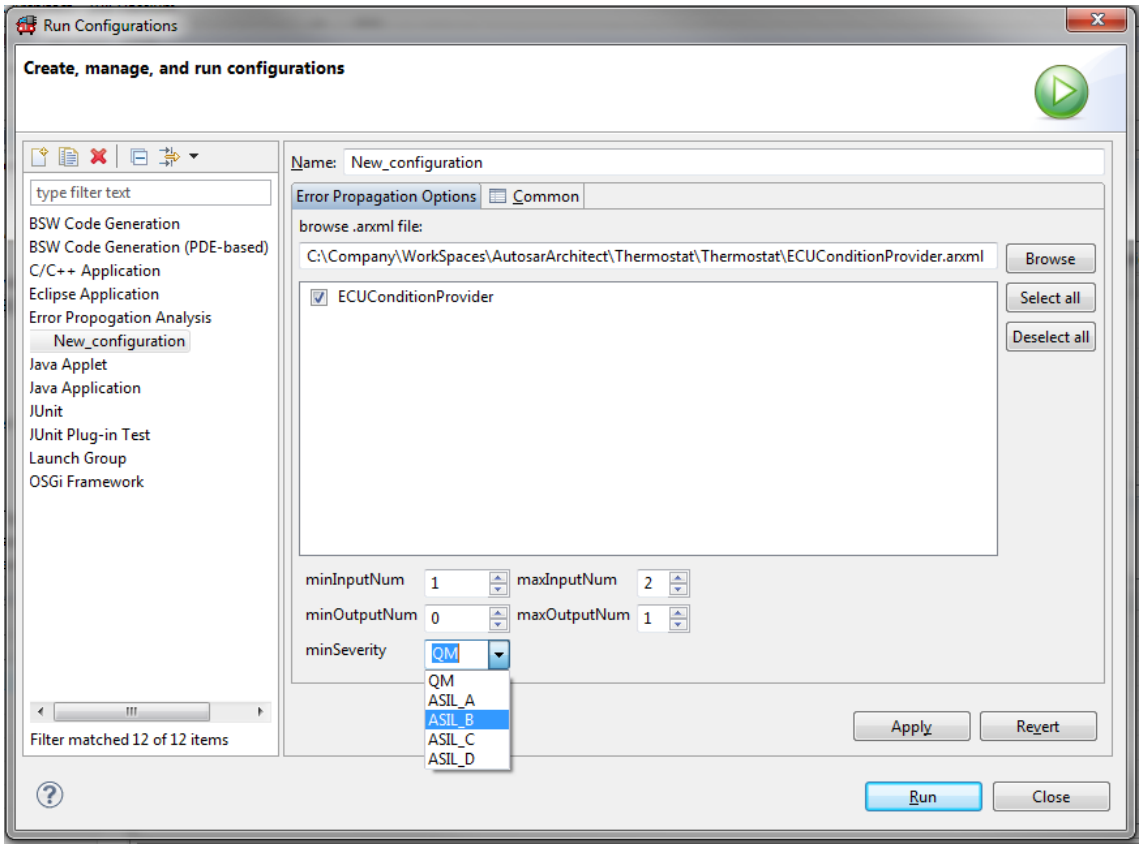
8-16A futtatási konfiguráció blokkvázlata

Az igazságtáblák alapján GNU Prolog bemeneti fájlt, majd annak futtatását egy run configuration-ben valósítottam meg. A 8-17 kép ennek a felületét mutatja. Az analízis elvégzéséhez egyszerűen ki kell választani azt az XML fájlt, amelyikben a keresett AtomicSwComponentType található. A szoftver ezután ellenőrzi, hogy létezik-e hozzá tartozó FMEA analízishez szükséges modell fájl.

Amennyiben létezik a fájl, az oldal felkínálja az XML fájlban található szoftverkomponenseket, és a felhasználó kiválaszthatja melyik(ek) analízisét szeretné végrehajtani.

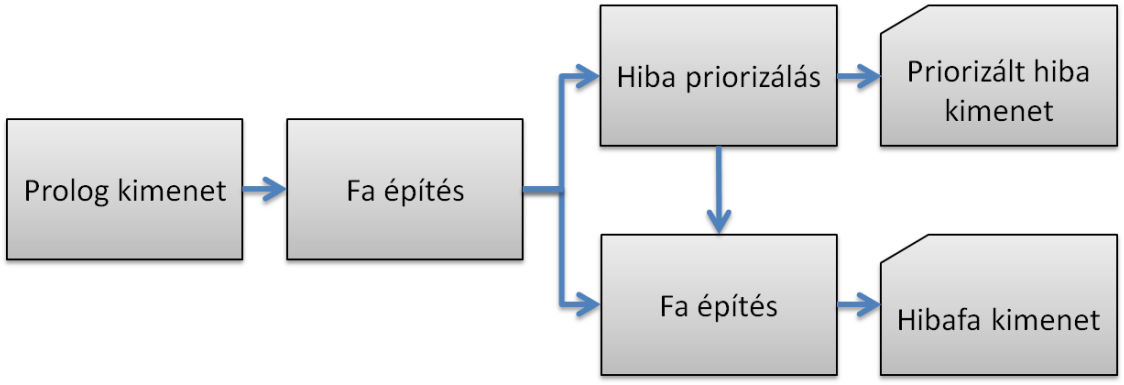
Végül az alsó szekcióban a korábban ismertetett értékeket – bemeneti, kimeneti hiba számok, valamint a minimum hiba súlyosság – adhatja meg a felhasználó.

A teljes analízis a GNU Prolog bemeneti fájl generálását, a GNU Prolog futtatását, majd a kimenet analízisét jelenti. Az első kettő lépést ismertettem, a prolog kimenetének értelmezését, abból kimenet generálást a következő alfejezetben mutatom be.



8-17 A megvalósított bővítmény futtatást biztosító run configuration ablaka

8.6. Kimenet készítése



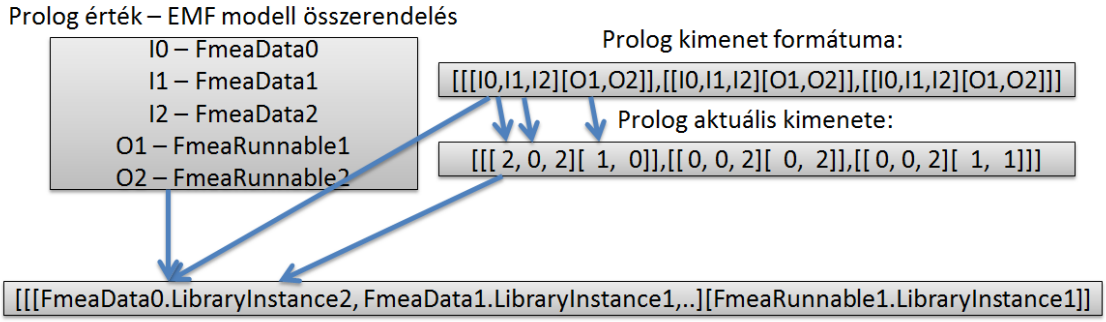
8-18 A kimenet készítésének blokkvázlata

A kimeneti hibafa készítéséhez a 8.4 fejezet végén mutatott GNU Prolog kimenetét kell feldolgozni, és könnyen értelmezhető formában a felhasználó számára biztosítani. A kimenet egy [[Bemenet1][Kimenet1],[Bemenet2][Kimenet2]] formájú szöveg, ahol a BemenetX, és KimenetX az FMEA komponensekhez rendelt változók értékeinek tömbjét jelentik. Ezek az értékek az adott komponens könyvtárában szereplő elemek indexei.

A feldologzás során a bemenetet Stringként olvassa be a program, majd ezt felbontja egy String tömbbe. Ebben a tömbben egymás után következik sorra a Bemenet1, Kimenet1, Bemenet2, Kimenet2 stb..

A Prolog bemeneti fájljának készítése során az FmeaData és FmeaRunnable elemekhez mint már írtam Prolog változók vannak rendelve. Ezeknek a sorrendje ismert, ez alapján pedig azonosítható, hogy a kimenetben lévő értékek melyik vizsgált komponens melyik könyvtár elemét jelentik.

Ezt az azonosítási algoritmust szemlélteti a 8-19 ábra.



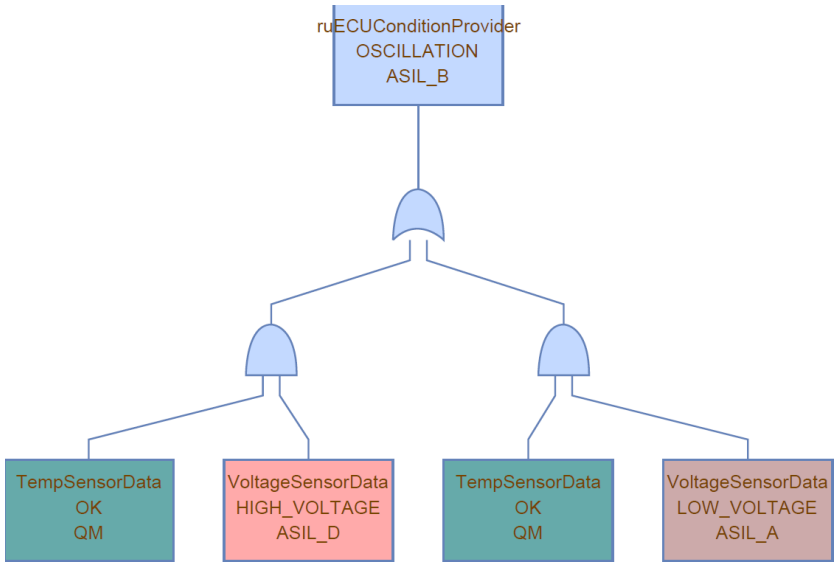
8-19 A prolog kimenet értelmezésének mechanizmusa

Az értékek értelmezése során minden BemenetX-KimenetX értéktömb párra felépít a program egy, vagy több fa struktúrát.

A Prolog kimeneti érték – EMF model hibakönyvtár elem összekapcsolással a szoftver a hibafában megjelenített objektumokat hoz létre. Ez a hibakönyvtár elemet, az őt tartalmazó elemet, valamint egy előfordulási számot foglal magába. Ezeket az objektumokat egy listában letárolja, ami ha már tartalmazza az adott elemet, akkor annak az előfordulási számát növeli.

A létrehozott listát első sorban ASIL értékek, másodsorban pedig előfordulás szerint sorba rendezve megkapom a bemeneti hibák prioritási listáját, ezt egy szöveges fájlba kiírva a felhasználó képet kaphat a hibák javításának javasolt sorrendjéről.

A fa építése során e listának az elemeire mutató referenciákat használok, így átadva egy-egy hiba előfordulási mennyiségét. A fa struktúrákat végül a JGraph bemeneteként megadva legenerálja a kimeneti fákat, megjelenítve a komponensek, valamint könyvtár értékük nevét, és a hiba súlyosságának értékét. A bemeneti hibák színezése a hiba javításának, megelőzésének fontosságát hivatott jelezni. Egy ilyen példa fát mutat a 8-20 ábra.



8-20 Kimeneti fa példa

8.7. FMEA módszer automatizálása

A feladat egyik célja, hogy az FMEA hibakeresési módszer lépései közül annyit tegyen automatikussá, amennyit csak tud. Ebből az irányból vizsgálom most a megoldást.

Az FMEA első lépése, a szoftver/termék vizsgálandó komponenseinek összegyűjtése, rendszerezése. A szerkesztő ezt a komponensek modelljének beolvasásával megvalósítja, a bemeneteket és kimeneteket reprezentáló modellelemeket összegyűjti, portok, illetve RunnableEntity-k alá rendeli.

A módszer második és harmadik lépése, azaz a komponensek közti kapcsolat, összeköttetés megadása, valamint a komponensek tulajdonságainak meghatározása szintén megtörténik a modell elemek beolvasása során.

A felhasználó ezután, a korábban bemutatott hiba könyvtár szerkesztő szekcióban létre tud hozni hibamódokat, és ezekhez súlyossági értékeket rendel. Az így létrehozott hibakönyvtárak felhasználásával készíthet igazságtáblákat a futtatható entitásokhoz. Ez a két lépés a felhasználó dolga, nincsenek automatikussá téve, a hibák, és kimeneteik meghatározása tehát az ő feladata.

A korábban látott kimeneti hibafát a szoftver az igazságtáblák, hibakönyvtárak alapján elkészíti a felhasználó számára.

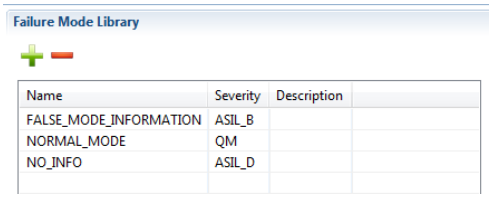
A módszer ezután a hibák priorizálását, majd javítását írja elő, ezután pedig az ismételt vizsgálatot. A hibák priorizálása megoldott, elhárításuk, a szoftver javítása ezután a fejlesztők, felhasználók feladata. Ez tehát azt jelenti, hogy a felhasználó beállításai alapján a futtatás során a szoftver priorizált eredményt ad vissza, ám ez a priorizálás nem teljesen automatikus, azonban nagyban megkönnyíti a hibák megtalálását.

A megvalósításom tehát a módszer azon részeit nem automatizálta, amelyhez mindenképp szükséges az emberi intelligencia. Ilyen a hibamódok meghatározása, ezek alapján igazságtáblák összerendelése, valamint a kimenetek megfelelő értelmezése.

9. Működés bemutatása

9.1. Modell

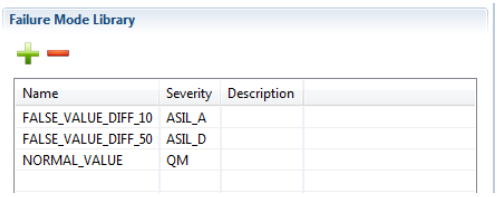
A korábban bemutatott Thermostat projekt EcuConditionLoggerCoordinator komponensét használom a működés bemutatásához. A szemléltetéshez hozzáadtam egy plusz portot, amelyen a hőmérő aktuális működési módját kapja meg a komponens, valamint egy futtatható entitást is létrehoztam, amely az ECU állapotának logolását végzi. Az átláthatóság érdekében erősen leegyszerűsített a példa.



9-1Az új bemeneti érték, az operationmode hibamódjai

Az operationMode a példámban azt adja meg, hogy a komponens éppen az ECU, vagy az ECU környezetének értékét kapja meg. Három hibamódot rendeltem hozzá:

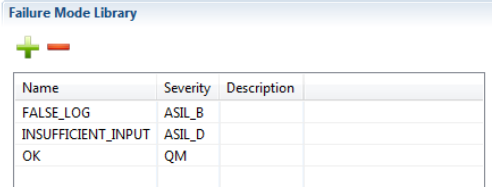
- NORMAL_MODE – ezesetben a rendes érték érkezik
- FALSE_MODE_INFORMATION – rossz módot közvetít az input
- NO_INFO – nem érkezik semmiféle információ arról, hogy honnan érkezik adat



9-2 A hőmérséklet érzékelő kimenetének értéke

A beérkező hőmérséklet szintén három hibamóddal rendelkezik:

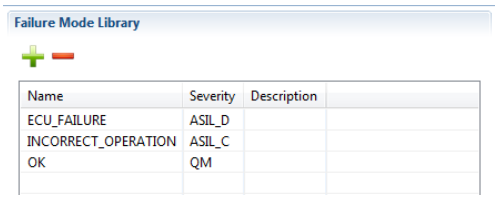
- NORMAL_VALUE – valós érték érkezik
- FALSE_VALUE_DIFF_10 – az érték legalább 10°C-szal eltér a valóstól
- FALSE_VALUE_DIFF_50 - az érték legalább 50°C-szal eltér a valóstól



9-3 Az ECU állapotáról logot készítő futtatható entitás hibamódjai

A logolás hibamódjainak értelmezése:

- OK – a logolás megfelelően működik
- FALSE_LOG – a logba hibás információk kerülnek
- INSUFFICIENT_INPUT – a bemeneti értékek hiányosak, a logba csak ez kerül



Name	Severity	Description
ECU_FAILURE	ASIL_D	
INCORRECT_OPERATION	ASIL_C	
OK	QM	

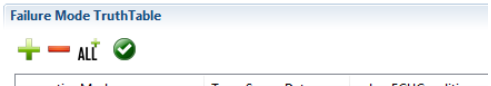
9-4 A hőmérséklet ellenőrző futtatható entitás hibamódjai

A hőmérséklet ellenőrzés hibamódjainak értelmezése:

- OK – a hőmérséklet ellenőrzése megfelelően működik
- INCORRECT_OPERATION – nem megfelelő az ellenőrzés, az ECU újraindítás elindulhat ok nélkül
- ECU_FAILURE – az ECU teljesen hibás működését jelzi

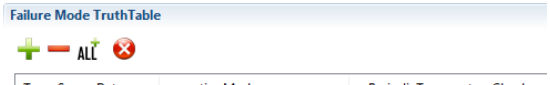
9.2. Megadott igazságtábla

A létrehozott hibamódok felhasználásával létrehoztam a két bemenettel rendelkező futtatható entitáshoz hibamód táblákat. Ezek egyszerűen értelmezhetőek az imént ismertetett hibamód jelentésekkel.



operationMode	TempSensorData	ruLogECUCondition
FALSE_MODE_INFORMATION	X	FALSE_LOG
NORMAL_MODE	FALSE_VALUE_DIFF_10	FALSE_LOG
NORMAL_MODE	FALSE_VALUE_DIFF_50	FALSE_LOG
NORMAL_MODE	NORMAL_VALUE	OK
NO_INFO	X	INSUFFICIENT_INPUT

9-5 A logolásért felelős futtatható entitás igazságtáblája



TempSensorData	operationMode	ruPeriodicTemperatureChecker
FALSE_VALUE_DIFF_10	FALSE_MODE_INFORMATION	INCORRECT_OPERATION
FALSE_VALUE_DIFF_10	NORMAL_MODE	OK
FALSE_VALUE_DIFF_10	NO_INFO	ECU_FAILURE
FALSE_VALUE_DIFF_50	FALSE_MODE_INFORMATION	ECU_FAILURE
FALSE_VALUE_DIFF_50	NORMAL_MODE	INCORRECT_OPERATION
FALSE_VALUE_DIFF_50	NO_INFO	ECU_FAILURE
NORMAL_VALUE	NORMAL_MODE	OK

9-6 A hőmérséklet ellenőrzését végző futtatható entitás igazságtáblája

9.3. Analízis

Ahhoz, hogy az analízis hasznosságát, valamint működésének helyességét bemutassam, több különböző beállítással futtatom le a fent ismertetett modellre. A komponens és az igazságtáblái mérete alapján az analízisek kézzel is elvégezhetők, így meg tudom mutatni, hogy a kimenetük megfelelő eredményt kínál.

Alapbeállításokkal végzett analízis során a bemeneti és kimeneti hibák közül akármennyit valamint akármilyen súlyossággal elfogadok. Tulajdonképp az eredmény nem más, mint a megadott igazságtáblák által definiált fák, tehát minden egyes kimenet minden megadott bemeneti kombinációja. A hibafa kimenet ez esetben átláthatatlan, és nem tartalmaz ténylegesen hasznos információt. A másik kimenet, a generált riport viszont megadja a hibamódok prioritási listáját.

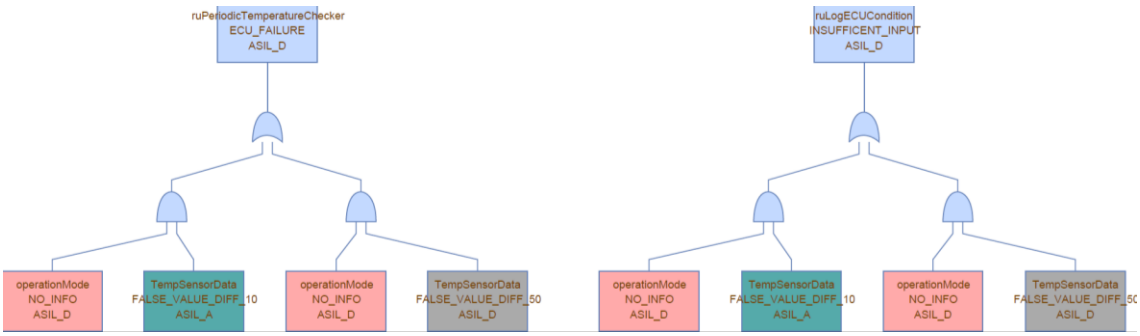
DataType	FailureMode	Sever-ity	Occurence
TempSensorData	FALSE_VALUE_DIFF_50	ASIL_D	6
operationMode	NO_INFO	ASIL_D	4
operationMode	FALSE_MODE_INFORMATION	ASIL_B	4
TempSensorData	FALSE_VALUE_DIFF_10	ASIL_A	6
operationMode	NORMAL_MODE	QM	6
TempSensorData	NORMAL_VALUE	QM	2

2. Táblázat A generált report fájl tartalma

Ahogy látható, a táblázat megadja, hogy a hibajavítási munkák esetén a hőmérséklet szenzor kimeneti adat hibás értékének kezelése kellene legyen az első lépés. Ez helyesnek mondható, hiszen a legsúlyosabb ASIL értéket rendeltem hozzá, ebből fakadóan a legtöbb komoly hiba forrásaként szerepel is az igazságtáblákban.

A következő két analízissel a célom megmutatni, hogy a bemeneti és kimeneti hibák számára és súlyosságára való szűréssel értékes, és helyes információkat nyerhetek a program használatával.

Első esetben a komponens olyan bemeneti kombinációját keresem, amely pontosan kettő, legalább ASIL_C kimenetet eredményez a komponens futtatható entitásai között, azaz mindkettő kimenetben súlyos hibát eredményez.

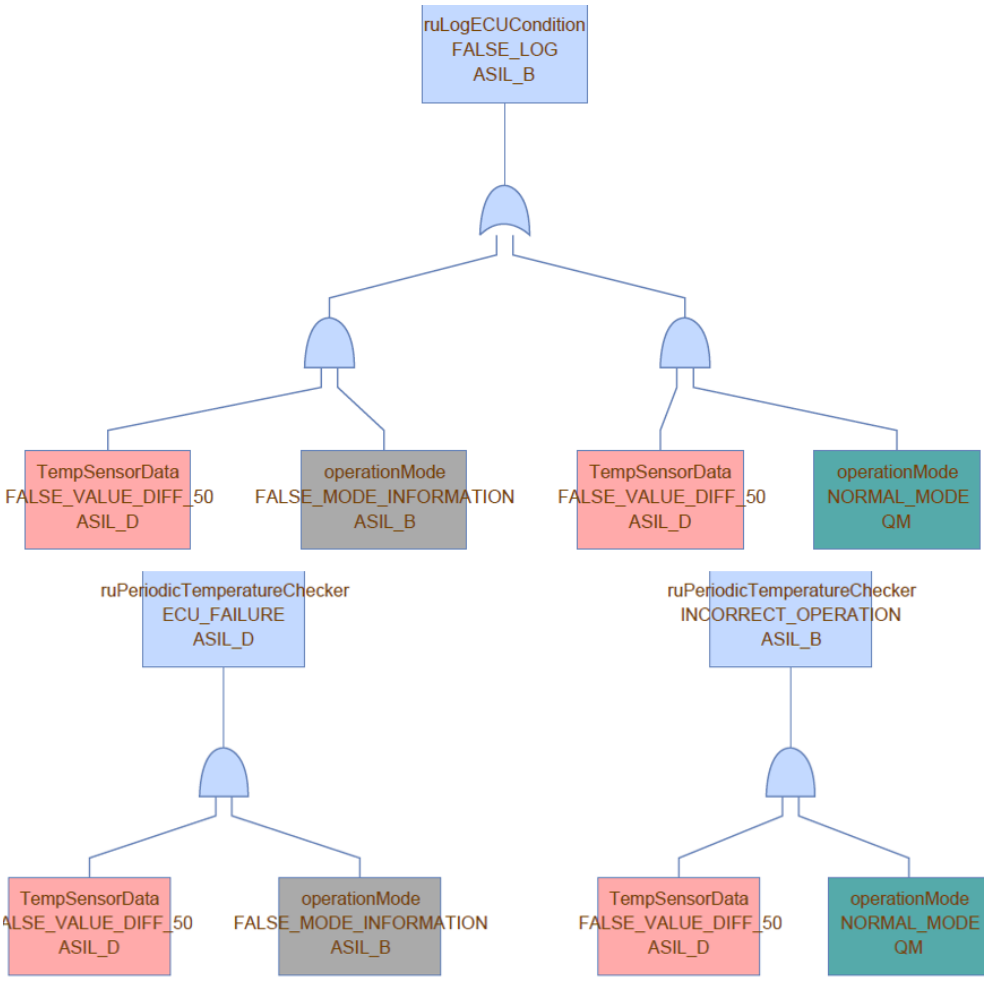


9-7 Kimeneti érték ASIL_D-re szűrés

A kimeneti hibafákból egyértelműen kiderül, hogy csak két ilyen bemeneti kombináció létezik. A hőmérő módjáról nem érkezik információ, valamint az érték is több mint 10 vagy 50°C-al eltér a valóstól. Ezek mind a hőmérséklet ellenőrző mind pedig a log készítő futtatható entitásban komoly, ASIL_D súlyosságú problémát okoznak.

Amennyiben a bemeneti adatokat, és a hibamódokat összehasonlítom kézzel, láthatóan nincs másik helyes megoldása ennek a szűrésnek.

A következő analízisben azt vizsgáltam, hogy legalább egy, ASIL_C súlyosságú hiba a bemeneti értékekben milyen kimenetet eredményezhet.



9-8 Legalább egy ASIL_C súlyosságú bemenet által okozott kimenetek

Az analízisből kiderül, hogy a hőmérséklet szenzor komoly hibája – a továbbított hőmérséklet jobban mint 50°C-al eltér a valóstól – esetén keletkezik a kimenetben valamilyen hiba. Minden esetben a logolás hibás kimenet mentését végzi, a hőmérséklet ellenőrzésnél azonban két különféle hiba léphet fel, annak függvényében, hogy a működési módról kapott információ helyes-e.

Az analízist így lefuttatva a kombinációkra a fejlesztők jobb rálátást kapnak a hibák kimeneteléről, hatásairól a komponensekre.

10. Eredmények értelmezése, továbbhaladás

A diplomatervezés során megismerkedtem több hasznos, az iparban is használható alkalmazással, valamint olyan technológiákkal, amik a későbbi munkáim során hasznos tudást jelentenek majd. Többek közt az autóipari szoftverfejlesztés során egyre szélesebb körben használt AUTOSAR szabvány, valamint az erre épülő AUTOSAR Architect és a segítségével történő szoftver elkészítésnek módszertana megismerése jelentették a munkám nagy részét. A hibakeresési módszerek közül a hibafával és a hibamód és effekt analízissel ismerkedtem, majd a kettőt kezdetleges szinten össze is kapcsoltam, így az FMEA eredményei alapján hibafák kirajzolását lehetővé téve.

A megvalósított Eclipse bővítmény lehetőséget nyújt egy-egy adott szoftver komponens egyszerű hibamodelljének létrehozására, azaz hibamódok hozzárendelésére annak adatelemeihez. A hibamodell alapján Prolog segítségével megoldja – a felhasználó által definiált paraméterek szerint – az igazságtáblákból generált véges domain problémákat, majd grafikus hibafa kimenetet ad az eredményről, valamint kezdetleges riporttal is szolgál a felhasználók számára. Ez kezdeti hibaanalízishez megfelelő alapot nyújt, rendszerszintű hibaterjedés azonban még nem számítható vele.

A továbbhaladás egyik lehetősége ebből fakadóan az analízis eltolása az egész rendszer analízise felé, ennek első lépése a szoftver komponensek összeköttetései alapján egy kompozíció meghibásodásainak vizsgálata. Érdekes kérdés lehet ezután a megoldás skálázódása a kompozíciók és az őket alkotó komponensek számának, bonyolultságának, azaz a modell méretének növekedésével. Nagy modell esetén elképzelhető, hogy a CSP megoldás keresésnél egy-egy direktíva használat nagyot gyorsíthat a program futtatásán.

A bővítmény jelenleg egy prototípus, ami még finomításra szorul, funkcionalitás és felépítés szempontjából is. Lényeges, hogy a további módosításokat a biztonsági analízist végző mérnökökkel együttműködve, az ő meglátásaik alapján végezzem.

Irodalomjegyzék

[1] Jos Warmer, Wim Bast, Anneke G. Kleppe
MDA Explained: The Model Driven Architecture : Practice and Promise
Addison-Wesley Professional, 2003
1. fejezet - The MDA Developement Process
1-11.oldal

[2] Failure Modes and Effects Analysis
http://www.npd-solutions.com/fmea.html
Hozzáférés dátuma: 2017.05.09.

[3] Automotive Safety Integrity Level
https://en.wikipedia.org/wiki/Automotive_Safety_Integrity_Level
Hozzáférés dátuma: 2017.05.09.

[4] Dr. Abonyi János, Dr. Fülep Tímea
Biztonságkritikus rendszerek
Pannon Egyetem, 2014
3.2 fejezet Biztonsági integritás – SIL és ASIL értékek

[5] Basics: AUTOSAR
http://www.autosar.org/about/basics/
Hozzáférés dátuma: 2017.05.09.

[6] Darko Durisic, Miroslaw Staron, Matthias Tichy
ARCA - Automated Analysis of AUTOSAR Meta-Model changes
Publikálás dátuma: 2015.05.16.
3. o. - 5. ábra
http://www.cse.chalmers.se/~miroslaw/papers/2015_ARCA_Darko.pdf
Hozzáférés dátuma: 2017.05.14.

[7] dr. Pintér Gergely
AUTOSAR alapú autóipari rendszerek előadás
Budapesti Műszaki és Gazdaságtudományi Egyetem - 2017.02.15.
AUTOSAR alapú autóipari szoftverrendszerek - Szoftverkomponensek
4-13. dia

[8] Technical Overview: AUTOSAR
http://www.autosar.org/about/technical-overview/.
Hozzáférés dátuma: 2017.05.14.

[9] AUTOSAR Basic Software
http://www.autosar.org/about/technical-overview/ecu-software-architecture/autosar-basic-software/.
Hozzáférés dátuma: 2017.05.09.

[10] AUTOSAR Runtime Environment
http://www.autosar.org/about/technical-overview/ecu-software-architecture/autosar-runtime-environment/
Hozzáférés dátuma: 2017.05.09.

[11] Eclipse Platform Technical Overview
http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html
Hozzáférés dátuma: 2017.05.15.
Part I: Eclipse Platform Technical Overview

[12] What are extensions and extension points?
https://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points%3F
Hozzáférés dátuma: 2017.05.09.

[13] Eclipse Modeling Framework (EMF)
https://eclipse.org/modeling/emf/
Hozzáférés dátuma: 2017.05.09.

[14] EclipseCon2008 - Fundamentals of EMF
Előadás dátuma - 2008.03.17.
19. dia
http://www.eclipse.org/modeling/emf/docs/presentations/EclipseCon/EclipseCon2008_309T_Fundamentals_of_EMF.pdf
Hozzáférés dátuma: 2017.05.14.

[15] EMF Client Platform
http://www.eclipse.org/ecp/
Hozzáférés dátuma: 2017.05.14.

[16] JFace
https://wiki.eclipse.org/JFace
Hozzáférés dátuma: 2017.05.14.

[17] Model-View-Controller
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
Hozzáférés dátuma: 2017.05.09.

[18] Defining custom launcher for the Eclipse IDE - Tutorial
http://www.vogella.com/tutorials/EclipseLauncherFramework/article.html
Hozzáférés dátuma: 2017.05.09.

[19] Zhe Liu - Thesis
Algorithms for Constraint Satisfaction Problems
University of Waterloo, 1998.
Chapter 1.1 - What is CSP?

[20] Zhe Liu - Thesis
Algorithms for Constraint Satisfaction Problems
University of Waterloo, 1998.
Chapter 1.2.1 - What is CSP?
2. o. - Figure 1.1

[21] Zhe Liu - Thesis
Algorithms for Constraint Satisfaction Problems
University of Waterloo, 1998.
Chapter 1.2.1 - Tree Search
10. o. - Figure 1.5

[22] Peter Norvig, Stuart Russell
Artificial intelligence - A modern approach - 2. kiadás
Chapter 5 - Constraint satisfaction problems
137-140. o.

[23] Peter Norvig, Stuart Russell
Artificial intelligence - A modern approach - 2. kiadás
Chapter 5 - Constraint satisfaction problems
143. o. - Figure 5.5

[24] GraphStream Getting Started
http://graphstream-project.org/doc/Tutorials/Getting-Started/
Hozzáférés dátuma: 2017.05.09.

[25] Graphviz - Graph Visualization Software
http://www.graphviz.org/Documentation.php
Hozzáférés dátuma: 2017.05.09.

[26] The DOT language
http://www.graphviz.org/content/dot-language
Hozzáférés dátuma: 2017.05.09.

[27] mxgraph
https://github.com/jgraph/mxgraph
Hozzáférés dátuma: 2017.05.09.

[28] HTML5 canvas path tutorial
http://www.html5canvastutorials.com/tutorials/html5-canvas-paths/
Hozzáférés dátuma: 2017.05.09.

[29] Daniel Diaz
GNU-Prolog Manual
http://www.gprolog.org/manual/gprolog.html#sec3.
Hozzáférés dátuma: 2017.05.16.