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

Elosztott biztonságkritikus rendszerek modellvezérelt fejlesztése

TDK DOLGOZAT

Készítette:
Horváth Benedek
Konnerth Raimund-Andreas
Mázló Zsolt

Konzulens:
Vörös András
Dr. Horváth Ákos

Külső konzulens
Jász Zoltán
(Ericsson Magyarország Kft.)

2014. október 27.

Tartalomjegyzék

1. fejezet

Bevezetés

Napjainkban a biztonságkritikus rendszerek az élet minden területén megjelentek, kezdve a kisméretű szívritmus-szabályozóktól egészen a hatalmas repülőgépekig. Ezen rendszerek komplexitása az elmúlt évek során rohamosan növekedett, pl. egy Airbus A380-as repülőgépen több millió sornyi kód felelős az utasok biztonságáért. Az összetett rendszerek elvárt funkcionalitásának teljesítéséhez, a bonyolultságukból adódóan, elosztott működés szükséges, melynek megtervezése egyre nagyobb kihívás elé állítja a fejlesztő mérnököket. Ez különösen igaz a biztonságkritikus rendszerekre, melyeknél elvárt a helyes működés: ezt alapos mérnöki tervezés és tesztelés mellett precíz matematikai ellenőrző módszerek segítségével lehet már csak garantálni (DO-178C [1] DO-333 kiegészítés [2], EN 50128 [3]).

Egy lehetséges megoldásként az elmúlt években meghatározó paradigmává vált a modell alapú rendszerfejlesztés biztonságkritikus rendszerek esetén. A metodika célja, hogy a rendszer elkészítése során már a korai fázisoktól, magasszintű modellekből kiindulva, finomítási lépéseken keresztül legyünk képesek automatikusan származtatni a rendszer egyes komponenseit, például a konfigurációt, a dokumentációt, forráskódot vagy akár tanúsítványozási bizonyítékokat is. A fejlesztés során elkészült modellek segítségével pedig akár már a fejlesztés korai fázisaiban lehetőség nyílik a tervek helyességének vizsgálatára formális módszerek segítségével. Ehhez szükség van olyan keretrendszerekre, amelyek képesek a mérnöki modellekből az analízis modellek automatikus előállítására.

Elmondhatjuk, hogy a modell alapú fejlesztés módszertana próbál megoldást nyújtani az elosztott, biztonságkritikus rendszerek tervezése során felmerülő problémákra. Sajnos azonban a kezdeti sikerek ellenére sok akadály nehezíti a módszertan alkalmazását. Munkánk során ezekkel a problémákkal küzdöttünk meg a saját esettanulmányunk megvalósításakor is. Az esettanulmány kiválasztásánál fontos szempont volt, hogy komplexitását tekintve közelítsen az ipari rendszerekhez: egy elosztott modellvasút rendszert terveztünk meg, mind a hardver, mind a szoftver komponenseket.

Az elkészített esettanulmány a vasúti szakterület problémáit szemlélteti, melyben az elosztott biztonsági logika feladata a vonatok ütközésének megakadályozása egyszerű szenzorok és beavatkozók segítségével. Azonban a vasúti szakterületnél nehezebb problémát kellett kezelnünk, mivel az irányítás többféle lehetőséget biztosít a vonatok mozgatására az esettanulmányunkban, mint a valós életben. Emellett fontos szempont volt, hogy nem központosított, hanem egy komplex,

elosztott biztonsági logikát dolgozzunk ki, amely a komponensek lokális megfigyelései alapján dönt a lehetséges beavatkozásról. Emiatt különösen fontos volt a logika megtervezése és a megtervezett logika formális vizsgálata.

A hardvert „commercial off-the-shelf" (COTS) komponenseket felhasználva terveztük meg úgy, hogy a vonatokat irányító rendszertől független beavatkozó rendszert ne veszélyeztesse az irányítás esetleges meghibásodása. Ezzel próbáltuk a biztonságkritikus rendszerek tervezésekor használt külön irányítási és biztonsági kör tervezési mintát megvalósítani. A hardver megtervezése során az integráció problémáját is meg kellett oldani, amihez többek között szükség volt saját nyomtatott áramkörök tervezésére is.

Az elosztott szoftverrendszer tervezését egy iparilag releváns, modell alapú szoftverfejlesztő eszközben végeztük, amely lehetőséget biztosított a modellek szimulációjára és a kódgenerálásra is. A tervezés során a magas szintű követelményeket finomítottuk, dekomponáltuk, és így állítottuk elő a szoftver modelljét. A modellek komplexitását jól jellemzi, hogy fejlesztésük során a tervezőeszköz több hibájára is fényt derítettünk, amelyeket elküldtünk a fejlesztőknek, megerősítést kapva feltételezéseinkre. Ezenkívül egy meghatározó telekommunikációs cég is technikai támogatást nyújtott a fejlesztő eszköz használatában, mert náluk is hasonló módon fejlesztik a komplex elosztott kommunikációs rendszereket.

A modell alapú szoftverfejlesztés támogatására modelltranszformációkat definiáltunk, amelyek megvalósítják a mérnöki modellek formális modellekké történő leképezését. Választásunk egy nyílt forráskódú, OMG szabvány nyelvre és az xtUML modellező nyelvre esett: mindkét esetben formalizáltuk a modellező nyelv elemeit a véges automata formalizmus segítségével és megadtuk a leképezési szabályokat.

A transzformációk felhasználásával leképeztük az esettanulmány modelljeit formális modellekké, elkészítettük a környezet egyszerűsített modelljét és a komponensek tulajdonságait formálisan ellenőriztük.

Munkánk célja egy olyan publikusan elérhető esettanulmány elkészítése volt, ami egyrészt szemlélteti a biztonságkritikus elosztott rendszerek modell alapú fejlesztését, másrészt a definiált modelltranszformációk segítségével bemutatja, hogy a formális módszerek a fejlesztés korai fázisaitól tudják támogatni a helyes rendszerek tervezését.

Az alkalmazott módszertan hatékonyságát jól mutatja, hogy az elkészült rendszer a Kutatók éjszakája rendezvényen is bemutatásra került.

A dolgozat első részében, a 2. fejezetben ismertetjük a modell alapú tervezéshez, fejlesztéshez és verifikációhoz szükséges háttérismereteket, majd a 3. fejezetben bemutatjuk az általunk fejlesztett modelltranszformációs eszközök működését egyszerű példákon keresztül. A 4. fejezetben az elkészített esettanulmány kerül részletes bemutatásra. Végül az 5. fejezetben az elért eredményeinket, következtetéseinket és jövőbeli terveinket fogalmazzuk meg.

2. fejezet

Háttérismeretek

A rendszertervezés a több szakterület együttműködését igénylő feladatok megoldására nyújt lehetőséget. Ez magában foglalja az üzleti és technológiai folyamatokat, melyek szükségesek a megoldások eléréshez és a projekt sikerességét befolyásoló kockázatok mérsékléséhez.

Az üzleti folyamatok a fejlesztési költségek, ütemterv, illetve a technológiai célok elérésnek biztosítása miatt szükségesek, míg a technológiai folyamatok magukban foglalják a rendszer specifikációját, megtervezését, elkészítését és az elkészült rendszer tesztelését, ellenőrzését, illetve verifikációját is [4].

2.1. Modell alapú rendszertervezés

1. Definíció (Modell). *A modell a valóság egy részének egyszerűsített képe, amely a modellezendő tulajdonságokat megtartja, a többi információt pedig leegyszerűsítve vagy nem ábrázolja.*

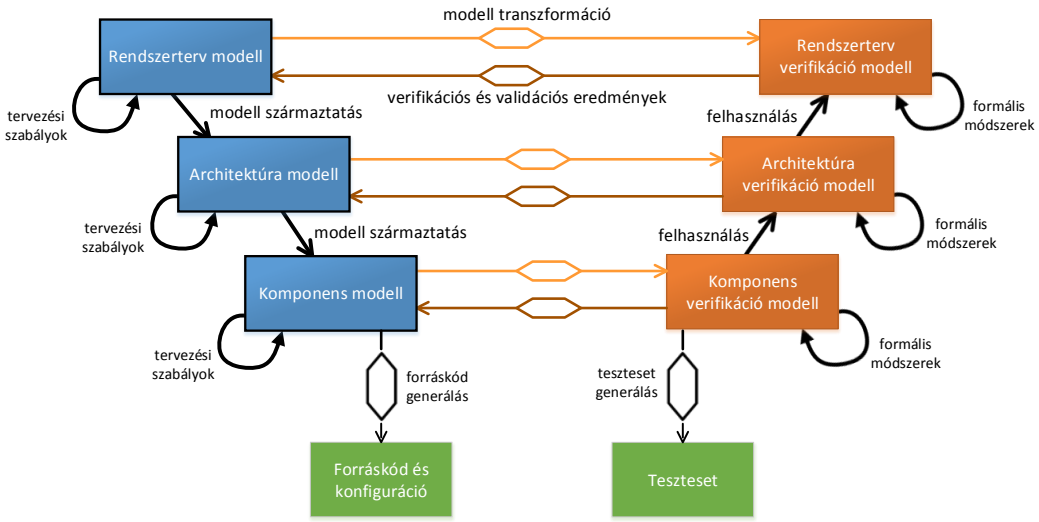
2. Definíció (Modell alapú rendszertervezés). *A modell alapú rendszertervezés egy formalizált alkalmazása a modellezésnek a rendszer követelményekre, tervezésre, analízisre, verifikációs és validációs tevékenységekre vonatkozóan. Az elsőszámú artifaktum a modell, amely a fejlesztés minden fázisában a legfőbb információ hordozó médium [5].*

A modell alapú megközelítésben a hangsúly a részletes elektronikus dokumentációk készítéséről a rendszer koherens modelljének elkészítésére, kezelésére és a modellel ekvivalens platformfüggő kód és dokumentáció generálására helyeződik. Ezáltal a rendszer komplexitása kezelhetővé, a rendszer hibamentes működése könnyebben bizonyíthatóvá válik.

A korábbi megközelítésben alkalmazott nagymennyiségű dokumentáció helyett komponens- és rendszerszintű modellek készülnek, melyekben egy helyen tárolódnak a komponensek belső működését leíró információk, a vonatkozó kényszerekkel együtt.

A modell alapú tervezés legnagyobb előnye, hogy a platformfüggetlen modellekből, kódgenerátor felhasználásával – szoftverrendszerek esetén – automatikusan származtatható a modellek viselkedését megvalósító forráskód és konfiguráció. Ezáltal a forráskód és a modell konzisztenciája biztosítható és a forráskód-implementáció során elkövetett emberi hibák minimalizálhatók.

A modell alapú tervezés további előnye, hogy a modellekből automatikusan generálhatóak a szöveges dokumentációk, melyek karbantartása a modellek karbantartásával egyidőben, auto-



2.1. ábra. Modell alapú rendszertervezés Y-modell részlet

matikusan megtörténik. Ezáltal nem lesz ellentmondás a szöveges dokumentumok és a rendszer működését leíró modellek között, viszont a dokumentáció egy részét közvetlenül illeszteni kell tudni a modellekhez, ami további költségeket jelenthet.

A modell alapú tervezés kiemelt fontossággal rendelkezik biztonságkritikus rendszerek (pl. repülőgép- és vasútirányítás) esetén, ahol komoly figyelmet igényel a rendszer helyes működésének és a balesetek elkerülésének biztosítása. Emiatt rendkívül szigorú szabványok (DO-178C [1], DO-278A [6], DO-331 [7]) vonatkoznak a rendszerekben alkalmazható szoftver- és hardver megoldásokra. Köztük a repülőgépiparban alkalmazott DO-331 szabvány, amely tartalmazza a repülőgép-ipari rendszerek modell alapú fejlesztésére vonatkozó szabályokat.

2.2. Modell alapú rendszertervezés folyamata

Modell alapú megközelítést a tervezés során gyakran biztonságkritikus rendszerek esetén alkalmaznak, ahol szükséges a rendszer működésének validációja (követelményeknek való megfelelésük ellenőrzése), a tervezés minden fázisában. Ezért a rendszertervezés folyamata során tipikusan a szoftver- és rendszerfejlesztésben elterjedt V-modellnek [8] egy kiegészített változatát, a 2.1 ábrán látható Y-modellt alkalmazzák.

Az Y-modellben a V-modellhez hasonlóan a rendszer működésével szemben állított komplex követelmények teljesítéséhez egy több szintre tagolódó tervezési folyamat valósul meg, melynek legfelső szintjén a magasszintű követelmények analízise helyezkedik el. Ennek során a rendszer működésével kapcsolatos követelmények pontosítása, összegyűjtése és rendszerezése történik.

A második szinten a követelményanalízis alapján előállított rendszerterv áll, amely tartalmazza a rendszert felépítő különböző architektúrák összekapcsolódásából álló konstrukciót, melynek feladata a magasszintű követelmények teljesítése.

A harmadik szinten a rendszert felépítő, egymással együttműködő heterogén architektúrák terve helyezkedik el. Az egyes architektúrák belső szerkezete egymástól eltérő lehet, ezek pontos leírását az őket alkotó komponensek tervei tartalmazzák.

Az architektúra különböző felelősségi- és szerepkörökkel rendelkező komponensekre való dekompozíciójával csökkenthető a kezdeti probléma (a rendszerrel szemben állított magasszintű követelmények) komplexitása és a komponensek minél részletesebb megtervezése által javítható a rendszer ellenőrizhetősége.

Az Y-modellben a V-modellhez hasonlóan minden szinten, az elkészült tervekhez kapcsolhatóak verifikációs és validációs lépések. Ezen lépések során formális módszerek és az alacsonyabb szinteken hozott verifikációs eredmények felhasználásával ellenőrizhető az adott szinten elkészült tervek helyessége. A verifikációs és validációs folyamatok által adott eredmények visszavezethetőek a hozzájuk tartozó tervekhez, ezáltal iteratívan javítható azok helyessége.

Az Y tervezési metodika minden szintjén a modell alapú szemantikához tartozóan modellek tartalmazzák az egyes terveket. Ennek köszönhetően az egyes komponensek megtervezése után a verifikált komponensek működését megvalósító, platformspecifikus forráskód és konfiguráció, kódgenerátorok felhasználásával automatikusan származtatható. Az egyes verifikációs modellekből teszteset generátorokon keresztül származtathatóak a különböző tesztesetek, melyekkel ellenőrizhető a generált kódok helyes működése.

A V-modellnek a kódgenerátorral és a formális verifikációval való kiterjesztésével elkerülhetőek az implementáció során a programozói hibák és tévedések, melyek a verifikált komponensek helyességét elronthatják. Ezáltal javítható a tervezés minősége és csökkenthetőek az implementációs költségek.

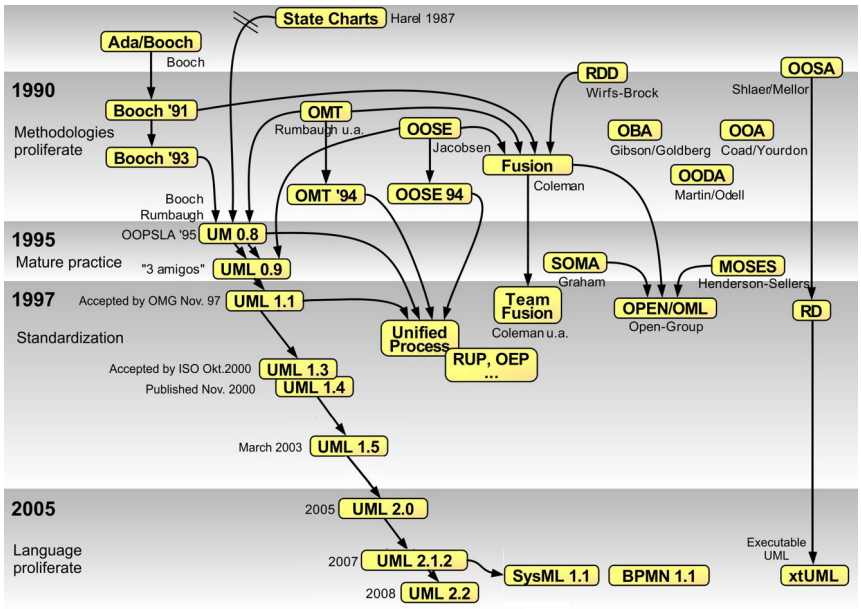
2.3. Szoftverrendszerek modell alapú fejlesztése

Szoftverrendszerek modell alapú fejlesztéséhez szükség van szabványos modellezési nyelvekre. Egy szabványos modellezési nyelvet többek között az alábbi négy tulajdonság jellemez:

- *Absztrakt szintaxis*: a nyelv elemkészletét és az elemek egymáshoz kapcsolódását írja le. Az absztrakt szintaxis a nyelv platformfüggetlen leírását teszi lehetővé.
- *Konkrét szintaxis*: a nyelv ábrázolásmódja.
- *Formális szemantika*: a nyelvi elemek jelentése.
- *Jólformáltsági kényszerek*: az absztrakt szintaxis által nem leírt követelmények a nyelvhez kapcsolódóan.

Szoftverfejlesztésben az OMG (*Object Management Group*) által definiált különböző modellezési nyelvek, köztük az egységes modellezési nyelv, az UML (*Unified Modeling Language* [9]), vagy a rendszerek modell alapú tervezésére használt SysML (*Systems Modeling Language* [10]), szabvánnyá váltak az évtizedek során.

Az UML nyelv 1.0-s változata 1997-ben jelent meg, jelenleg legújabb verziója a 2.4.1, amely 2011-ben jelent meg. A nyelv absztrakt szintaxisa, a nyelv metamodellje, a MOF (*Meta Object*



2.2. ábra. Modellezési nyelvek fejlődése

Facility). A konkrét szintaxist az egyes UML diagram reprezentációk jelentik. Jólformáltsági kényszereket a nyelvhez kapcsolódóan OCL-ben (*Object Constraint Language*) lehet megadni. Az UML formális szemantikáját nem definiálták, helyette egy, az UML-ből származtatott nyelv, az fUML [11] tartalmazza ezt a leírást.

A 2.2 ábrán [12] látható az UML-hez kapcsolódó modellezési nyelvek fejlődése az 1990-es évektől kezdve. Az ábrán megfigyelhető, hogy az UML-lel párhuzamosan, attól függetlenül fejlődött a *Shlaer-Mellor módszer*, melyet a 2.3.3 szakaszban mutatunk be.

2.3.1. fUML

Az fUML (*Semantics of a Foundational Subset for Executable UML Models*) bevezetése előtt az UML modellekben a viselkedést definiáló részeknek (pl. tokenek, vezérlési csomópontok) nem volt egyértelmű, szemantikus leírása. Ennek következtében az említett modellelemek inkonzisztens használata gyakran félreérthetővé, többértelművé tette a modelleket, ami megnehezítette a szabványos modell alapú tervezés alkalmazhatóságát.

A felmerülő problémák megoldására, 2010-ben az OMG létrehozott egy új szabványt, az fUML-t. A szabványban definiálta a viselkedést leíró modellelemek formális szemantikáját, ezáltal megszüntetve a korábban említett többértelműséget és inkonzisztenciát. Ez alapján az fUML a 2.3 szakaszban említett tulajdonságok szerint szabványos modellezési nyelvvé egészítette ki az UML egy részhalmazát [13].

2.3.2. Alf

Az fUML aktivitás diagramok absztrakt végrehajtási szemantikájának megadásával lehetővé vált az UML által definiált elemkészlet egy részének platformfüggetlen modelleken (*Platform Independent Models*) történő formális végrehajtása és verifikációja. A platformfüggetlen modellek

viselkedésének leírásához szükség van egy, a modell implementációjától független viselkedésleíró nyelvre [11].

Egy lehetséges akciónyelv az OMG által definiált Alf (*Action Language for Foundational UML*) [14]. Az Alf konkrét szintaxisa alapján a *Papyrus UML* elkészítette a nyelv absztrakt szintaxisát, az elterjedt, nyílt-forráskódú modellezési környezettel, az EMF-fel (*Eclipse Modeling Framework*[1]) kompatibilis metamodelljét[2] és a metamodellhez egy példánymodellek készítésére alkalmas szerkesztőt. Ezáltal lehetővé vált Alf kódok modell alapú leírása és transzformációja.

2.3.3. Shlaer-Mellor módszer

A Shlaer-Mellor módszer (*Shlaer-Mellor method* [15]) eredetileg objektum-orientált rendszerek elemzésére, tervezésére és implementációjára használt eljárás, amely az alábbi részekből áll:

1. Objektum-orientált analízis (*Object-Oriented System Analysis, OOSA*):
   (a) A rendszer komponensekre bontása.
   (b) Részletes objektum-orientált analízis elvégzése minden komponensre.
   (c) Komponensek analízisének verifikációja.
2. Rekurzív tervezés (*Recursive Design*):
   (a) OOSA modellek transzformációs szabályainak meghatározása.
   (b) Transzformációt végző komponensek elkészítése.
   (c) OOSA modellek transzformálása platformfüggő implementációra.

Az objektum-orientált analízis során a rendszert alkotó komponenseket osztályoknak tekintve megfeleltethetőek egy-egy platformfüggetlen modellnek, melyek viselkedése és tulajdonságai az eredeti osztály viselkedésével és tulajdonságaival egyezik meg. A platformfüggetlen modelleken a részletes, strukturális analízist és verifikációt elvégezve megismerhetővé válik belső szerkezetük és a verifikációnak köszönhetően működésük helyessége bizonyítható.

A platformfüggetlen modellt (*Platform Independent Model, PIM*) a rekurzív tervezési fázisban definiált transzformációs szabályok alapján platformspecifikus modellé (*Platform Specific Model, PSM*) alakítva, a platformspecifikus modellből egy kódgenerátor segítségével automatikusan előállítható az implementáció.

Az automatikus kódgenerálás következtében a forráskód-implementáció során elkövetett emberi hibák mértéke minimalizálható és a részletes analízisnek köszönhetően a rendszerstruktúra megismerhető.

2.3.4. Executable UML

Az Executable UML (*xtUML*) mérnöki modellek készítésére definiált modellezési nyelv, amely a *Shlaer-Mellor módszerből* fejlődött tovább. A nyelv elemkészletét a a 2.1 táblázat tartalmazza.

[1] http://eclipse.org/modeling/emf/
[2] http://www.eclipse.org/papyrus/

A modellben a valóságból absztrakcióval származtatott tulajdonságokat az objektum-orientált tervezéshez hasonlóan osztályokba szervezve, az egyes osztályokban az adatokat attribútumokként tárolva, az osztályok kapcsolatát asszociációval leírva lehet a modellezett valóság strukturális ábrázolását elvégezni.

Az osztályok különböző események hatására a tárolt attribútumok értékeiben bekövetkező változásait állapotátmenetekként értelmezve lehet az adott osztályt életciklusát ábrázolni. Az állapotváltozásokat kiváltó eseményeket egy viselkedést leíró nyelven megfogalmazva, az osztályok életciklusával együtt lehet az osztályok viselkedését dinamikai tartományban vizsgálni.

A Shlaer-Mellor módszerhez kapcsolódva az xtUML nyelv segítségével megvalósítható egy rendszer platformfüggetlen modelljének analízise és a platformfüggetlen modellből a viselkedést leíró nyelv alapján egy kódgenerátor segítségével a platformfüggő modell előállítása. Ezáltal megvalósítható egy komplex rendszer modell alapú analízise, tervezése és automatizált implementációja.

Az Executable UML metodikát követő, mérnöki modellek tervezésére ipari környezetben használt eszköz a BridgePoint, melyet a 2.3.5 szakaszban mutatunk be röviden.

Valóság szegmens	**Absztrakciós szint**	**Modellbeli megjelenítés**
adat	osztály attribútum asszociáció	UML osztálydiagram
vezérlés	állapot esemény állapotátmenet eljárás	UML állapottábla diagram
végrehajtás	akció	viselkedést leíró nyelv

2.1. táblázat. Executable UML modellelemek

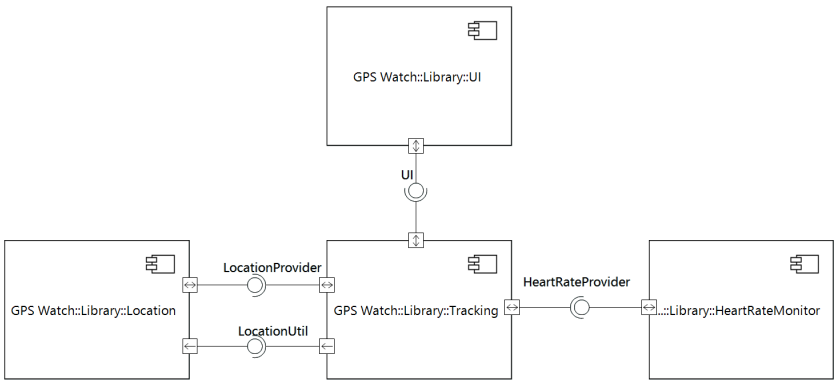
2.3.5. Mérnöki modellek tervezése BridgePoint-ban

A *Mentor Graphics*® által fejlesztett BridgePoint egy mérnöki modellek tervezésére, szimulációjára, a modellekből forráskód automatikus származtatására használt alkalmazás, melyet többek között az Ericsson telekommunikációs vállalat is használ fejlesztései során.

A szoftverrel a modellvezérelt szoftver- és rendszerfejlesztés metodikája alapján lehet komplex szoftverek és rendszerek szakterületspecifikus mérnöki modelljét elkészíteni [16].

A rendszert felépítő egyes komponensek külön megtervezhetőek és interfészeken keresztül egymáshoz kapcsolhatóak, ezáltal megvalósítva a komponensek felelősségének szeparációját a 2.3 ábrán látható módon. A négy komponensből (*Location, Tracking, UI, HeartRateMonitor*) álló rendszer egyes komponensei különböző (pl. a *Tracking* és a *Location* komponensek a *LocationProvider* és a *LocationUtil*), definiált interfészeken keresztül kommunikálnak egymással, ezáltal az egyes komponensek felelősségének egymástól való szeparációja megvalósítható.

A komponenseken belül UML osztálydiagramszerű struktúrákkal lehet leírni az adott komponens belső struktúráját. Az osztálydiagramon ábrázolt egyes osztályok viselkedése állapotgépekkel írható le.



2.3. ábra. Komponens diagram példa

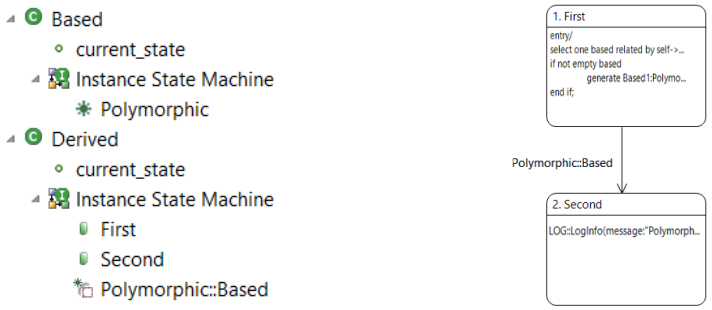
Az egyes osztályokhoz legfeljebb egy osztályszintű állapotgép és legfeljebb egy példányszintű állapotgép tartozhat. A példányszintű állapotgép az osztály példányosításával létrejövő „objektumok” viselkedését, míg az osztályszintű állapotgép az összes példányra kiterjedő, közös viselkedést definiálja.

Az egyes állapotgépek állapotokból és események hatására bekövetkező állapotátmenetekből épülnek fel. Az egyes állapotátmenetekhez és állapotokhoz tartozó viselkedések a BridgePoint saját viselkedésleíró nyelve, az OAL (*Object Action Language*) segítségével adhatók meg [17].

BridgePointban lehetőség van a komponensen belüli osztályok között specializációs viszonyt definiálni. Ezáltal lehetőség van a polimorf eseménykezelés bevezetésére, mellyel csökkenthető a komponensen belül az osztályok egymástól való függősége.

Polimorf eseménynek nevezünk egy leszármaztatási hierarchiában egy olyan eseményt, amely az ősosztály állapotgépében nem, de a leszármazott osztály állapotgépében legalább egy állapotátmenethez kötve van a 2.4 ábrán látható módon.

Az ábrán a *Based* ősosztály és a belőle származtatott *Derived* leszármazott osztály látható. A *Based* osztály rendelkezik egy *Polymorphic* eseménnyel, amely a leszármazott osztály két állapotú állapotgépében a *First* állapotból a *Second* állapotba vezető állapotátmenetre van őrfeltételként helyezve. Ezáltal, ha a *Based* osztály egy példányának küldenek egy *Polymorphic* eseményt, az a *Derived* osztály példányszintű állapotgépében lesz kezelve.



2.4. ábra. Polimorf eseménykezelés példa

Az egyes komponenseket lehetőség van interfészeken keresztül olyan komponensekhez csatolni, melyeket nem lehet modellszinten megalkotni, hanem közvetlenül platformspecifikus forráskó-

dot tartalmaznak. Így megvalósítható, hogy a közvetlenül nem modellezhető részek az egyes komponensek modelljeiből is elérhetőek legyenek, ezáltal hozzájárulva az integrálhatósághoz.

2.3.6. Iparban elterjedt modell alapú fejlesztőeszközök

A *Mentor Graphics*® által fejlesztett BridgePoint-on kívül számos modell alapú fejlesztőkörnyezet létezik, melyek közül néhányat ismertetünk ebben a szakaszban.

A *MathWorks*® által fejlesztett Simulink® és Stateflow®, valamint az *Esterel Technologies*® által fejlesztett SCADE Suite® nevű keretrendszerek többek között komponensmodellek belső működésének állapotgép-alapú leírására és verifikációjára széleskörben használt tervezőeszközök. A BridgePoint által követett xtUML-hez képest a használt belső modellezési nyelv nem szabványos, és a zárt működés következtében formális szemantikájuk sem ismert. Azonban fontos megjegyezni, hogy ezen zárt eszközöknek a kódgenerátoruk a legmagasabb szinten hitelesített, ezért biztonságkritikus rendszerek tervezésére közvetlenül alkalmazhatóak.

Az *IBM*® által fejlesztett Rational® Software Architect, a BridgePoint-hoz hasonló tervezőeszköz, melynek belső nyelve (UMLrt) szorosabban kapcsolódik az OMG által definiált szabványokhoz. A BridgePoint-hoz képest előnyként említhető meg, hogy hierarchikus állapotgépek tervezésére alkalmas. Azonban a BridgePoint-tól eltérően, ennél a szoftvernél csak platformspecifikus C++ kód alkalmazható akcionyelvként. Ezzel szemben a BridgePoint-ban OAL kódot alkalmaznak, amely egy magasabb szintű nyelv. Ezáltal lehetőség nyílik platformfüggetlen modellek viselkedésének definiálására, melyből platformspecifikus forráskód generálható.

Végül, de nem utolsó sorban az *Abstract Solutions*® keretrendszere alapvetően a Kennedy Carter által megvalósított iUML nevű keretrendszeren alapul, amely követi az MDA (*Model Driven Architecture* [18]) és Executable UML metodikát, azonban a BridgePoint-hoz képest kevés, nyilvánosan elérhető dokumentációval rendelkezik, ezért nem ezt a megoldást választottuk.

2.4. Verifikáció

Munkánk során kiemelt figyelmet fordítottunk, hogy olyan tervezési eljárásokat és módszereket alkalmazzunk, amelyek biztosítják, hogy az elkészült rendszer helyesen működjön. Kódgenerátorok alkalmazásával automatikusan származtatjuk a szoftver implementációt a modellből, ezzel elkerülve a kódolási hibákat.

Azonban a tervezési hibák kiszűrésére új módszereket kellett alkalmaznunk: formális verifikációt használtunk. A formális verifikáció olyan módszer, amely a rendszer tervek formális modelljein képes bizonyítani azok helyességét, vagy megtalálni az esetleges tervezési hibákat. Munkánk során állapotgép alapú modellezési megközelítést alkalmaztunk, ezért a lehetséges formális megközelítések közül a modellellenőrzésre esett a választásunk. A modellellenőrzés során azt vizsgáljuk, hogy egy adott formális modell megfelel-e a vele szemben támasztott formális követelményeknek.

Ebben a fejezetben bemutatjuk az általunk használt formális modellező nyelvet, a temporális logikai specifikációs nyelvet, továbbá az ezek ellenőrzésére szolgáló, az iparban is elterjedten használt modellellenőrző keretrendszert.

2.4.1. Véges automata formalizmus

Véges állapotterű rendszerek modellezése gyakran véges állapotú automatákkal történik. Az automaták megalkotásával a rendszer viselkedését lehet modellezni, szimulálni valamint ellenőrizni. A véges automatáknak a matematikában használt formális definíciója a következő:

3. Definíció (Véges automata). *Egy véges automata $\mathcal{A}$ egy olyan $\langle N, l_0, \Sigma, \delta, F \rangle$ ötös, ahol*

- $N \neq \emptyset$ *az automata állapotainak halmaza,*
- $l_0 \in N$ *kezdőállapot,*
- $\Sigma \neq \emptyset$ *az automata ábécéje,*
- $\delta : N \times \Sigma \to N$ *az automata állapotátmeneti függvénye,*
- $F \subseteq N$ *az elfogadó állapotok halmaza.*

A modellezésben és általában a számítástechnikában legtöbbször a fenti véges automatáknak egy speciális változatát használjuk, éspedig az eseményvezérelt véges automatákat. Ez annyiban módosítja a fenti definíciónkat, hogy az automata ábécéje Σ az automatában előfordulható események összessége, beleértve az üres eseményt, illetve a definíció a későbbi egyszerűség kedvéért kibővíthető az állapotátmeneti függvény által meghatározott átmenetek halmazával, a következőképpen:

- $E \subseteq N \times \Sigma \times N$ élek (állapotátmenetek) véges halmaza.

Az $l \xrightarrow{g,a,r} l'$ jelölés akkor használható, ha $\langle l, g, a, r, l' \rangle \in E$, azaz vezet l állapotból átmenet l' állapotba, amelyhez g őrfeltétel, a akció és r órahalmaz van rendelve. Az r órahalmaz azokat az órákat tartalmazza, amelyeket az állapotátmenet során le kell nullázni.

2.4.2. Automaták hálózata

Elosztott, időzített rendszerek esetén több időzített automata hálózata jelentheti a megoldást a modellezésben. Az időzített automaták hálózata tulajdonképpen az $\mathcal{A}_1 \ldots \mathcal{A}_n$ automaták párhuzamos kompozícióját jelenti. Az automaták között szinkronizációs akciók segítségével szinkron kommunikáció hozható létre (a? fogadó és a! küldő szinkronizáció), míg globális változók segítségével aszinkron kommunikáció is megvalósítható. A szinkronizáció pedig előre definiált csatornákon keresztül történhet.

2.4.3. Az UPPAAL modellellenőrző eszköz

Az UPPAAL egy valósidejű rendszerek modellezésére, szimulációjára és verifikációjára használt keretrendszer, melyet az Uppsala-i és az Aalborg-i egyetemek közösen fejlesztenek. Első változata 1995-ben jelent meg, jelenleg a 4.0-s verzió a legfrissebb.

Az UPPAAL képes automaták hálózatát is leírni, ahol az automata jelentése a fent definiáltaknak felel meg. Ezen felül pedig további bővítéseket is illeszt az automata formalizmushoz, ezeket a továbbiakban mutatjuk be [19].

2.4.4. Véges automaták UPPAAL-ban

Az UPPAAL-ban definiálható véges automaták felépítése hasonló az UML állapotgépekéhez. A két alapeleme egy automatának a következő:

- *állapotok*: névvel vannak ellátva és egy adott időpillanatban csak egy lehet aktív
- *állapotátmentek* (tranzíciók): a rendszer lehetséges állapotváltozásait írják le, *őrfeltételeket* és *akciókat* lehet hozzájuk rendelni.

UPPAAL-ban a fent említett alapelemekhez további tulajdonságokat rendelhetünk. Ilyenek például *helyinvariánsok* definiálása, *szinkronizációk* megadása az átmenetekhez vagy *óraváltozók* definiálása.

2.4.5. Időzítések UPPAAL-ban

Az UPPAAL támogatja a valósidejű rendszerek modellezését és ellenőrzését is. Mindezt az automaták valósidejű *óraváltozókkal* való kiterjesztésével teszi lehetővé, melyeket a többi változóhoz hasonlóan lehet definiálni, valamint különböző kifejezésekben használni. Az óraváltozó egy logikai órát jelent, aminek beállításával és olvasásával időfüggő viselkedést is képesek vagyunk modellezni a rendszerünkben. Általában ezen óraváltozók pontos értékét nem ismerjük, viszont össze tudjuk őket hasonlítani konstansokkal, ezáltal óra intervallumokat kapva. Az óraváltozóknak két fontos tulajdonságuk van:

- Az óraváltozók értéke vagy monoton növekszik vagy nullázódik, ha ezt explicit módon megtesszük.
- Egy rendszer egy állapotban tetszőleges (véges) ideig maradhat, vagy azonnal továbbléphet (0 időegységig marad az állapotban). Ez utóbbi esetben állapotinvariánsok, őrfeltételek és úgynevezett speciális állapotok (Urgent, Committed) definiálásával szabályozhatjuk az állapotokban eltöltött időt.

Mivel a jelenlegi modellünk időzítéseket nem tartalmaz, ezért a dolgozatban ezzel a területtel nem foglalkozunk, azonban a 3.3 fejezetben bemutatott leképezés segítségével időzített analízisre is lehetőségünk lenne.

2.4.6. Szinkronizáció UPPAAL-ban

Elosztott, időzített rendszerek működését UPPAAL-ban az előzőekben említett időzített automaták hálózatával tudjuk modellezni. Mivel legtöbbször a különböző komponensek közötti helyes együttműködést kell vizsgálni, a kommunikáció modellezésére UPPAAL-ban bevezették a szinkronizációs operátorokat. Az UPPAAL kétféle szinkronizációt támogat: *egyszerű*, illetve *broadcast* jellegű szinkronizációt. Mindkét esetben szükség van úgynevezett szinkronizációs csatornák definiálására, ugyanis ezeken keresztül történik az üzenetváltás. A csatorna definiálása után az élekhez (állapotátmenetekhez) kétféle szinkronizációs akciót tudunk csatolni: az `a` csatornán küldés az `a!` operátorral, fogadás pedig `a?` operátorral történik.

Egyszerű szinkronizáció akkor és csak akkor valósul meg $\mathcal{A}$ és $\mathcal{B}$ automata között, ha a küldő $\mathcal{A}$ automata egy olyan állapotban tartózkodik, ahonnan a `a!`-t tartalmazó él engedélyezve van és a fogadó $\mathcal{B}$ automata pedig egy olyan állapotban van, ahol legalább egy `a?`-t tartalmazó él engedélyezve van. Ha több fogadó létezik, akkor a szinkronizáció csak az egyikkel (véletlenszerűen kiválasztva) fog megvalósulni.

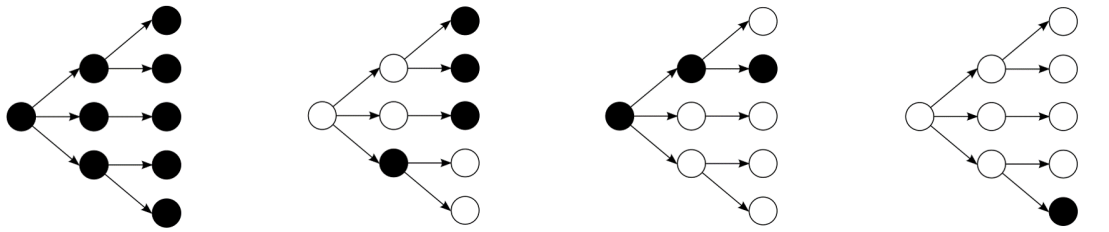
Broadcast szinkronizáció egy küldő $\mathcal{A}$ és (0..*) $\mathcal{B}_i$ fogadó automata között valósulhat meg. A fogadó fél viselkedése megegyezik az egyszerű szinkronizációban leírtakkal, viszont a küldő fél az összes $\mathcal{B}_i$ fogadó automatának elküldi a szinkronizációs üzenetet, még akkor is ha ezek száma 0.

2.4.7. Követelmények specifikációja UPPAAL-ban

A modellellenőrző eszközök fő célja a követelmények ellenőrzése a modellen. Ahogy a modellek, úgy a követelmények specifikációja is egy formálisan jól definiált nyelvvel vannak kifejezve, ami nem más mint a temporális logika. Az UPPAAL a CTL elágazó idejű temporális logika egy részhalmazát használja a követelmények megfogalmazására. Egy UPPAAL követelmény tulajdonképpen egy útvonal kvantorból és az útvonalon az állapotokra vonatkozó operátorok kombinációjából áll. Ezeket a követelményeket három csoportba sorolhatjuk:

- elérhetőségi követelmények
- biztonsági követelmények
- élőségi követelmények

a 2.5 ábrán láthatók az UPPAAL által elfogadott CTL kifejezések. A kitöltött körök azokat az állapotokat jelképezik, amelyekre teljesül a ϕ állapotkifejezés.



(a) `A[]`ϕ (b) `A<>`ϕ (c) `E[]`ϕ (d) `E<>`ϕ

2.5. ábra. UPPAAL-ban használt CTL operátorok

2.4.8. Állapotkifejezések

Egy állapotkifejezés egy olyan kifejezés, mely kiértékelése az adott állapoton történik. A szintaxisa pedig az őrfeltételekéhez hasonló, azaz a kifejezések mellékhatásmentesek, de a diszjunkció használata nem tiltott. Ugyanakkor lehetőség van leellenőrizni, hogy egy folyamat (automata) egy adott állapotban tartózkodik-e vagy sem *P.loc* típusú kifejezések segítségével, ahol *P* a folyamat és *loc* az adott állapot. Az UPPAAL a holtpont ellenőrzésére egy speciális állapotkifejezést használ, ami a `deadlock` kulcsszóból áll és ez mindig teljesül, ha egy állapot holtpontban van.

Például, ha azt akarjuk leellenőrizni, hogy a modellünk holtpontmentes, akkor az `A[] not deadlock` kifejezést kell megadni az eszköz verifikációs moduljának.

2.4.9. Elérhetőségi követelmények

Az elérhetőségi követelmények alatt azt kérdezzük a modellellenőrzőtől, hogy egy adott ϕ kifejezés teljesül-e a jövőben valamely állapot elérésekor. Az ilyen követelményeket CTL-ben a $\texttt{EF}\phi$ kifejezéssel, míg UPPAAL-ban a $\texttt{E<>}\phi$ lehet megfogalmazni.

2.4.10. Biztonsági követelmények

A biztonsági követelmények (vagy más néven: invariáns követelmények) a veszélyes helyzetek elkerülését írják elő, azaz minden időpillanatban és állapotban a kifejezésnek teljesülnie kell. Ilyen követelmény például a fent említett holtpontmentesség, de a kölcsönös kizárás és az adatbiztonságot is fel lehet hozni példának. Ha ϕ egy állapotkifejezés, akkor azt mondjuk, hogy ϕ-nek minden elérhető állapotban igaznak kell lennie. Az ezt kifejező formula CTL-ben $\texttt{AG}\phi$, míg UPPAAL-ban $\texttt{A[]}\phi$.

2.4.11. Élőségi követelmények

Az élő jellegű követelmények kívánatos helyzetek elérését írják elő, azaz azt, hogy valamikor a jövőben az adott kifejezés igaz lesz. Az ennek megfelelő CTL kifejezés $\texttt{AF}\phi$. A legegyszerűbb esetben UPPAAL-ban az élőségi kifejezéseket a $\texttt{A<>}\phi$ kifejezéssel fogalmazhatjuk meg. Emellett az UPPAAL definiálhatunk egy másik élőségi követelményt is, az úgynevezett „leads to $\rightarrow$” operátorral. Például a $\phi \rightarrow \xi$ azt jelenti, hogy amikor ϕ teljesül, akkor előbb-utóbb ξ-nek is teljesülnie kell.

2.4.12. UPPAAL használata ipari környezetben

Az UPPAAL-t, mint véges automaták segítségével formális verifikációra alkalmas eszközt, számos ipari esettanulmányban, protokollok helyességének ellenőrzésére használták sikeresen az elmúlt évek során. Ezek közül csak néhányat megemlítve:

- Egy olyan kommunikációs protokoll helyességét ellenőrizték UPPAAL-lal, amely protokoll biztosítja vivőérzékeléses többszörös hozzáférésű környezetben a szállított keretek ütközésének elkerülését és felső korlátot ad a hálózati csomópontok közti kommunikációs késleltetésre [20].
- Philips által kifejlesztett protokollt is verifikáltak UPPAAL-lal, amely protokoll audió komponensek közti vezérlési információk cseréjét végzi, Manchester-kódolással. A verifikáció során bizonyították a Manchester-kódoláshoz szükséges időzítési kritériumok teljesülését a protokollban [21].

A UPPAAL-t az ipari elterjedtsége, relevanciája, valamint a véges automata, és a 3.3.1 szakaszban bemutatandó állapotgép ekvivalenciája miatt választottuk TDK dolgozatunkban formális modellellenőrző keretrendszernek.

3. fejezet

Modell alapú szoftverfejlesztés támogatása formális ellenőrzésekkel

Munkánk egyik célja egy olyan keretrendszer fejlesztése volt, amely támogatja a mérnöki modellek formális analízisét. Ehhez modelltranszformációkat fejlesztettünk, amelyek megvalósítják a mérnöki modellek formális modellekké való leképezését. Munkánk során egy nyílt forráskódú modellező nyelvet és egy kereskedelmi modellező eszközt használtunk.

Első megközelítésben a nyílt-forráskódú lehetőségekhez kapcsolódva az UML állapotgépek viselkedésének leírására definiált Alf nyelvet [14] felhasználva készítettünk el egy Alf modellből UPPAAL automatát előállító modelltranszformációt [22].

Második megközelítésben egy ipari környezetben széleskörben elterjedt, zárt forráskódú szoftverrel megtervezett mérnöki modellek verifikációjára alkalmas megoldást készítettünk el.

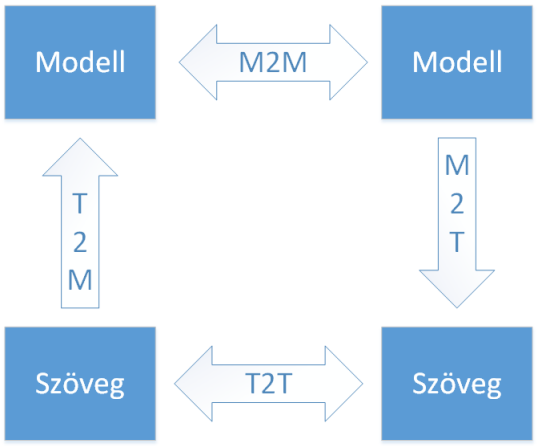
3.1. Modellek transzformációja

Annak érdekében, hogy a magasszintű mérnöki modellek helyességét vizsgálni tudjuk, szükségünk van formális modellek származtatására. A származtatás alatt rendszerint valamilyen transzformációt értünk, ami lehet manuális, részben automatizált vagy teljesen automatizált.

4. Definíció (Modelltranszformáció). *A modelltranszformáció egy olyan folyamat, mely során előre definiált transzformációs szabályok alapján egy forrásmodellből automatikusan generálható egy célmodell. A transzformációs szabályok együttese leírja, hogyan lehet egy adott modell forrásnyelvét a célmodell nyelvére lefordítani. Egy transzformációs szabály pedig nem más, mint a két nyelvbeli (forrás- és célnyelv) egy vagy több konstrukció közötti megfeleltetés [23].*

A kiinduló- és a célformátum alapján négy csoportba sorolhatjuk a modelltranszformációkat, a 3.1 ábrán látható módon: modell-modell (M2M), modell-szöveg (M2T), szöveg-szöveg (T2T), illetve szöveg-modell (T2M) közötti transzformáció.

Modell-modell transzformáció esetén a kiinduló- és a célformátumot a modell alapú tervezőeszközök számára értelmezhető modelleknek tekintjük. Modell-szöveg transzformációnál a kiindulási modellnek tekintett formátumból egy szöveges reprezentáció készül, ami a nem modell

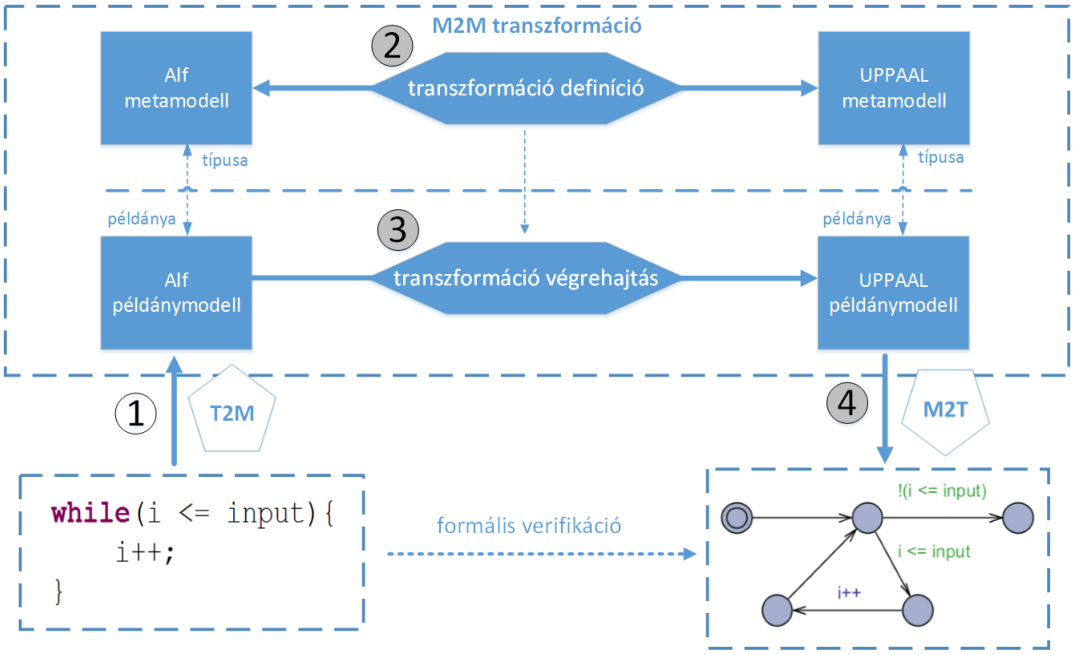


3.1. ábra. Transzformáció típusok

alapú tervezőeszközök számára értelmezhető lesz. Ezáltal megvalósítható a modell alapú és a nem modell alapú tervezőeszközök közti átjárás és kódgenerálás.

3.2. Alf modell leképzése UPPAAL automatára

Alf példánymodellek formális modellekre transzformálásának négy fő fázisa a 3.2 ábrán látható: az első egy szöveg-modell (T2M) transzformáció, a második a transzformáció definíciója a metamodellekre nézve, a harmadik egy modell-modell (M2M), míg a negyedik egy modell-szöveg (M2T) transzformáció.



3.2. ábra. Alf - UPPAAL transzformációs folyamat

Az első fázisban (1), a T2M transzformáció során, a szöveges reprezentációjú Alf forráskódot egy elemző (*parser*) beolvassa és elkészíti belőle a vele ekvivalens Alf példánymodellt. Ennek

a fázisnak a végrehajtására a *Papyrus UML* által készített elemzőt és példánymodell-készítő eszközt használtuk.

A transzformáció modell alapú végrehajtásához szükséges a nem modell alapú UPPAAL szabadon hozzáférhető metamodellje[1], amely tartalmazza az UPPAAL által használt strukturális elemeket.

A második fázisban (2), a Papyrus UML által készített, az OMG-s platformspecifikus implementációhoz képest korlátos kifejezőerejű Alf metamodell és a szabadon hozzáférhető UPPAAL metamodell alapján definiáltuk a leképzési szabályokat a metamodellekre nézve:

- Változó deklarációt és definíciót az automatához tartozó lokális változó deklarációra és definícióra.
- Változó értékének megváltozását az automata helyek közti átmenethez tartozó értékmódosító utasításra.
- Feltételes elágazó utasítást és ciklust a predikátum alapján elágazó helyekre, a predikátumot a helyeket összekötő élek őrfeltételeként értelmezve.
- Feltételes elágazó utasítás és ciklus magját az automatában helyek sorozatára, a helyeket összekötő élekhez tartozó őrfeltételekre és értékváltoztató utasításokra.
- Minden, az UPPAAL automatában lévő hely commitált típusú, ami megegyezik a kiindulási forráskódban lévő utasítások atomi végrehajtásának szemantikájával.

A harmadik fázisban (3), a M2M transzformáció során a példánymodellként beolvasott Alf forráskódot szisztematikusan bejárva, az egyes Alf elemeket az előfordulások sorrendje alapján a cél UPPAAL példánymodellben a nekik megfeleltethető metamodell-elemek példányaira iteratívan képeztük le. Ezáltal az egyes Alf utasítások sorrendileg és szemantikailag helyesen transzformálódtak a velük ekvivalens UPPAAL példánymodellbe.

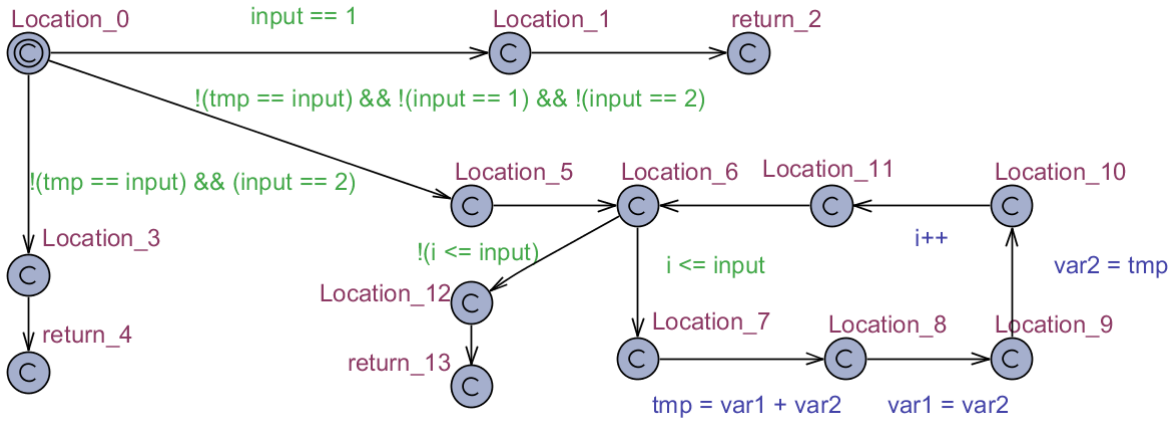
Az utolsó, negyedik fázisban (4), a M2T transzformáció során a M2M transzformáció eredményeként létrejött példánymodell leírását átalakítjuk az UPPAAL által értelmezhető formátumra, így a keletkező automatán formális verifikáció hajtható végre az UPPAAL-ban. Ezáltal a korlátos kifejezőerővel rendelkező Alf metamodellhez tartozó példánymodellek szekvenciális részének formális helyessége bizonyítható.

TDK dolgozatunkban a teljes transzformációs láncnak a 2-4. fázisait valósítottuk meg. Az Alf metamodell és a transzformációs folyamat kibővítésével az eddig támogatott Alf elemek halmaza bővíthető, ezáltal pontosabb verifikációs eredmények kaphatók.

3.2.1. Példa: Fibonacci-sorozat formális verifikációja

A transzformáció szemléltetésére példaként a Fibonacci-sorozat k-adik elemét kiszámító algoritmus Alf megvalósítását mutatjuk be. A kiindulási Alf forráskódot modellként értelmezve transzformáltuk a 3.3 ábrán látható UPPAAL automatára, melyen verifikáció hajtható végre. Az

[1] https://svn-serv.cs.upb.de/muml_verification/trunk/de.uni_paderborn.uppaal/



3.3. ábra. Példa: Fibonacci-sorozatot megvalósító UPPAAL automata

algoritmus Alf implementációja terjedelmi okok miatt a Függelékek között, az F.1.1 fejezetben található meg.

A transzformáció során a kód elején lévő feltételes elágazás az UPPAAL automatában egy helyből (*Location_0*) kivezető három élre és hozzájuk tartozóan három helyre képződött le. Az elágazás két, automatikus visszatérést tartalmazó ága (*Location_1*, *Location_3*) végén lévő helyek (*return_2*, illetve *return_4*) tartalmazzák az algoritmus visszatérési értékét, a „triviális" esetekben.

Ha a feltételes kifejezés egyik ága sem teljesül, akkor a *Location_5* helyen folytatódik a végrehajtás, ahonnan a kódban lévő ciklus miatt egy újabb elágazás következik a ciklus feltétele szerint. Ha a ciklusban maradás feltétele ($i <= input$) teljesül, akkor az automatában egy, a *Location_6*-ból induló, majd ugyanoda visszatérő láncon marad a vezérlés.

Ha a ciklusban maradás feltétele nem teljesül, akkor a *return_13* hely tartalmazza a ciklus végén a keresett változó (*input*) értékét.

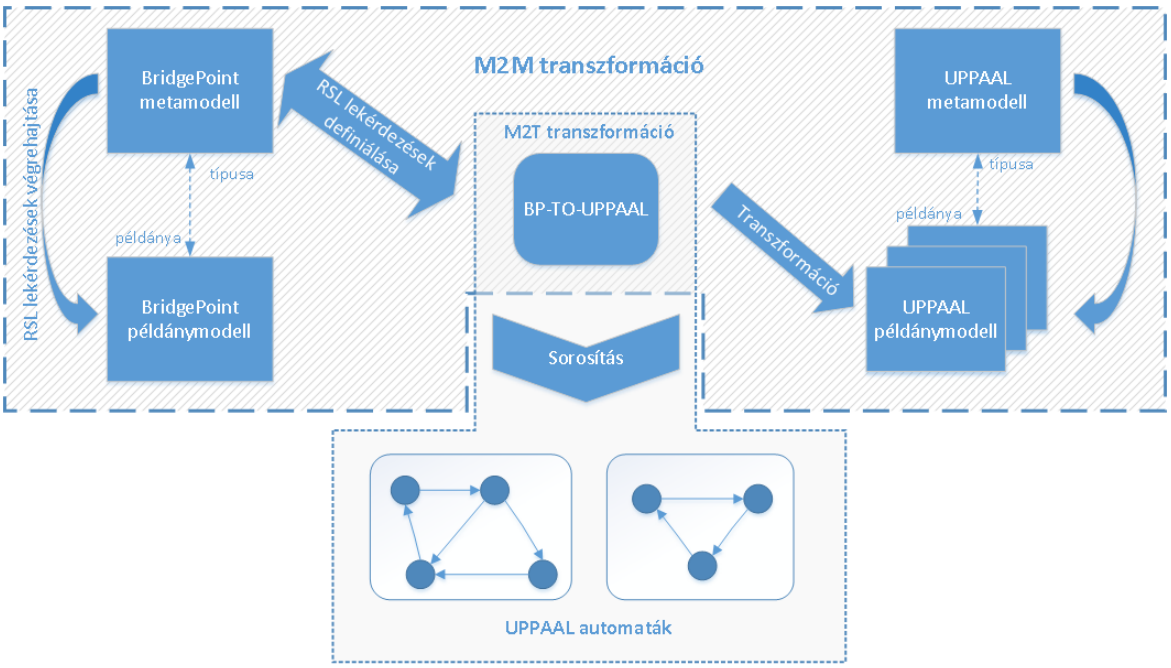
3.3. Mérnöki modellek formális verifikációja

A BridgePoint-ban elkészített mérnöki modellek transzformálása - az Alf modellekhez hasonlóan - formális modellekre két fő fázisból áll: az első egy modell-modell (M2M) transzformáció, míg a második egy modell-szöveg (M2T) transzformáció. A teljes folyamatot a 3.4 ábra szemlélteti. A formális modellbe való átalakításnak a lényegi része az első fázisban történik.

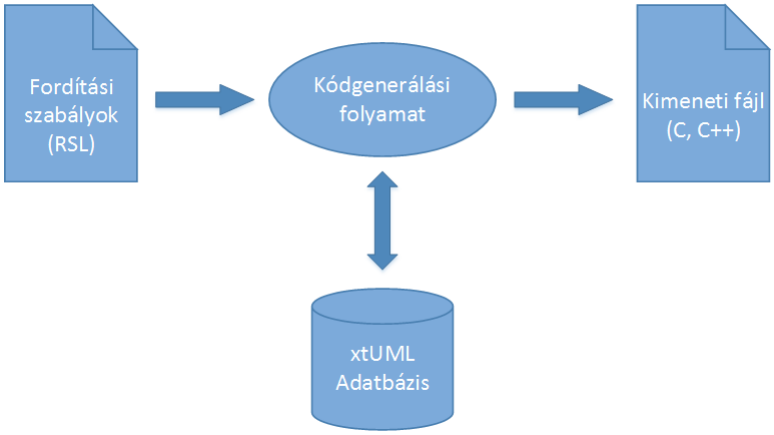
A BridgePoint-ban elkészített modellünkből a fordító forráskódot generál, amihez két bemeneti adatra van szüksége: az xtUML adatbázisra, ami tartalmazza a modell adatokat, valamint a fordítási szabályok halmazára, amit a BridgePoint saját, RSL (*Rule Specification Language*) nevű belső nyelvén kell megfogalmazni. Ezen fordítási szabályok egy specifikációt alkotnak, ami alapján egy vagy több szöveges fájl keletkezik. A 3.5 ábra ezt a folyamatot szemlélteti.

Az RSL-ben megfogalmazott szabályokkal meghatározhatjuk a generált fájlok struktúráját. A nyelv szintaxisa a következő elemekből áll:

- szöveges literálok,
- vezérlőszerkezetek,



3.4. ábra. BridgePoint - UPPAAL transzformációs folyamat



3.5. ábra. BridgePoint kódgenerálási folyamata

- helyettesítő változók.

A fordító a szöveges literálokat változtatás nélkül a kimeneti fájlba írja, a vezérlőszerkezetek segítségével az xtUML adatbázison lehet lekérdezéseket és bejárásokat megfogalmazni, a helyettesítő változók segítségével pedig ugyancsak az xtUML adatbázisból nyerhetünk ki adatokat, és a kimeneti fájlba írás előtt formázhatjuk őket.

A lekérdezések szintaxisa nagyon hasonló az SQL nyelven írt lekérdezésekhez. Például a 3.1 kódrészlet kódrészlet első sora visszaadja a modellben szereplő összes objektumot, amit utána a vezérlés egy ciklussal végigjár és kiírja a megtalált objektumok nevét.

Az RSL lekérdezéseket a BridgePoint metamodellbeli elemeire kell definiálni, amik majd a példánymodellből felépített adatbázison hajtódnak végre. Az RSL kifejezések segítségével alapesetben batch-jellegű kódgenerálás hajtható végre, mert a kifejezőereje nem teszi lehetővé, hogy a kódgenerálási folyamatban inkrementalitást alkalmazzunk.

```
.select many objects from instances of O_OBJ    //vezérlőszerkezet
        .for each object in objects
                ...
        Név             // szöveges literál
        ${object.Name}  // helyettesítő változó,
                        // mely kiírja az objektum név attribútumát
                ...
        .end for
```

3.1. RSL példakód

Ezenkívül a BridgePoint metamodellje – a korábban említett Alf metamodellhez képest – nem kompatibilis az elterjedt, nyílt-forráskódú modellezési környezettel, az EMF-fel (*Eclipse Modeling Framework*) sem.

A problémára a megoldást az RSL-ben értelmezett **`.invoke SHELL_COMMAND()`** speciális utasítás jelenti, melynek segítségével az operációs rendszer parancssoros felhasználói felületén hajthatók végre utasítások. Ezáltal lehetőség adódik egy olyan külső alkalmazással való kommunikációra, amely képes parancssori utasítások feldolgozására.

Az általunk választott megoldás egy Eclipse alapú alkalmazás készítése volt, amely tartalmazza az UPPAAL EMF alapú metamodelljét és képes inkrementálisan felépíteni egy példánymodellt, majd azt a megfelelő XML formátumú UPPAAL-kompatibilis fájlba sorosítani. Ezáltal első fázisban egy M2M transzformációt hajtunk végre, melynek eredménye a teljes UPPAAL példánymodell, majd a második fázisban e példánymodellhez egy kódgenerátort csatolva szöveges fájlba sorosítunk, ami egy M2T transzformációnak felel meg.

A kommunikáció a BridgePoint és az Eclipse alkalmazás között egy általunk specifikált interfészen keresztül történik. Az általunk definiált parancsokat és argumentumaikat a 3.1 táblázat foglalja össze.

A modelltranszformáció során az alábbi döntéseket hoztuk:

- a váltó három ágához egy-egy azonosítót rendelünk az alábbiak szerint:
  - kitérő ág: DIV = 0
  - egyenes ág: STR = 1
  - váltócsúcs: TOP = 2
- minden UPPAAL automata egyedi azonosítóval rendelkezik, ezáltal lehetőség van célzott szinkronizációk küldésére.
- a BridgePoint állapotokba vagy átmenetekre írt üzenettípusok három fő csoportba oszthatók:
  - eseménykuldés
  - függvényhívás (üzenettovábbítás a megfelelő csatornán)
  - interfészen keresztül történő hívás, melynek fajtái:
    - szignál küldés
    - függvényhívás

Parancs	**Argumentum**	**Leírás**
NTA	ntaName	gyökérelem létrehozása
GLOBALDECLARATION	expression	deklaráció létrehozása
TEMPLATE	templateName	automata létrehozása
LOCATION	templateName locationName isCommitted	hely létrehozása és típusának beállítása
LOCATIONTYPECHANGE	templateName locationName type	létrehozott hely típusának megváltoztatása
REARRANGEMODEL	templateName	élek és helyek rendezése (lásd 5. szabály)
INITIAL	templateName locationName	kezdőállapot beállítása
TRANSITION	templateName sourceLocation targetLocation synchronization snycTarget guardExpression updateExpression	él létrehozása a megadott paraméterekkel
COMPLETEMODEL	templateName	lásd 6. szabály
GENERATE	outputFile	UPPAAL fájl generálás

3.1. táblázat. BP-TO-UPPAAL alkalmazás parancsai argumentumokkal

A továbbiakban bevezetjük a BridgePoint állapotgépek és az UPPAAL automaták közti transzformációs szabályok definiálásában szerepelő jelöléseket:

- Legyen N a BridgePoint-beli állapotgép állapotainak halmaza.
- Legyen E a BridgePoint-beli állapotgép éleinek halmaza.
- $OAL\{x\}$ operátor, ahol $x \in N$ vagy $x \in E$ és az $OAL\{x\}$ jelentse az x-ben található OAL üzenethívások számát. Továbbá az $OAL_i\{x\}$ jelentse az x-ben található i-ik OAL üzenetet.
- Legyen $e_0 \in E$ olyan él, mely nem tartalmaz OAL kódot.
- Legyen L az UPPAAL automata helyeinek halmaza.
- Legyen T az UPPAAL automata tranzícióinak halmaza.
- Legyen $C \subseteq L$ az UPPAAL automata commitált helyeinek halmaza.

1. szabály (Üres állapot). *Egy BridgePoint-beli üres állapot UPPAAL-ban egy hely.*

$$\begin{array}{c}\underline{\text{BridgePoint}}\\ A \in N\\ OAL\{A\} = 0\end{array} \quad \Rightarrow \quad \begin{array}{c}\underline{\text{UPPAAL}}\\ A \in L\end{array}$$

2. szabály (Nem üres állapot). *Egy BridgePoint-beli OAL kifejezéseket tartalmazó állapot leképzése UPPAAL-beli helyekre az alábbi módon történik: az eredeti állapotból két hely lesz (kezdő és végső) és közötte pedig annyi hely, ahány OAL kifejezés van az eredeti állapotban majd ezen helyeket az OAL kifejézésekből képzett tranzíciók kötik össze. A kezdő és a köztes állapotok típusa commitált lesz, mert BridgePoint-ban egy állapotban lévő OAL utasítássorozat az állapotba lépéskor, megszakítás nélkül lefut (run-to-completion).*

$$\begin{array}{c}\underline{\text{BridgePoint}}\\ \\ A \in N\\ OAL\{A\} > 0\\ \\ \text{event}\end{array} \quad \Rightarrow \quad \begin{array}{c}\underline{\text{UPPAAL}}\\ \exists\, A_{kezdő}, q_1, q_2, \ldots, q_{OAL\{A\}}, A_{végső} \in L \,ahol\, A_{kezdő}, q_i \in C\\ \{A_{kezdő}, q_1\} = t_0 \in T\\ \{q_i, q_{i+1}\} = OAL_i\{A\}, 1 \leq i < OAL\{A\}\\ \{q_{OAL\{A\}}, A_{végső}\} = OAL_{last}\{A\}, last = OAL\{A\} \in T\\ \text{event!}\end{array}$$

3. szabály (Üres él). *Egy BridgePoint-beli üres él UPPAAL-ban egy tranzíció. Az élen szereplő triggerfeltételből a tranzíción egy fogadó szinkronizáció lesz.*

$$\begin{array}{c}\underline{\text{BridgePoint}}\\ A, B \in N\\ \exists\, \{A, B\} = e \in E\\ OAL\{e\} = 0\\ \text{trigger}\\ \text{event}\end{array} \quad \Rightarrow \quad \begin{array}{c}\underline{\text{UPPAAL}}\\ A, B \in L\\ \{A, B\} = t \in T\\ \text{trigger?}\\ \text{event!}\end{array}$$

4. szabály (Nem üres él). *Egy BridgePoint-beli OAL kifejezéseket tartalmazó él leképzése UPPAAL-ra a következőképpen történik: a küldő és fogadó állapotból egy-egy hely keletkezik az automatában. Köztük eggyel több köztes hely lesz az automatában, mint amennyi OAL kifejezés az adott élen volt. Az élen szereplő triggerfeltételből egy fogadó szinkronizációs tranzíció keletkezik a küldő és az első köztes hely között. A további köztes helyek között pedig az OAL kifejezéseknek megfelelő küldő szinkronizációval ellátott tranzíciók lesznek.*

$$\begin{array}{c}\underline{\text{BridgePoint}}\\ \\ A, B \in N\\ \exists\, \{A, B\} = e \in E\\ OAL\{e\} > 0\\ \\ \text{trigger}\\ \text{event}\end{array} \quad \Rightarrow \quad \begin{array}{c}\underline{\text{UPPAAL}}\\ \exists\, A, q_0, q_1, \ldots, q_{OAL\{e\}}, B \in L,\, ahol\, q_i \in C\\ \{A, q_0\} = t \in T\\ \{q_{i-1}, q_i\} = OAL_i\{e\}, 1 \leq i \leq OAL\{e\}\\ \{q_{OAL\{e\}}, B\} = t_0 \in T\\ \text{trigger?}\\ \text{event!}\end{array}$$

5. szabály (Élek átrendezése). *Ha egy A állapot esetén érvényesül a 2. szabály, akkor szükség van az A-ból kimenő és bejövő élek átrendezésére, ami a következő lépésekből áll:*

- $\forall X \in L - re$, *ha* $\exists \{X, A\} = t \in T$ *él* $\Rightarrow \{X, A_{kezdő}\} = t$
- $\forall X \in L - re$, *ha* $\exists \{A, X\} = t \in T$ *él* $\Rightarrow \{A_{végső}, X\} = \{A, X\}$ *és* $TÖRÖL(\{A, X\})$

Mivel a BridgePoint-ban létrehozott osztályok egyedi azonosítóval rendelkeznek, így az ezekhez tartozó állapotgépek számossága is legfeljebb egy lesz. Viszont, míg BridgePoint-ban futás közben hozhatunk létre új- és törölhetünk meglévő objektumokat (és így állapotgépeket), UPPAAL-ban verifikáció közben ezt nem tehetjük meg. Ezért UPPAAL-ban szükség van a végállapotok visszacsatolására a kezdőállapotba (6. szabály).

6. szabály (Végállapot visszacsatolása). *Ha a BridgePoint állapotgépben létezik olyan állapot, melyből nincsen kivezető él (azaz Final State-ben van), akkor UPPAAL-ban a neki megfelelő automatában ezt a helyet vissza kell csatolni a kezdőállapotba.*

A fenti szabályokat RSL kifejezésekkel megfogalmazva, majd az általunk elkészített alkalmazás (BP-TO-UPPAAL) segítségével sikerült automatikusan létrehozni az adott BridgePoint modellnek megfelelő UPPAAL példánymodellt, melyből generáltuk az UPPAAL által értelmezhető, automatákat tartalmazó bemeneti fájlt. Az automatákon formális kifejezéseket felírva (lásd 4.5.1 alfejezet) verifikálható azok működése, ezáltal a kiindulási mérnöki modellek működése is.

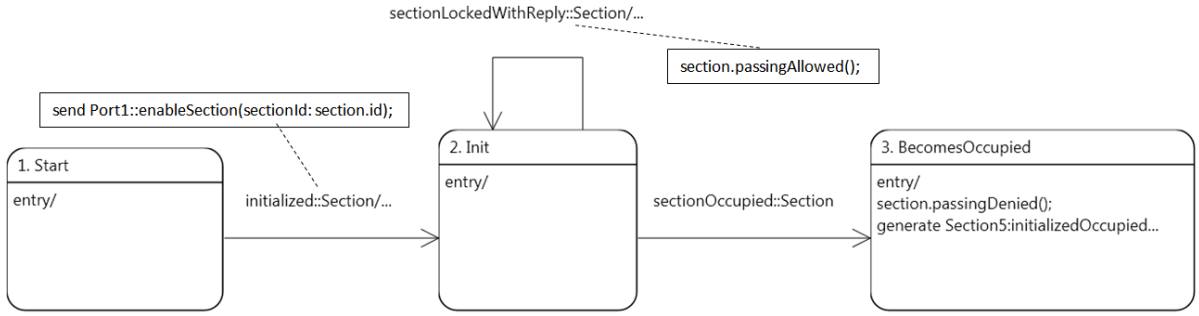
3.3.1. Példa: Szabad szakasz állapotgépének transzformációja automatára

A 3.6 ábra a fenti szabályok alkalmazását mutatja be a BridgePoint-ban elkészített *szabad* szakasz (*FreeSection*) állapotgépének transzformációjára. A szabad szakasz működése a következő: a *Start* kezdőállapotból egy *initialized* üzenet hatására az *Init* állapotba kerülünk, miközben a külső interfészen küldünk egy *enableSection* üzenetet. Ha a szakasz foglalt lesz, azaz *sectionOccupied* üzenet érkezik, akkor továbblépünk a *BecomesOccupied* végállapotba. Mivel ez az állapot tartalmaz OAL kifejezéseket, ezért amint aktívvá válik küld egy *passingDenied* üzenetet a váltónak, majd inicializálja a *foglalt* szakaszt (*OccupiedSection*) az *initializedOccupied* üzenet elküldésével. Továbbá láthatjuk, hogy az *Init* állapothoz tartozik egy hurokél is, mely szintén tartalmaz OAL kódot és a *sectionLockedWithReply* üzenet hatására triggerelődik.

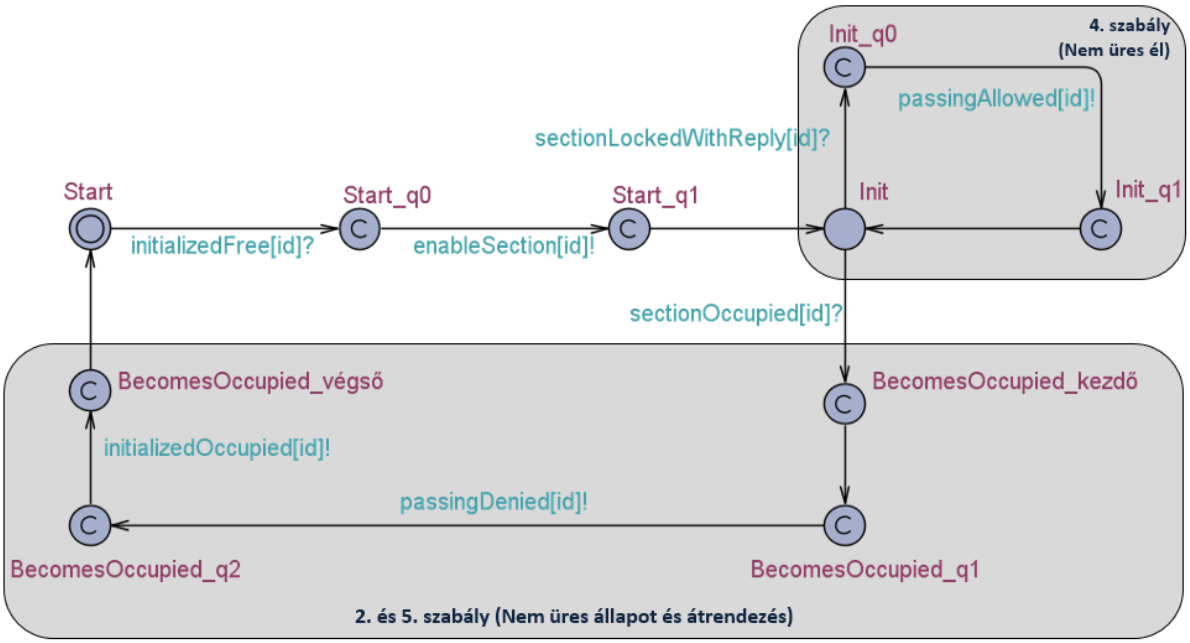
Az alsó ábrán láthatjuk a fent definiált szabályok alkalmazásával generált UPPAAL automatát. Az alkalmazott szabályok a következők: a *Start* és *Init* állapotok esetén az üres állapotra vonatkozó 1. szabály értelmében megfeleltetünk egy-egy helyet. Mivel a két állapot között menő él egy nem üres él, ezért itt a 4. szabályt kell alkalmazni, mely további két *commitált* helyet - az *Start_q0* és *Start_q1* - hoz létre a *Start* és *Init* helyek közé. Az *Init* állapotból egy hurokél indul, ami szintén nem üres, ezért újra alkalmazni kell a 4. szabályt, mely során újabb két hely - az *Init_q0* és *Init_q1* - keletkezik. Az utolsó átmenet pedig egy üres él (*sectionOccupied*) mentén történik, tehát a 3. szabályt kell alkalmazni. Továbbá, mivel a *BecomesOccupied* állapot nem üres, a 2. szabály értelmében ez kifejtésre kerül, azaz létrejön egy *BecomesOccupied_kezdő*, majd további két hely - *BecomesOccupied_q1* és *BecomesOccupied_q2* - az állapotban lévő két

OAL kifejezés elsütésére és egy *BecomesOccupied_végső* hely. Ebben az esetben szükség van az élek átrendezésére is (5. szabály), hiszen az eredeti, *BecomesOccupied* állapotból kimenő éleket a *BecomesOccupied_végső* helyből, míg a bejövő éleket a *BecomesOccupied_kezdő* helybe kell irányítani. Jelen példa esetén látszólag ez nem változtat a helyzeten, mivel a *BecomesOccupied* állapotból nem megy kifele él, viszont a 6. szabály alkalmazása során a visszacsatolást a *Start* állapotba a *BecomesOccupied_végső* állapotból kell megtenni.

Az ábrán a könnyebb beazonosítás érdekében megjelöltük a három összetettebb (2., 4. és 5.) szabály alkalmazásával megvalósult automata részeket. Ezen alapszabályok alkalmazása után a verifikáció optimalizálása érdekében végrehajtunk még egy átrendezést, melynek során az üres éleket összehúzzuk így csökkentve az állapotok számát.



(a) Állapotgép



(b) Automata

3.6. ábra. BridgePoint állapotgép transzformációja UPPAAL automatára

4. fejezet

Esettanulmány

A TDK dolgozat során elkészült biztonsági logika bemutatására egy olyan terepasztalt készítettünk, amelyen könnyen szemléltethető, ahogy a logika megakadályozza a síneken közlekedő mozdonyok ütközését.

Bemutatjuk a modellvasút-hálózat elrendezését (a 4.2 fejezetben), a biztosító berendezéshez használt hardvert (a 4.2.2 fejezetben), a BridgePoint állapotgépek csatlakozását a rendszerhez (a 4.3 fejezetben), a biztonsági logikát, amely megakadályozza a vonatok összeütközését (a 4.4 fejezetben), végül a biztonsági logikát felépítő egyik komponensre vonatkozó verifikációs eredményeinket (a 4.5 fejezetben).

4.1. Tervezés során követett módszertan

A TDK dolgozatban megválasztottuk azt a domaint (szakterületet), amellyel a biztonsági rendszer működése (és fontossága) jól demonstrálható. Mi a vasút területét választottuk erre a célra. A vasúton történő személyi szállítás során több száz ember életét kell biztosítani az előforduló veszélyektől, úgymint technikai problémáktól vagy esetleges emberi mulasztásoktól. A könnyebb szemléltetés miatt készítettünk egy modellvasút-hálózatot, amelyben a valóságban fellépő problémák némi eltéréssel könnyen bemutathatóak: a valós közlekedés során a vonatoknak kötelező haladási irányuk van egy-egy szakaszon, amely az általunk elkészített terepasztalon elektronikai okokból nem megvalósítható. Tehát a logikánk komplexitása növekszik a menetirány fogalom hiánya miatt (mindkét irányban ellenőriznünk kell a szabad utat).

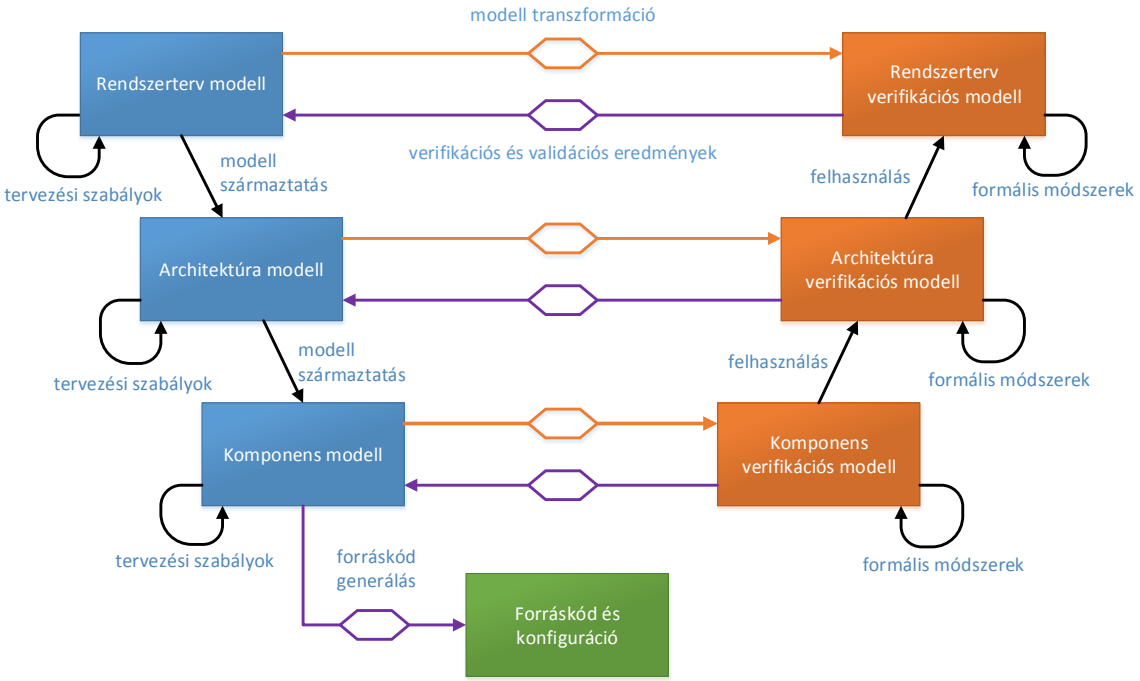
A tervezés első fázisában megismerkedtünk a modellvasút terminológiával és a terepasztaltervezés szabályaival és kényszereivel. A tervezéskor az egyik szempont az volt, hogy egy előző esettanulmányból származó elemeket, úgymint mozdonyokat, síneket is felhasználjuk, így a méretarány már adott volt. Az említett demonstrátor alkalmazás elektronikai szempontból egy régebbi megoldással, úgynevezett analóg vezérléssel rendelkezett, amely jelen esetben már nem szolgálta ki az összes követelményt: szükségünk volt a központi vezérelhetőségre, amely során a vonatokat és a pálya többi elemeit egységes módon, valamint egymástól függetlenül tudjuk irányítani. A mai modellezésben elterjedt digitális vezérlés pont ezeket az igényeket szolgálja ki, ennek megismerése és alkalmazása is feladat volt a hálózat elkészítése során.

Ehhez a digitális vezérlési rendszerhez kellett illesztenünk a saját rendszerünket, amely a vezérlés zártsága miatt egy teljesen különálló, független rendszerként viselkedik, és emiatt vezérlési hibák esetén is megakadályozza a veszélyes szituációkból kialakuló balesetek bekövetkezését. A biztosítóberendezés hardvere esetén az elosztott rendszerek tervezésekor általánosan alkalmazott dekompozíciós elvet alkalmaztuk: a biztonsági vezérléshez tartozó hardveres végrehajtó egységeket a váltók mentén, és azon belül a szerepkörök (felelősségek) mentén dekomponáltuk:

- *master*: a biztonsági logika által utasított egységek.
- *slave*: a master egységek alá tartozó eszközök, amelyek a fizikai beavatkozást végzik.
- *szakaszfoglaltság-gyűjtő*: a pályát folyamatosan monitorozó egység, amely a szakaszok foglaltságát vizsgálja és adja vissza.

A modellezéshez vásárolt alkatrészek csak a mozdonyok irányításában vesznek részt, a teljes biztonsági rendszer saját eszközökből épül fel.

A szoftvert a biztonságkritikus rendszerek tervezésénél általánosan használt Y-modell (a 4.1 ábra) mentén készítettük el. A korai szakaszban magasszintű követelményeket specifikáltunk, amelyek alapján a tervezéskor dekomponáltuk a rendszert kisebb komponensekre, majd ezeket a komponenseket egyenként terveztük meg és verifikáltuk őket.

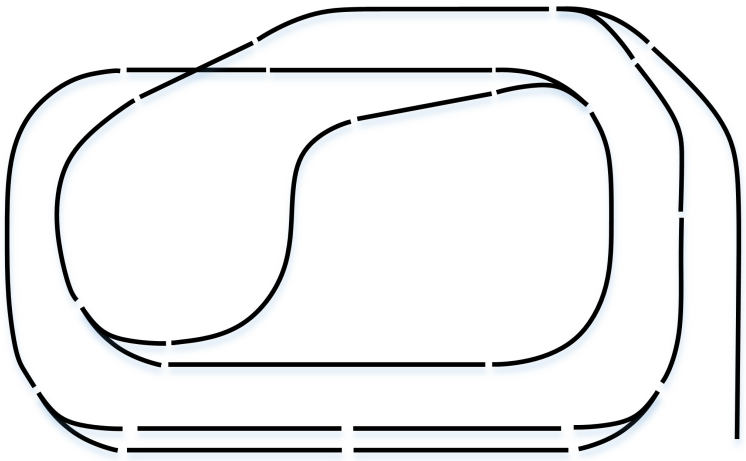


4.1. ábra. Tervezés során használt Y modell

4.2. Hardver felépítése

A vasúthálózat felépítésekor a legfontosabb szempont a komplexitás helyes meghatározása volt. Létre kellett hoznunk egy olyan pályát, amelyben akár veszélyes szituációk (úgymint mozdo-

nyok ütközése, váltók felvágása[1]) is előfordulhatnak; az elkészült pálya sematikus elrendezése a 4.2 ábrán látható. A mozdonyok egymástól független, párhuzamos irányítását csak digitális vezérléssel lehet megvalósítani, amelynek alapja a DCC protokoll [24]. A protokoll zártsága is indokolta a vezérléstől elszeparált biztonsági rendszer kiépítését.

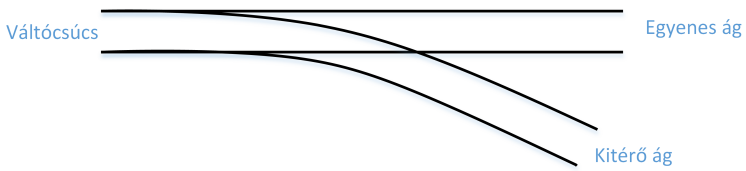


4.2. ábra. Vasúthálózat sematikus képe

4.2.1. Információk a pályáról

A biztonsági rendszer működéséhez szükség volt a pályán mozgó mozdonyok pozíciójára, valamint a váltók aktuális állására. A pozíció visszafejtése szakaszokra bontáson alapul: a sín megszakításával megoldható, hogy a pálya egyes részeit külön egységként lehessen kezelni, és ezeken az egységeken való mozdonytartózkodást feleltetjük meg a mozdonyok pozíciójának. A szakaszon belüli tartózkodását viszont nem kell ismernünk, hiszen a pálya tervezése során figyeltünk arra, hogy minden szakasz legalább olyan hosszú legyen, mint a leghosszabb mozdonyunk. Az elkészült pályán a szakaszhatárokat a sín vonalát megszakítva jeleztük a 4.2 ábrán. A komplexitást növeli a pálya közepén látható hurokvágány, hiszen erre a pályarészre érkező mozdonyok folytonos előrehaladással is menetirányt tudnak váltani.

A váltók állásának megismeréséhez felhasználjuk a váltókat mozgató elektromágneses állítóműveket, ahol a vezérlő elektronikához hozzáépítettünk egy saját feszültségmérő egységet, így a két állást (kitérő vagy egyenes állás – a 4.3 ábra) egyértelműen meg tudjuk különböztetni.



4.3. ábra. Váltó részek nevei

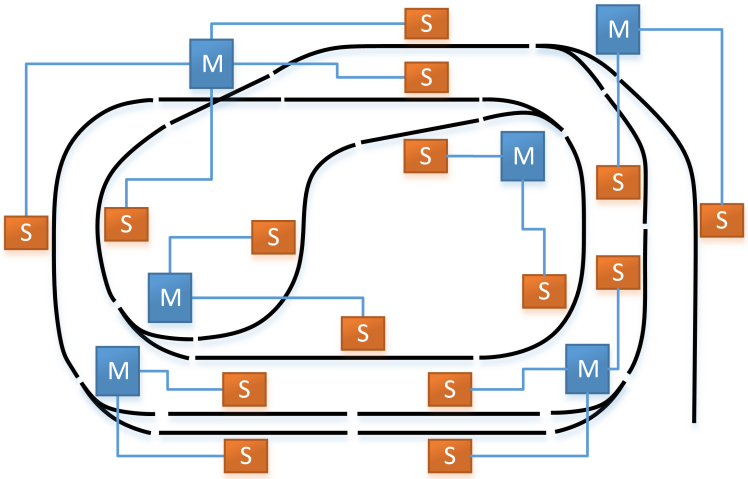
[1]Váltófelvágásnak nevezzük azt a tiltott cselekményt, amikor egy, a váltócsúccsal szemben érkező vasúti jármű a váltót a kerekeivel állítja át.

4.2.2. Architektúra

Elosztott, mikrokontroller-alapú hálózat

A biztonsági rendszer megvalósításához kétrétegű, mikrokontrollerekből álló hálózatot alakítottunk ki, amely során megkülönböztetünk *master* és *slave* egységeket, melyek kapcsolata és felelősségi köre a 4.5 ábrán látható. A master egységek felelősek a biztonsági logika döntéseinek végrehajtásáért, a hálózatba való becsatlakozásért, a hozzájuk rendelt váltók állásáról való informálásért, valamint a slave egységek utasításáért. A slave egységek pedig egy-egy szakasz engedélyezéséért felelősek. Ha a szakasz engedélyezett, akkor az arra belépő mozdonyoknak szabad a mozgás, míg tiltott állásban a szakaszra való belépés után meg kell állniuk.

Master egységeknek 7 db Arduino [25] kártyát választottunk, amelyből hat végrehajtó és egy foglaltság-érzékelésért felelős egység került a rendszerbe. A slave egységekből a szakaszok számával megegyező számú kellett, így 15 db került elhelyezésre a 4.4 ábrán látható módon.



4.4. ábra. Beavatkozó egységek elhelyezkedése a hálózatban

Az így kialakult hierarchia előnye a könnyebb telepíthetőség, ugyanis a slave egységek funkcionalitása nem bővül annyira dinamikusan, mint a mastereké, így azok implementálása egyszeri alkalommal megoldható volt. A masterek fejlesztése viszont inkrementálisan történt. A folyamatos, újabb konfiguráció feltöltését megkönnyítette az Arduino keretrendszer adta soros porton való feltöltés lehetősége, valamint a keretrendszer által biztosított függvénykönyvtárak is.

Hardver igények és a választás elve

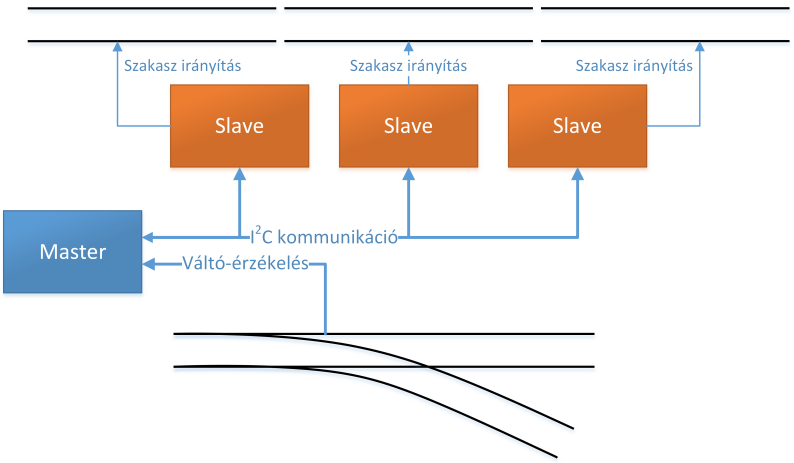
A slave egység feladataira az ATmega48-as mikrokontrollert választottunk, amely 4kB flash memóriája, 256B EEPROM, valamint 512B SRAM memóriája elegendő a célra, viszont gond nélkül kicserélhető az ATmega88, ATmega168 vagy ATmega328-as egységekkel, amelyek csak a fent említett paraméterekben különböznek. Ezzel szemben a master egységek komplexitása megkövetelte a nagyobb programtárhely és memória paramétereket, így az ATmega328-as chippel szerelt Arduinokat választottuk, amelyek 32kB flash memóriával, 1kB EEPROM nem felejtő memóriával és 2kB SRAM memóriával rendelkeznek.

Master egységek feladatköre

A hálózat gerincét alkotó Arduinok közül hat darab az elosztott logikát valósítja meg, míg egy egységeknek a feladata a teljes pálya foglaltságának ismerete és kérés esetén való kiszolgálása: ezt az egységet nevezzük SOC-nek (*Section Occupancy Collector*), azaz szakaszfoglaltság-gyűjtőnek. Dedikált funcionalitása annak köszönhető, hogy a szakaszok foglaltsága alapvető eleme a biztonsági logikánknak, így ennek az információnak kell a legfrissebbnek és legpontosabbnak lennie.

A további hat egység azonos feladatkörrel rendelkezik:

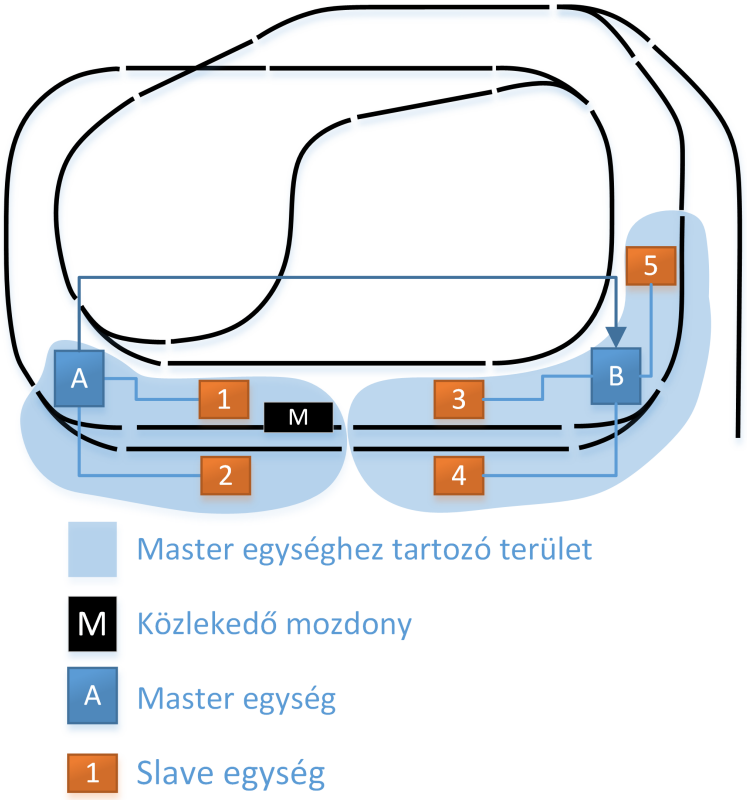
- megvalósítják azt a biztonsági logikát, amely biztosítja az egyes vonatokat az adott váltó területén, így nem ütközhetnek ezeken a szakaszokon (lokális döntés),
- kommunikálnak a szomszédos váltókhoz tartozó masterekkel, így a területről való kihajtást vagy behajtást is kölcsönösen jóvá kell hagyni a szomszédos mastereknek (globális döntés),
- kommunikálnak a slave egységekkel, amelyeket utasítják az egyes szakaszok engedélyezésével kapcsolatban,
- valamint folyamatosan figyelik a hozzájuk tartozó váltók állását.



4.5. ábra. Master-slave kapcsolat és felelőségek

A távoli kérés (globális döntés) fogalma a 4.6 ábrán látható: az *A jelű* master egy távoli kéréssel fordul a *B jelű* egységhez, hogy az ő területét elhagyó mozdony tovább haladhat-e a másik területre. A *B jelű* master erre az aktuális állapotától függően reagál, így az *A jelű* egység lokális döntése – miszerint a mozdony közlekedhet azon a szakaszon – felülíródik a távoli kérés során kapott válasszal.

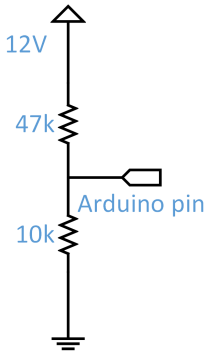
A váltók irányának a visszafejtése az elektromágneses állítóműveken keresztül oldható meg. A vezérlőkörbe iktatott dekóder három vezetéken keresztül irányítja az egyes állítóműveket: a két álláshoz tartozó pozitív potenciálú vezetéken ad ki $12V$ feszültséget, majd a közös földvezetékkel zárja az áramkört. Emiatt elegendő megmérnünk a két ág a közös földhöz mért potenciálkülönbségét, a master egységek két analóg kivezetésén ezek valósulnak meg. Ezekhez a kivezetésekhez egyenként egy-egy integrált ADC (*Analog to Digital Converter*) tartozik, így a kapott két érté-



4.6. ábra. Távoli kérés két master egység között

ket egymáshoz komparálva megadható, melyik irányban van nagyobb potenciálkülönbség. Fontos megjegyezni, hogy nem egy egzakt értékhez, hanem egymáshoz hasonlítjuk a két értéket, hiszen az áramkörben történhetnek olyan kilengések, amelyek a maximális feszültséget eltolhatják valamely irányba[2], valamint a minimális feszültség sem egyenlő mindig a referencia-feszültséggel, a környezetből felvett töltések miatt.

A megoldáshoz hozzátartozik továbbá egy saját feszültségosztó áramkör is, mer az Arduino eszközök $3.3V$-os rendszerfeszültségűek. A bemenetekre kötött $12V$ tönkretenné azokat, így szükséges a bemenetekre kapcsolt feszültség megfelelő skálázása. A megoldáshoz felhasznált két ellenállás pontatlansága is befolyásolja a kapott értékeket, így a konkrét értékhez való komparálás nem működne minden esetben.



4.7. ábra. Feszültségosztó áramkör

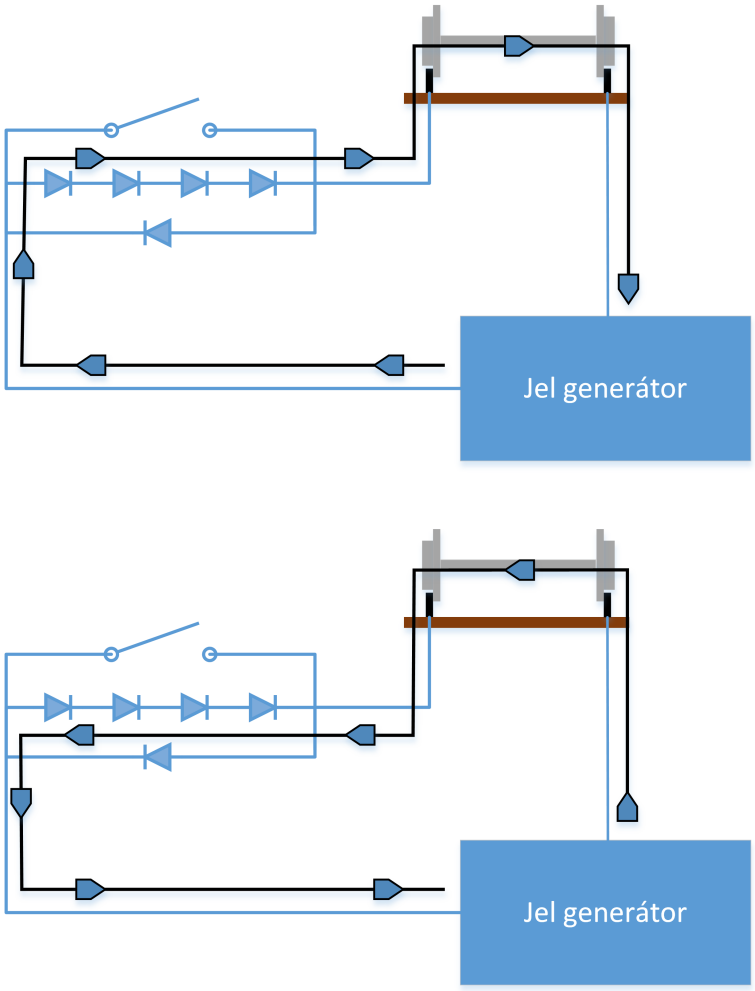
Mivel a feszültség mérésénél nincs szükségünk a pontos feszültségértékre, hanem csak a két érték arányára, ezért az ellenállások megválasztásánál $18V$ bemenő feszültségig jól működő alkatrészeket választottunk, a megoldás a 4.7 ábrán látható. Az áramkör másik előnye, hogy így a lapka bemenetét terhelő áramerősséget is tized milliamper nagyságrendűre korlátozzuk.

[2]kb. $11.54V - 12.1V$ közötti értékeket vehet fel

Slave egységek feladatköre

A slave egységek feladata jelenleg a szakaszok engedélyezésére korlátozódik. A tervezés során szempont volt a kiterjeszthetőség, így későbbi új funkciók beépítése – úgymint jelzők kezelése – könnyen megvalósíthatóak. Jelenleg a slave-ek számontartják (és kiajánlják a masternek) a hozzájuk tartozó szakaszok állapotát, illetve ha a master felől ennek a státusznak a megváltoztatására való igény érkezik, akkor kiszolgálják azt.

A szakaszok engedélyezése valójában a mozdonyok megállítását vagy továbbengedését jelenti; a zárt DCC jelet módosítjuk néhány passzív elektronikai eszközzel. Ezt érzékeli a mozdonyba épített dekóder, amely ennek hatására lefékezi a vonatot. A fejlesztő Lenz cég ezt a funkcionalitást ABC módnak [26] nevezte el.



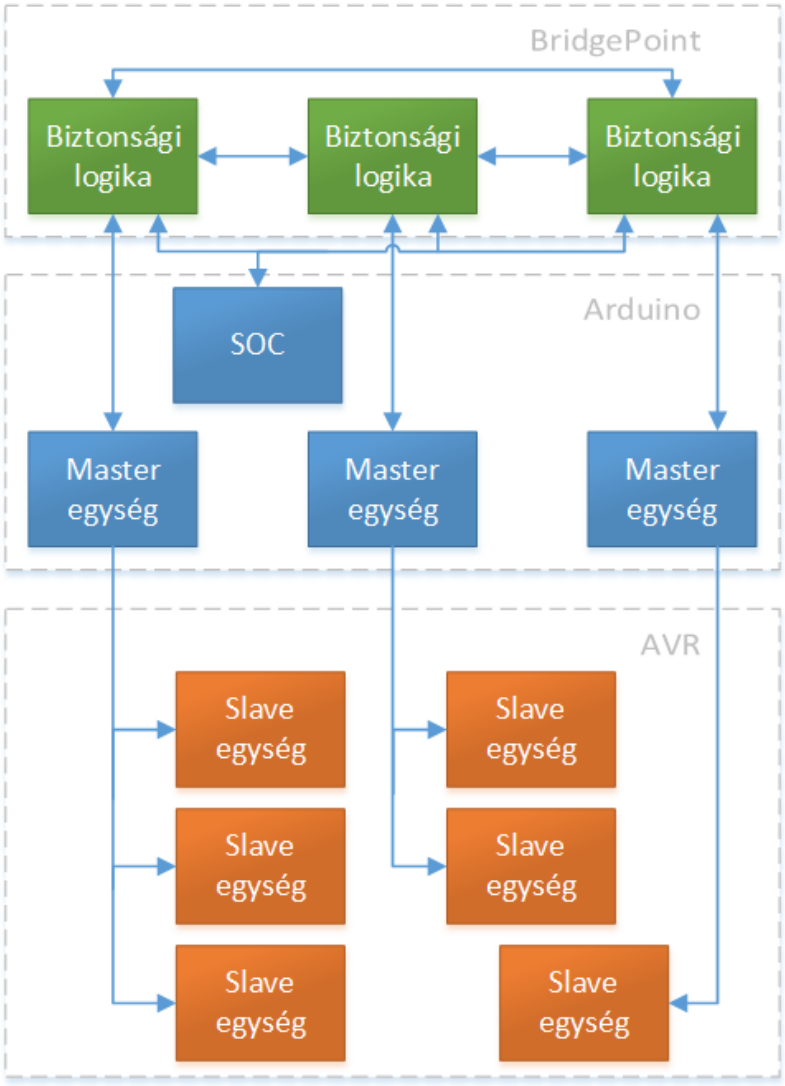
4.8. ábra. A váltóáram útvonala a diódákkal kialakított áramkörben

Az ABC mód a 4.8 ábrán látható módon működik. A váltóáram körébe – amely a mozdonyokat irányítja – két irányban 4+1 db diódát tettünk, így az egyik irányba $4 \cdot 0.7V = 2.8V$, a másik irányba $0.7V$ esik a feszültségből. Ezt az aszimmetriát érzékelik a dekóderek, melynek hatására megállnak a vonatok az adott szakaszon. Ezt a működést beszüntethetjük, ha a harmadik ágon lévő relét aktiváljuk, kiiktatva az áramkörből az 5 db diódát.

Kommunikáció

A rendszer hierarchiája a kommunikációban is megmutatkozik, mivel a masterek egymás között UDP [27] (Ethernet [28]) protokollon, a master és slave egységek pedig egymás között I^2C [29] protokollon közölnek információkat. Az Ethernet protokollt univerzalitása tette alkalmassá a feladatra, hiszen ennek köszönhetően más gépek, programok is beköthetőek a hálózatba, megadva a lehetőséget a masterekkel való kommunikációra. Az I^2C két vezetéken alapuló interfésze egyszerű csatlakozást ad a slave egységekhez, valamint elég stabil és gyors kisebb információk küldésére: az implementált kommunikáció során 1-2 Byte-nyi adatokkal dolgozunk.

A vasútmodellezésben leggyakrabban használt kommunikációs protokoll az S88 [30], amely működését tekintve az SPI-hoz hasonlít a duplexitás hiányát leszámítva. A kommunikáló egységek felé nem lehet adatot küldeni csak adatot fogadni, valamint biztosítani kell a kommunikációhoz szükséges órajelet, vezérlőjeleket is. A szakaszérzékeléshez használt eszköztől is ezzel a protokollal lehet a foglaltsági adatokat lekérdezni, melynek során az egy soros kimenetű shiftregiszterként funkcionál.



4.9. ábra. A hálózat összeköttetése a BridgePoint állapotgépekkel

4.3. Konfiguráció a BridgePoint-tal való együttműködéshez

A hardver architektúrát a BridgePoint-ban általunk létrehozott biztonsági logikával a 4.9 ábrán látható módon kapcsoltuk össze.

A biztonsági logikát a BridgePoint modellek valósították meg saját állapotgépeikkel, amelyek a 4.4.4 szakaszban leírtak alapján viselkednek. A BridgePoint lehetőséget biztosít arra, hogy a modellt úgynevezett realizált komponensen keresztül kapcsoljuk össze a környezetével, melyben platformspecifikus C++ vagy Java kód futtatására van lehetőség.

Minden master egységhez egy, a BridgePoint szimulátorában futtatott példánymodellt társítottunk. A master egységek statikus, parancsokon alapuló funkcionalitással rendelkeztek, és a realizált komponenseken keresztül egy általunk definiált formátum alapján kommunikáltak a BridgePoint állapotgépekkel.

Egy külön interfészen keresztül a master egységek a hozzájuk tartozó slave egységekkel kommunikáltak, amelyek a BridgePoint állapotgépek által hozott szakasz lekapcsolási és engedélyezési döntések fizikai végrehajtói voltak.

Ezáltal megvalósítottuk a 4.4.7 szakaszban bemutatandó, modellszinten megalkotott biztonsági logika összekapcsolódását a hardver architektúrával, így a modellvasút terepasztallal is.

4.4. Biztonsági logika modell alapú tervezése

4.4.1. Magasszintű követelmények

A vonatok a bennük lévő digitális dekóderek alapján egy erre szolgáló céleszköz segítségével címezhetőek, és egy kommunikációs protokollon keresztül irányíthatóak. Ettől a protokolltól és irányítási alrendszertől függetlenül dolgoztunk ki egy olyan elosztott, komplex biztonsági logikát, ami megakadályozza két, vagy több, digitális dekóderrel rendelkező mozdony összeütközését a pályán.

A digitális modellvasút fizikai tulajdonságai miatt a vonatok egymástól függetlenül mindkét irányban közeledhetnek. Emiatt biztosítani kell, hogy ha a vonat haladás közben vagy megállás után váratlanul haladási irányt változtat, akkor se alakulhasson ki kritikus szituáció, ne ütközhessenek össze a mozdonyok egymással.

A 4.1 fejezet első bekezdésében említett okok következtében nem tudunk egyértelmű menetirányt definiálni a pályán, ezért biztosítani kell, hogy bármely két mozdony között mindig legyen legalább egy egységnyi szabad távolság mindkét irányban. Egy egység a vasúthálózat sematikus képén (a 4.2 ábrán) folytonos vonallal jelölt szakasz vagy váltó. Ha ez a követelmény nem teljesíthető, akkor a közvetlenül szomszédos szakaszokon lévő mozdonyokat feltétlenül meg kell állítani.

4.4.2. Specifikáció

A követelmények alapján elkészítettük a rendszer specifikációját. A biztonsági logika vonatok vezérléstől való szeparációjának teljesítéséhez egy teljesen független alrendszert hoztunk létre.

Az alrendszernek egyedül a szakaszok és váltók (pályaegységek) foglaltságát kell ismernie, amit egy dedikált hardveregységtől kap meg egy szám formájában. A szám bitjei, mint pályaegység azonosítóként tekintett indexnek felelnek meg. Az adott helyiértékű bit jelöli a hozzá tartozó egység foglaltságát. Egy pályaegységet foglaltnak tekintünk, ha van rajta mozdony, vagy más, ellenállással rendelkező tárgy vagy élőlény.

Az adott szakaszon a vonatok megállítása és engedélyezése a slave hardver egységek feladata. Az egyes slave egységeket a hozzájuk tartozó master egységek utasítják kizárólag. Tehát a biztonsági logikát megvalósító modellnek a master egységeken keresztül parancsot adniuk a szakaszok le- és felkapcsolására, a vonatok és váltók mindenkori állásától függően.

A pálya mindenkori állapotának pontos leírásához a pályaegységek foglaltságán és engedélyezettségén kívül a modellnek szükséges a váltók állásáról is ismerettel rendelkeznie. Ezt az információt az adott váltóhoz tartozó master egységtől kapja egy számérték formájában, amely érték tárolja, hogy a váltó melyik irányban (kitérő, vagy egyenes) állt a mérés pillanatában.

A vonatok összeütközésének magakadályozására a feladat komplexitásának kezelése érdekében szükséges, hogy a biztonsági logikát elosztottan, több szintre bontva valósítsuk meg. Az elosztottságot a pályához tartozó váltók mentén teljesítettük.

4.4.3. Tervezési feltételezések és kényszerek

A magasszintű specifikáció és a követelmények teljesítéséhez a biztonsági logika környezetére nézve az alábbi feltételezéseket tettük:

- Sínpályával és mozdonyokkal kapcsolatos feltételezések:
  - A sínpályán ellenállással rendelkező tárgyakon kívül más tárgy nem tartózkodhat, különben a hardver architektúra nem tudja érzékelni az adott szakasz foglaltságát.
  - Mozdonyok nem állhatnak meg a váltókon, mert ha közben a váltót átállítják, akkor az adott mozdony kisiklik, ezzel rövidzárlatot okozva a digitális vezérlésben.
  - Tilos egy váltót átállítani másik állásba, amíg egy mozdony tartózkodik, vagy áthalad rajta, az előző pontban említett okok miatt.
  - Kezdeti állapotban nem helyezhető két mozdony ugyanarra a szakaszra vagy két szakasz közti határra, mert a biztonsági logika működését megzavarja.
  - A mozdonyok nem közlekedhetnek tetszőlegesen nagy sebességgel a pályán, mert a BridgePoint szimulátorában való futtatás miatt a biztonsági logikát megvalósító modellek véges reakcióidővel rendelkeznek (lásd a 4.4.9 szakaszban).
- Elosztott biztonsági vezérlőkkel kapcsolatos feltételezések:
  - A vezérlők közti kommunikáció gyorsabb, mint a vonatok szakaszokon való áthaladásának sebessége azért, hogy a biztonsági logika időben tudjon reagálni a változásokra.
  - Kommunikációs csomag nem veszik el a vezérlőket összekötő hálózaton, hogy a biztonsági logika által hozott döntések érvényre jussanak.

  - A vezérlőknek a hozzájuk tartozó szakaszok foglaltsága mindig rendelkezésre áll, mert a biztonsági logika működése ezen alapul.
  - Egy vezérlő meghibásodása vagy leállása esetén, a vezérlőhöz tartozó szakaszokon a biztonsági logika helyes működése nem garantált.
  - A master biztonsági vezérlőkhöz tartozó slave egységek, vagy a köztük lévő kommunikációs csatorna meghibásodása esetén a hozzá tartozó szakaszon a mozdonyok megállítása, ill. engedélyezése fizikai okok miatt nem biztosított.

- A vonatok vezérlését végző gyári architektúrával kapcsolatos feltételezések:
  - Ha egy vonat kisiklik, akkor rövidzárlatot okoz.
  - Rövidzárlat esetén a digitális jeleket előállító vezérlődoboz áramtalanítja a pályát. A vonatok megállnak, ezáltal megakadályozva a rövidzárlat okozta meghibásodást a mozdonyokban lévő dekóderekben.
  - A vonatokba épített dekóderek úgy vannak konfigurálva, hogy ha a két sínszálban eltérő feszültségszintet mérnek, akkor megállnak és nem indulnak el egyik irányban sem. Ezáltal a biztonsági logika meg tudja állítani a vonatokat.

A feltételezések teljesülését az általunk kidolgozott biztonsági alrendszer nem biztosítja, annak helyes működésének szükséges, de nem elégséges feltételei.

4.4.4. Szakasz lezárási protokoll

A specifikáció alapján a követelmények teljesítéséhez elkészítettünk egy szakasz lezárási protokollt, amely alapján meg lehet határozni, hogy milyen esetben szükséges a vonatok megállítása.

Egy váltó területének a váltót és a hozzá közvetlenül csatlakozó szakaszok összességét tekintjük. Minden váltóhoz tartozik egy lokális biztonsági logika, amely az adott váltóhoz csatlakozó szakaszok foglaltsága alapján dönti el, hogy melyik szakaszt szükséges lekapcsolni.

A biztonságkritikus szituációk elkerülésének szükséges, de nem elégséges feltételei a lokális döntések. Az elégségesség teljesítéséhez a szomszédos váltókhoz tartozó szakaszok állapotait is ismerni kell, ezért az egyes váltókhoz tartozó biztonsági logika modelleknek kommunikálniuk kell egymással. Az elosztott kommunikációhoz tartozó döntéseknek és a lokális döntéseknek együtt kell érvényre jutniuk és bármelyik döntésben egy szakasz lekapcsolása szerepel, akkor annak a döntésnek a 4.4.3 szakaszban írt feltételezések figyelembevételével érvényre kell jutnia.

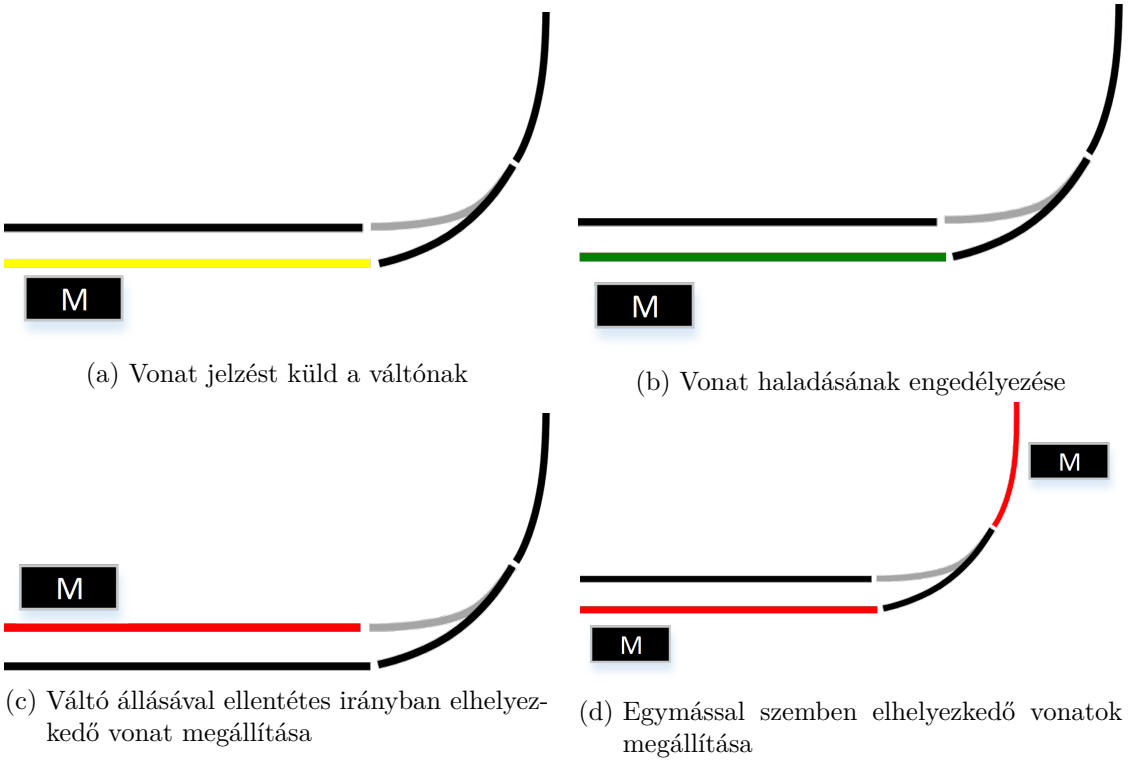
Ha egy mozdony egy váltóhoz tartozó szakaszra ér, akkor az adott váltó *lokális döntése*, és a vele szomszédos váltók *globális döntése* alapján határozódik meg, hogy szükséges-e a mozdonyt megállítani.

Lokális döntés

Lokális döntés esetén csak az adott váltóhoz csatlakozó szakaszok foglaltsága alapján határozódik meg, hogy melyik mozdonyt szükséges megállítani.

Egy mozdony, amikor a váltóhoz csatlakozó bármely szakasz területére érkezik, akkor az adott szakasz foglalt lesz, melyről a hozzá tartozó váltó egy jelzést kap. A váltó megvizsgálja, hogy a jelenlegi állásával ellentétes irányból érkezett-e a jelzést.

Amennyiben igen, akkor egy tiltó üzenetet küld a vonathoz tartozó szakaszt kezelő mikrokontrollernek. Ennek hatására a mikrokontroller megállítja a vonatot.



(a) Vonat jelzést küld a váltónak

(b) Vonat haladásának engedélyezése

(c) Váltó állásával ellentétes irányban elhelyezkedő vonat megállítása

(d) Egymással szemben elhelyezkedő vonatok megállítása

4.10. ábra. Szakasz lezárási protokoll lokális döntés

Ha a váltó állásával megegyező irányból érkezett a jelzés, akkor a váltó megvizsgálja a vonat állásától nézve a váltó túlsó oldalán, folytonosan kapcsolódó szakasz foglaltságát. Amennyiben azon a szakaszon nincsen vonat, akkor engedélyezi a mozdony áthaladását. Amennyiben van a túlsó oldalon vonat, akkor mindkét szakaszhoz tartozó mikrokontrollernek tiltó üzenetet küld a vonatok megállítása céljából.

Váltóhoz tartozó *lokális döntés* esetén a váltó foglaltságát nem vizsgáljuk, mert feltételezzük, hogy azon vonat nem állhat meg, illetve a szakasz lezárási protokollnak azelőtt kell megállítania a *globális döntés* alapján a vonatot, hogy az a váltó területére érne.

Globális döntés

Globális döntés esetén a *szomszédos* váltók a *lokális* információikat felhasználva eldöntik, hogy melyik hozzájuk tartozó szakasz foglaltsága esetén melyik szomszédos váltónak kell jelzést küldeni. A szomszédos váltótól kapott jelzések és a lokális információk alapján meghatározzák, mely szakaszok lekapcsolása szükséges a biztonságkritikus szituációk elkerülése érdekében.

Egy helyi váltó engedélyezi egy szomszédos váltó területéről a vonat beérkezését, ha:

- a váltó területén nincsen vonat.

- a váltó területére úgy érkezik, hogy az érkezési szakasz csatlakozási iránya a váltó állásával ellentétes és szabad.
- a váltó területére úgy érkezik, hogy az érkezési szakasz csatlakozási iránya a váltó állásával megegyező, nincsen rajta vonat és a váltó, valamint a túloldalán folytonosan csatlakozó szakasz is szabad.

Egy helyi váltó megtiltja egy szomszédos váltó területéről a vonat beérkezését, ha:

- a váltó foglalt.
- a váltó területére úgy érkezik, hogy az érkezési szakasz foglalt. Utóbbi esetben a helyi váltó területén lévő vonatot is megállítja.
- a helyi váltót, a szomszédos váltótól érkező kérés elküldésének és a helyi váltón történő feldolgozásának pillanata között, átállítják a korábbi állásától eltérő irányba.
- a helyi váltó egy másik, szomszédos váltótól érkező kérést szolgál ki.

Ezekben az esetekben a helyi váltó annak a szomszédos váltónak küld tiltó üzenetet, amelyik számára megtiltja annak a vonatnak az érkezését. Ennek hatására az említett szomszédos váltó megállítja a területén lévő vonatot, a szakaszhoz tartozó mikrokontrolleren keresztül.

A 4.2 ábrán látható pályarajz felső részén, bal oldalon elhelyezkedő, két egymást metsző szárral rendelkező váltót nevezzük angolváltónak, amellyel szomszédos váltók és közötte egy, míg a többi esetben a szomszédos váltók között két szabad szakasz van.

Annak érdekében, hogy az esettanulmány bemutatása során elkerüljük, hogy két, szomszédos váltó területén, de egymástól még legalább két szabad szakasz és váltó távolságra lévő vonatok megálljanak, prioritásokat rendeltünk minden váltóhoz. Ez alapján a tiltási szabály a következő kiegészítéssel együtt érvényes:

Egy helyi váltó megtiltja egy szomszédos váltó területéről a vonat beérkezését, ha az érkezési szakasz csatlakozási iránya a váltó állásával megegyező, nincsen rajta vonat, a váltó szabad, de a váltó túloldalán folytonosan csatlakozó szakasz foglalt és a szomszédos váltó prioritása kisebb a helyi váltó prioritásánál (lásd a 4.11b ábrán).

Lokális és globális döntések kiértékelése

Egy adott váltóhoz tartozó lokális és globális döntések kiértékelése során az ellentmondások elkerülése érdekében prioritásokat rendelünk az egyes döntésekhez. Egy váltóhoz tartozó szakaszra nézve tiltást elrendelő döntések nagyobb prioritással rendelkeznek az engedélyező döntéseknél. Ennek megfelelően ezeknek mindenképpen végre kell hajtódniuk és az adott szakaszhoz tartozó beágyazott mikrovezérlőnek meg kell állítania a szakaszon lévő vonatot.

Ha egy szakaszra nézve a lokális és a globális döntések egymásnak ellentmondó eredményt tartalmaznak, akkor a tiltó döntés jut érvényre az engedélyező döntéssel szemben.

A lokális és a globális döntések kiértékelése alapján biztosítható, hogy ha egy vonatot megállító döntés létrejött, akkor annak eredményéhez vezető cselekmények logikai szinten megtörténjenek. A fizikai végrehajtás biztosításához további hibatűrő eljárások alkalmazása szükséges.

(a) Vonatok haladásának engedélyezése mindkét váltóra

(b) Vonat megállítása prioritás alapján

4.11. ábra. Szakasz lezárási protokoll globális döntés

4.4.5. Szakasz engedélyezési protokoll

A modellvasút-terepasztal működésének demonstrálása során el akartuk kerülni, hogy ha egy vonatot megállított a biztonsági logika, akkor azt a vonatot csak emberi beavatkozással lehessen újra elindítani. Ezért a szakasz *lezárási* protokollon kívül kidolgoztunk és megvalósítottunk egy szakasz *engedélyezési* protokollt is.

A szakasz *engedélyezési* protokoll egy olyan protokoll, amely biztosítja a vonat megállítását okozó kényszerek megszűnése esetén a vonat továbbhaladását. Működését tekintve teljes egészében megegyezik a szakasz *lezárási* protokollal.

A szakasz *engedélyezési* protokoll végrehajtódik, ha egy váltót az aktuális állási irányából átváltanak egy másik irányba. Ekkor a váltó megvizsgálja, hogy az így folytonosan kapcsolódó szakaszok közül csak pontosan az egyiken van-e vonat. Ha igen, akkor engedélyezi, különben a szakasz *lezárási* protokoll alapján megtiltja a vonatnak a továbbhaladást.

A két protokollt együttesen alkalmazva megvalósítható, hogy a biztonsági logika megakadályozza a biztonságkritikus szituációk kialakulását,és azok megszűnése esetén engedélyezze a vonatok továbbhaladását.

4.4.6. Terminológia definiálása

A valóságnak a feladat szempontjából releváns absztrakciójához a 4.1 táblázat szerint definiáltuk a vasúti terminológiából kapcsolódó fogalmakat a biztonsági logika (*szakasz lezárási és engedélyezési protokoll*) modell alapú elkészítéséhez.

Vasúti szakkifejezésben két állomás közti térközök összességét nevezik szakasznak. Ettől eltérően, szakasznak mi azt a legkisebb granularitású pályaelemet neveztük, ami nem váltó. Erre

(a) Vonatok haladásának tiltása (b) Vonat haladásának engedélyezése

4.12. ábra. Szakasz engedélyezési protokoll

Fizikai szegmens	Absztrakciós szint	Jelentés
térköz	szakasz	A pálya legkisebb granularitású szegmense.
váltó	váltó	A pályán a vonatok továbbhaladásának irányát statikusan megváltoztató elem.
pályaelem-foglaltság	pályaelem-foglaltság	Egy adott szakaszon, vagy váltón tartózkodik-e vonat.
biztosítóberendezés	vezérlő és megszakító-berendezés	Biztonságkritikus szituáció kialakulása esetén megállítja a vonatokat.

4.1. táblázat. Vasúti terminológia leképzése

azért volt szükség, mert a terepasztalon nincsenek állomások, így nem szükséges a térköz fogalmát definiálni.

A biztosítóberendezések felelősek a vonatok pályán történő összeütközésének megakadályozására. A valóságban ez a berendezés nem szeparálódik két jól elhatárolható részre, kívülről egy egységként látszik.

Azért volt szükség a fogalom két részre bontása, mert minden váltóhoz tartozik egy master egység, amely az adott váltóhoz tartozó szakaszokon a biztonságkritikus szituációk kialakulásának megakadályozásáért felelős. Minden szakaszhoz tartozik egy slave egység, ami az adott szakaszon vonatok haladásának engedélyezéséért felelős. Ezáltal a master egységeket (biztonsági) vezérlőknek, a slave egységeket megszakító-berendezéseknek nevezzük.

A vezérlő és megszakító-berendezések modellszinten nem kerültek ábrázolásra, hozzájuk a biztonsági logika egy interfészen keresztül kapcsolódik.

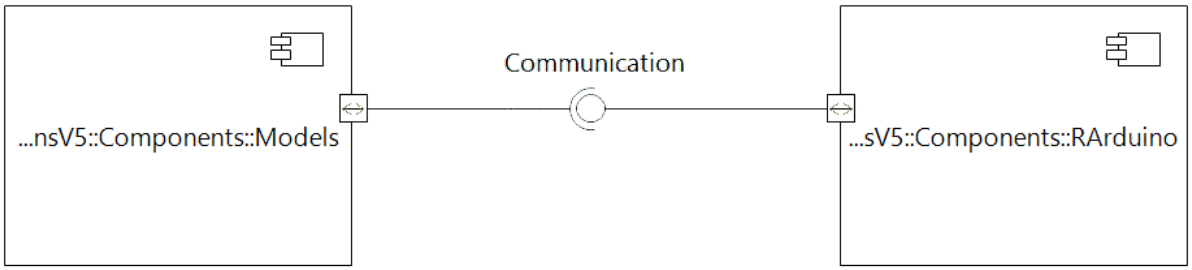
4.4.7. Biztonsági logika modell alapú tervezése

A biztonsági logika megvalósítása céljából a vasúti terminológiában szereplő fogalmak egy részét leképeztük modellekké. A modellek statikus részei az *UML* szabvány által definiált osztályokból, viselkedési részeinek leírása állapotgépszerű struktúrákból és egy, az *Executable UML* szemantikának megfelelő viselkedésleíró nyelvből, az *OAL*-ből (*Object Action Language*) épül fel.

A *Mentor Graphics*® által fejlesztett BridgePoint nevű, mérnöki modellek készítésére alkalmas tervezőszoftverben két nagy komponensre bontottuk a logikát a 4.13 ábrán látható módon. Az egyik komponens (*Models*) a modelleket, a másik (*RArduino*) a nem modellezhető részeket, natív forráskódot tartalmazza.

A két komponens egy interfészen (*Communication*) keresztül kommunikál egymással. Az interfészen keresztül történik a modell részére szükséges külső információk (pl. szakaszok foglalt-

sága, váltó állása), és a modellből a külvilág részére küldött üzenetek (pl. szakaszok letiltása, engedélyezése) továbbítása.



4.13. ábra. Modellezett és nem modellezett komponensek

A szakasz lezárási protokollban előforduló fogalmakat a modelleket tartalmazó komponensen belül leképeztük egy osztályhierarchiába. Az egyes osztályokból példányosított objektumok viselkedését állapotgépekben, a protokollbeli üzeneteket a modellekben az egyes osztályok közti jelzésekben (*signalokban*) írtuk le.

Minden osztály csak a saját működéséhez szükséges információval és funkcionalitással rendelkezik, ezáltal biztosítva a heterogén rendszerben az egységek közti felelősségek szétválasztását és a feladat komplexitásának csökkentését.

Szakasz dekompozíció

Egy modellvasút terepasztali szakaszt a modellben egy egyedileg azonosítható *Section* osztály reprezentál. A szakasz, foglaltsága alapján lehet szabad (*FreeSection*), vagy foglalt (*OccupiedSection*). Foglalt szakasznak egy olyan szakaszt nevezünk, amin legalább egy mozdony tartózkodik.

Egy szakaszra nézve egy ősosztálybeli példány és a két leszármazott osztályból mindig csak az egyik rendelkezik egy konkrét példánnyal a foglaltságtól függően. Ezáltal a szakasz viselkedése mindig csak a foglaltságától függ.

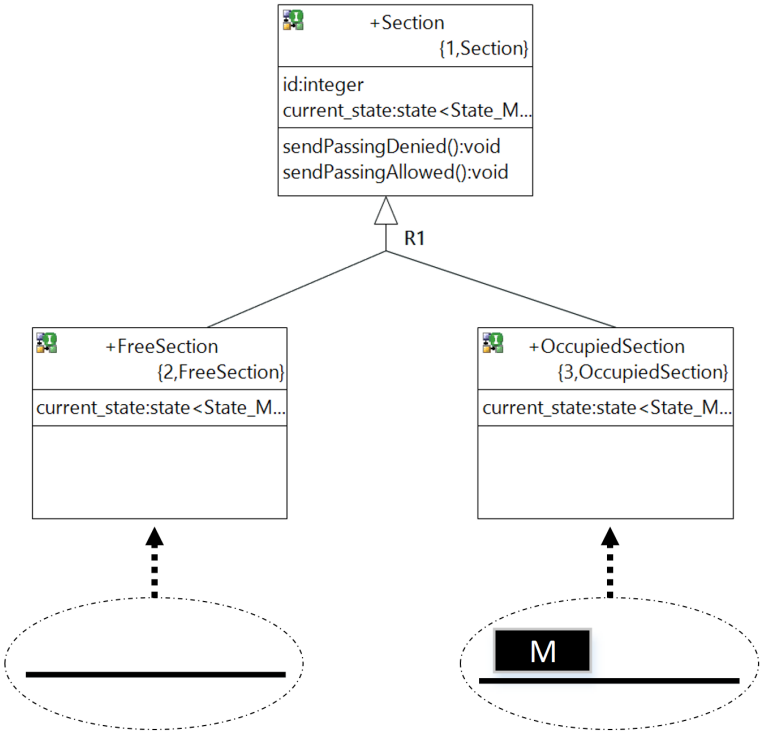
A szakasz a *lezárási protokoll* tekintetében három felelősséggel rendelkezik:

- Ha foglalt, akkor egy jelzést kell küldenie a váltónak a lezárási protokoll elindítására.
- Ha foglalt és egy státusz kérést kap a váltótól a protokoll alapján, akkor vissza kell utasítania, így jelezve, hogy rajta vonat tartózkodik.
- Ha szabad és egy státusz kérést kap a váltótól, akkor engedélyező választ ad a váltónak jelezve, hogy jöhet vonat.

A foglalt szakasz állapotgépének illusztrációja a Függelékek között, az F.3.4 ábrán látható. Az állapotgép a váltótól kapott események, jelzések hatására egy korábbi állapotból egy új állapotba megy át, végrehajtva az adott állapothoz vagy állapotátmenethez tartozó *OAL* kódot; ezáltal megvalósítva a *lezárási protokoll* adott szakaszra vonatkozó részét.

Váltó dekompozíció

Egy modellvasút terepasztali váltót a modellben egy egyedileg azonosítható *Turnout* osztály reprezentál. A szakasz lezárási protokoll *lokális döntésének* modellezéséhez elegendő a váltó



4.14. ábra. Szakaszok absztrakciója

egyenes (*StraightTunrout*), és kitérő (*DivergentTurnout*) állásbeli viselkedését megkülönböztetni, új osztályokra nincsen szükség.

A váltó a *lezárási protokoll* tekintetében a legnagyobb felelősséggel rendelkezik:

- Fogadja a kapcsolódó, foglalt szakaszoktól érkező kéréseket.
- Továbbítja a kapott kéréseket az állásától függő irányban csatlakozó szakasznak.
- A kapott választ továbbítja a kérést indító szakasznak.
- Visszautasítja az állásával ellentétes irányból érkezett kéréseket.

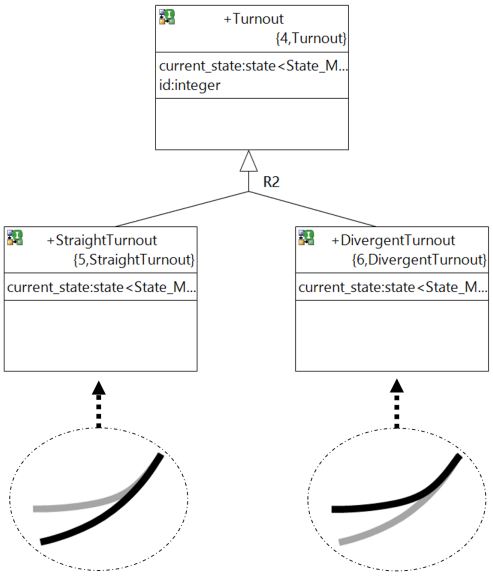
A váltóhoz tartozó szakaszok modellszinten a szakasz, váltóhoz való csatlakozási irányától függően tárolódnak a 4.16 ábrán látható módon.

Egyenes állású váltó lokális döntéseit leíró állapotgép illusztrációja a Függelékek között, az F.3.5 ábrán látható. A váltóhoz tartozó aktuális állapotgép határozza meg a kapott jelzések alapján, hogy mely szakasz lekapcsolása szükséges a biztonsági logika *lokális döntésének* megvalósítása céljából.

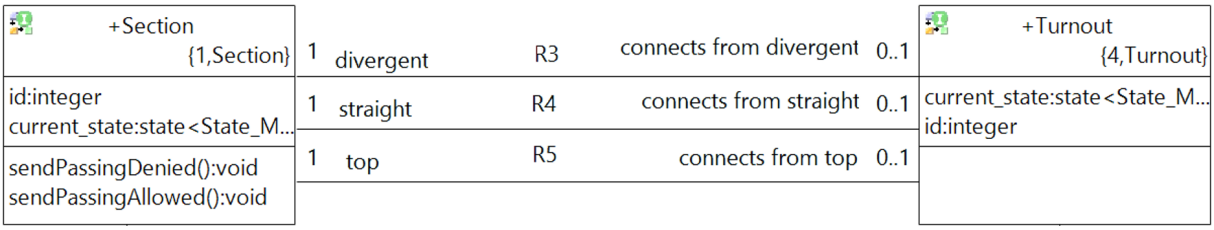
Angolváltó dekompozíció

Egy modellvasút terepasztali angolváltó a modellben két hagyományos váltó csúcs felőli egymáshoz csatlakoztatásának feleltethető meg.

A hagyományos váltóhoz hasonlóan az angolváltó esetében is, az angolváltó két oldalához kitérő, vagy egyenes irányból csatlakozó szakaszokat az adott váltó példány tárolja egy-egy asszociációval.



4.15. ábra. Váltó absztrakció



4.16. ábra. Szakaszok és a váltó kapcsolódása

Az angolváltó két részének állásától függő viselkedést leíró állapotgépek a hagyományos váltó megfelelő állapotgépeihez hasonlóak, felelősségi körük megegyezik a hagyományos váltók felelősségével.

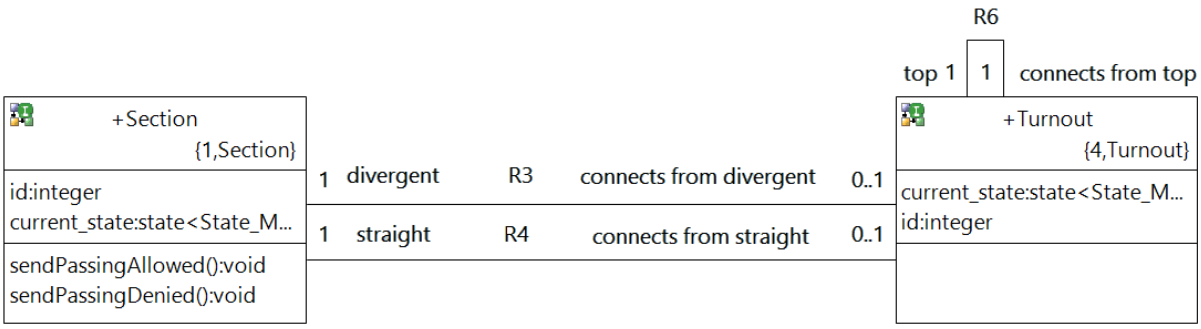
Szakasz lezárási protokoll globális döntése

A szakasz lezárási protokoll működésében nagy szerepe van a szakasz-foglalási kérések érkezési irányának. Emiatt a globális döntés meghozása során az egyes lokális váltókra nézve szükséges tárolni, hogy hozzájuk milyen irányból csatlakoznak a szomszédos váltók.

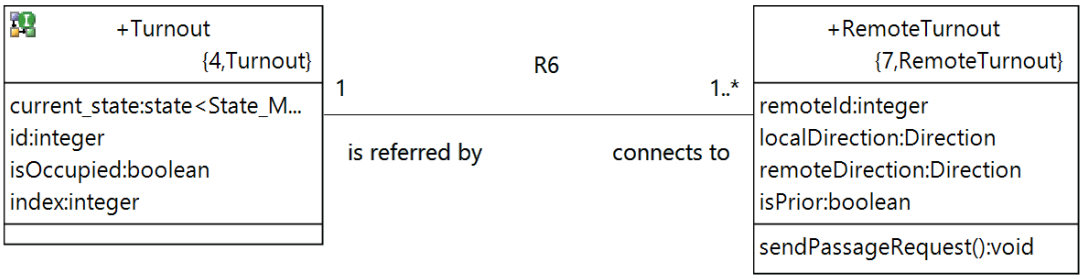
Ezért a modellben minden váltó objektumhoz tartozik egy szomszédos váltót reprezentáló távoli váltó (*RemoteTurnout*) példány, melynek segítségével a váltó állapotgépében a szomszédos váltóknak küldött irányfüggő kérések megadhatóak.

A folyamat komplexitásának kezeléséhez a lokális döntéshez képest a váltó állásának megkülönböztetésén kívül új osztályok bevezetése szükséges. Ezen osztályok a protokoll különböző rétegeit valósítják meg, ezáltal a felelősségek különválnak és a hozzájuk tartozó állapotgépek is egyszerűbb szerkezetűek lesznek. Ezt a hierarchiát ábrázolja a 4.19 ábra.

A hierarchia egyes szintjein azonos szerepkörű, de a váltó állásától függően eltérő működésű állapotgépeket tartalmazó osztályok vannak. A hierarchia gyökerében a lokális döntéseknél bevezetett váltót reprezentáló osztály áll.



4.17. ábra. Szakaszok és az angolváltó kapcsolódása



4.18. ábra. Távoli váltó és lokális váltó modellszintű kapcsolata

Az első szinten a szomszédos váltóktól érkező szakaszfoglalási kéréseket fogadó osztályok (*DispatchStraight, DispatchDivergent*) állnak. Ezen osztályok feladata a beérkező kérések transzformálása új, a hierarchia alsó szintjein lévő állapotgépek által feldolgozható jelzésekké, és egy kérés feldolgozása közben, a váltó állásával megegyező irányból érkező újabb kérések automatikus visszautasítása.

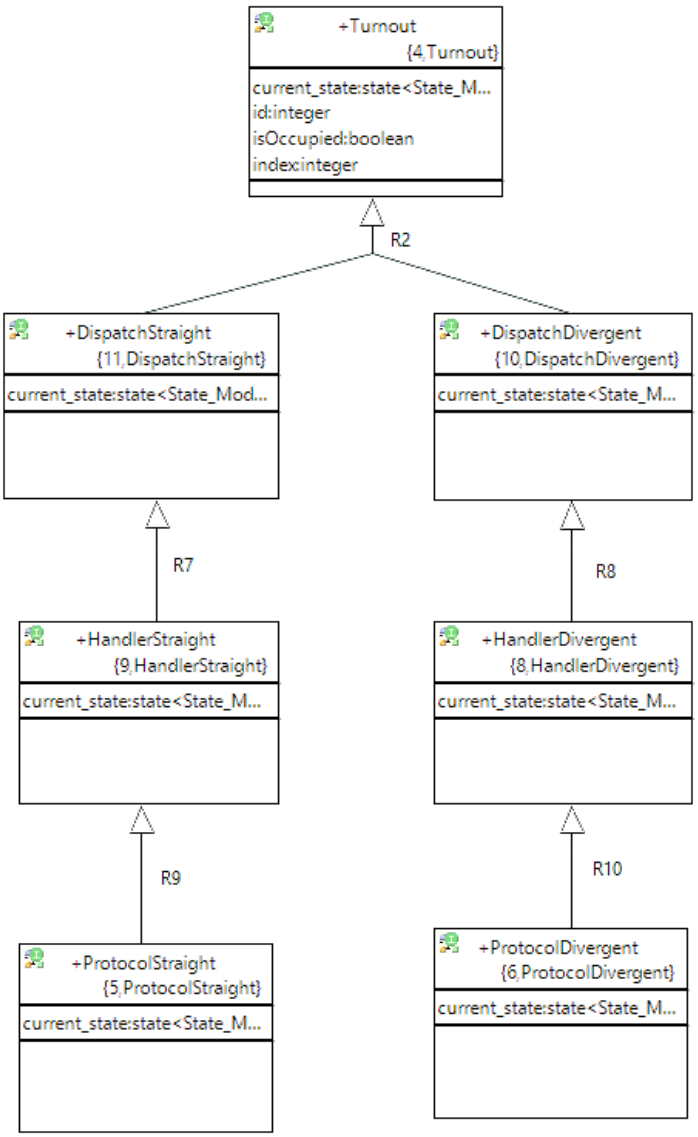
A második szinten a váltó állásával ellentétes irányból érkező kéréseket feldolgozó osztályok (*HandlerStraight, HandlerDivergent*) állnak. Ezen kéréseket a többitől függetlenül fel lehet dolgozni, ezáltal az állapotgépek működése egyszerűsödik.

A harmadik szinten lévő osztályokhoz (*ProtocolStraight, ProtocolDivergent*) tartozó állapotgépek feladata az adott váltóhoz tartozó lokális döntés és a szomszédos váltókkal közös globális döntés meghozása és végrehajtása. Ezen osztályokhoz tartozó állapotgépeknek struktúrája a legbonyolultabb a hierarchiában szereplő összes állapotgép közül, mert ők határozzák meg a kapott szakaszfoglalási kérések és válaszok alapján, hogy mely szakaszok lekapcsolása szükséges. Ezáltal a legnagyobb felelősséggel is ők rendelkeznek.

Az egyes szintekhez tartozó állapotgép részletek a Függelékben, az F.3 fejezetben megtalálhatók.

4.4.8. Váltóhoz tartozó biztonsági logika modellszintű kompozíciója

Az előző szakaszokban vázolt modellelemek kompozíciójával megalkottuk a váltókhoz tartozó biztonsági logika modelljeit. Kétféle modell készült: egy a három ággal rendelkező hagyományos váltókhoz és egy a négy ággal rendelkező angolváltóhoz. A hagyományos váltóhoz tartozó modell kompozíciója a 4.20 ábrán látható, melyen látható osztályok állapotgépei közül néhány illusztrációként megtalálható a Függelékek között az F.3 fejezetben. Az angolváltó modelljé-



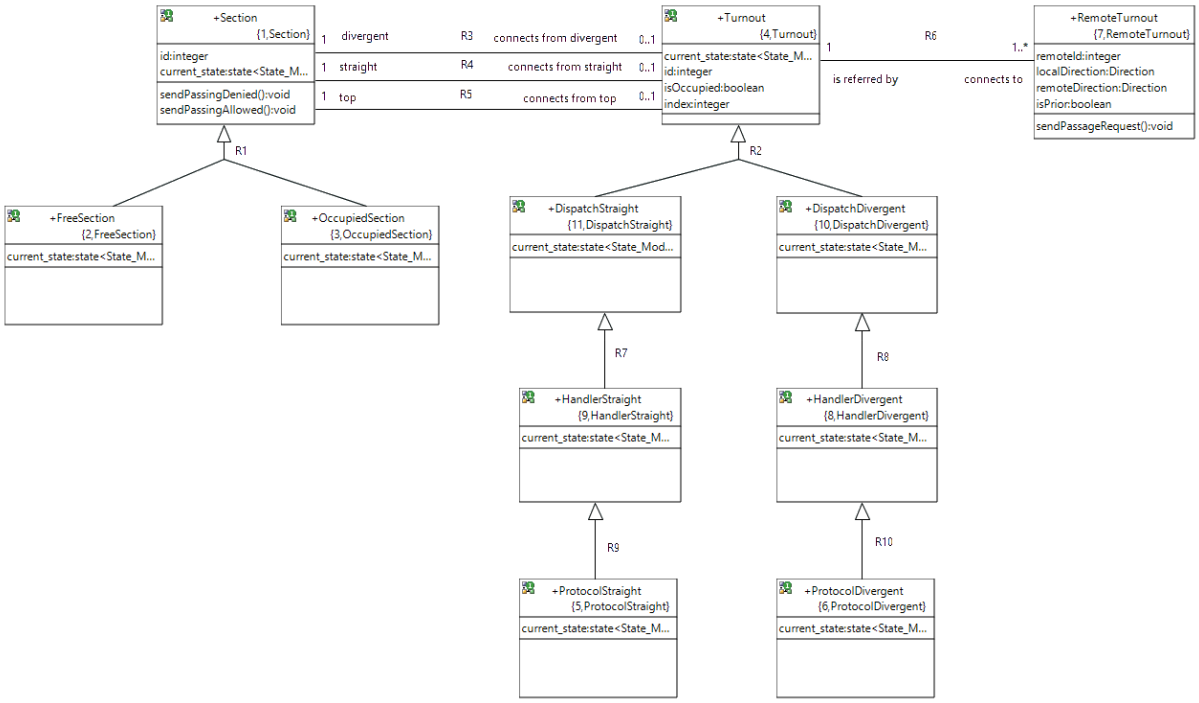
4.19. ábra. Globális döntéshez tartozó többszintű osztályhierarchia

nek strukturális kompozíciója a hagyományos váltó modelljéhez hasonló, egyedüli különbség a 4.17 ábrán látható váltó (*Turnout*) osztály önmagával vett asszociációja.

A modellezett és a nem modellezett (master egységekkel való kommunikációhoz szükséges) komponensek összekapcsolásával megvalósítottuk a váltókhoz tartozó biztonsági logika rendszerszintű modelljét. A biztonsági logikát alkotó modellek állapotgépeinek komplexitását jól jelzik a 4.2 táblázatban olvasható adatok.

Érdemes megfigyelni, hogy a lokális és a szomszédos váltókkal közös globális döntés meghozataláért felelős állapotgépek (*ProtocolStraight*, *ProtocolDivergent*) állapotainak száma hagyományos váltó és angolváltó esetén megegyezik, míg a lehetséges állapotátmenetek száma a hagyományos váltó esetén magasabb, mint az angolváltónál.

A többi esetben az angolváltó és a hagyományos váltó esetén eltérő állapotgépeknél a komplexitás (állapotok, illetve állapotátmenetek száma) az angolváltó esetén magasabb.



4.20. ábra. Hagyományos váltóhoz tartozó modell kompozíciója

4.4.9. Biztonsági logika szimulációja

Az elkészített modelleket az egyes váltókhoz tartozóan specializáltan példányosítva (melyre egy példa OAL kód a Függelékek között az F.2.1 fejezetben található), a BridgePoint szimulátorában futtatva szimuláltuk a biztonsági logika lokális döntésének működését.

Több példányban futtatva a BridgePoint szimulátorát, az egyes szimulátorokat futtató számítógépek és a terepasztal lokális, vezetékes hálózatának összekapcsolásával az egyes biztonsági logika modellpéldányok egymással és a hozzájuk tartozó master egységekkel kommunikálva megvalósítják a biztonsági logika elosztott működését, és a 4.21 ábrán látható módon megakadályozzák a vonatok összeütközését a modellvasút terepasztalon a 4.4.3 szakaszban írt feltételezések figyelembevételével.

4.5. Biztonsági logika verifikációja

Annak ellenére, hogy a megalkotott modell és a biztonsági logika helyesen működött a szimulációk során, előfordulhat, hogy egy olyan, a szimulációval nehezen előállítható állapotkonfiguráció áll elő, amely hibás viselkedéshez vezet. Ezért kiemelten fontos a biztonsági rendszereket nem csak kizárólag szimulációnak alávetni, hanem formálisan is verifikálni.

A formális verifikáció alatt azt értjük, hogy a verifikálni kívánt modellekre meghatározunk követelményeket valamely temporális logika segítségével, majd az összes lehetséges esetet (állapotkonfigurációt) megvizsgálva ellenőrizzük a követelmények teljesülését. A követelmény nem teljesülése egyúttal egy ellenpéldát is generál, melyet felhasználva felderíthető a hibás működést okozó komponens és javítható a nem megfelelő viselkedés, illetve konfiguráció.

Állapotgép neve		Állapotok száma	Átmenetek száma
FreeSection		2	4
OccupiedSection		4	17
DispatchStraight	hagyományos váltó	3	11
	angolváltó	4	19
HandlerStraight	hagyományos váltó	3	8
	angolváltó	3	9
ProtocolStraight	hagyományos váltó	10	57
	angolváltó	10	34
DispatchDivergent	hagyományos váltó	3	11
	angolváltó	4	19
HandlerDivergent	hagyományos váltó	3	8
	angolváltó	3	9
ProtocolDivergent	hagyományos váltó	10	53
	angolváltó	10	34

4.2. táblázat. Biztonsági logika állapotgépeinek komplexitása

Fontos kiemelni, hogy a formális verifikáció egy komplex, nagy idő- és memóriaigényű feladat, ezért szükség van a magasszintű, bonyolult mérnöki modellekből transzformációkon keresztül viszonylag egyszerű, implementációs részleteket elhagyó formális modelleket származtatni, melyeknek komplexitása a modellellenőrző eszközök által kezelhető.

A BridgePoint-ban elkészített állapotgépek és ezekből az általunk megalkotott, a 3.3 fejezetben ábrázolt transzformációs folyamat által előállított UPPAAL automaták állapotainak számát a 4.3 ábra mutatja. A transzformáció során létrejött automaták közül néhány a Függelékek között, az F.4 fejezetben megtekinthető.

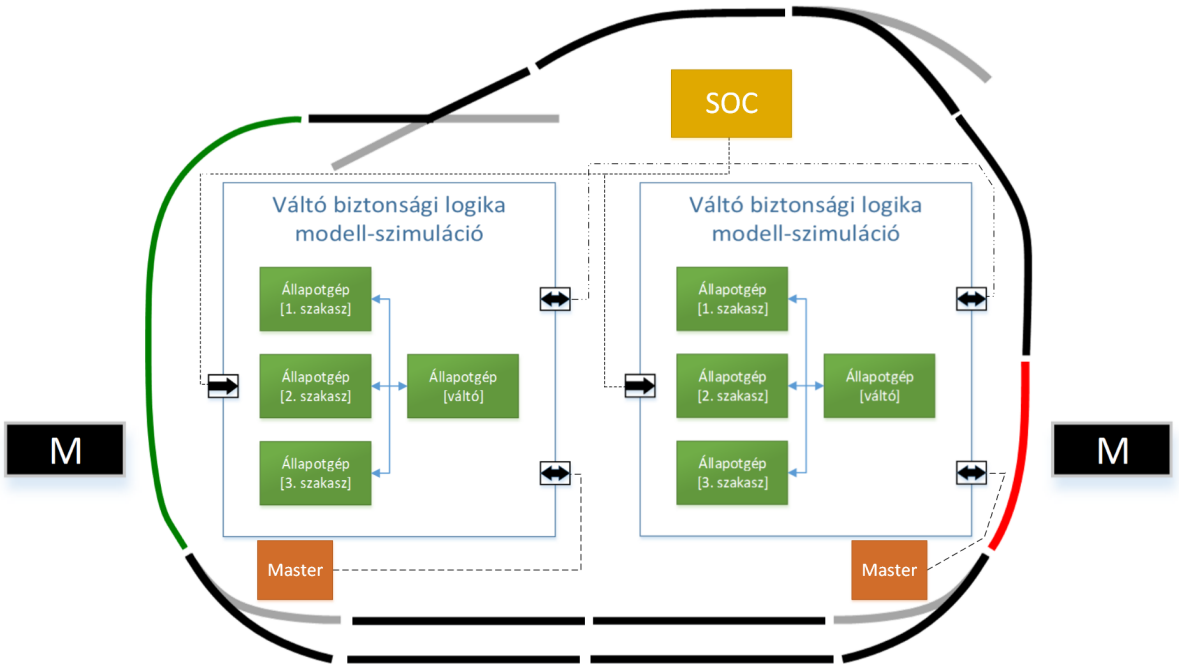
Állapotgép	Állapotok/helyek száma	
	BridgePoint	UPPAAL
FreeSection	3	6
OccupiedSection	5	13
StraightTurnout	5	14
DivergentTurnout	5	14

4.3. táblázat. BridgePoint állapotok és UPPAAL helyek száma a generálást követően

Látható, hogy a transzformáció során jóval több UPPAAL hely keletkezik, mint ahány állapot van az eredeti állapotgépben, azaz egy kis méretű mérnöki modellből is egy komplex formális modell áll elő. Továbbá, a keletkezett automaták kompozíciója esetén, ami jellemző az elosztott rendszerek modellezésére, könnyen szembesülhetünk az *állapottér robbanás* problémájával is.

Az általunk elkészített teljes biztonsági logika formális ellenőrzését három fázisra bontottuk:

1. fázis: lokális döntések verifikációja egy váltó esetén
2. fázis: globális döntések verifikációja két váltó esetén
3. fázis: globális döntések verifikációja több mint két váltó esetén



4.21. ábra. Elosztott modell alapú biztonsági logika szimulációja

Dolgozatunkban a biztonsági logika ellenőrzésének első fázisát, egy váltó lokális döntésének verifikációját végeztük el a 4.5.1 fejezetben leírtak szerint.

4.5.1. Lokális döntésének verifikációja

A BridgePoint modellekből származtatott UPPAAL automatákban minden szinkronizáció *broadcast* csatornán keresztül történik. Ennek következménye, hogy egy szinkronizációs csatorna *küldő* oldala akkor is elvégzi a csatornán a szinkronizációs jelzés átküldését, ha a *fogadó* oldalon senki nem vár erre. Ez szemantikailag azonos a BridgePoint állapotgépek szimulációja során az adott állapotgép által, adott állapotban eldobott események kezelésével.

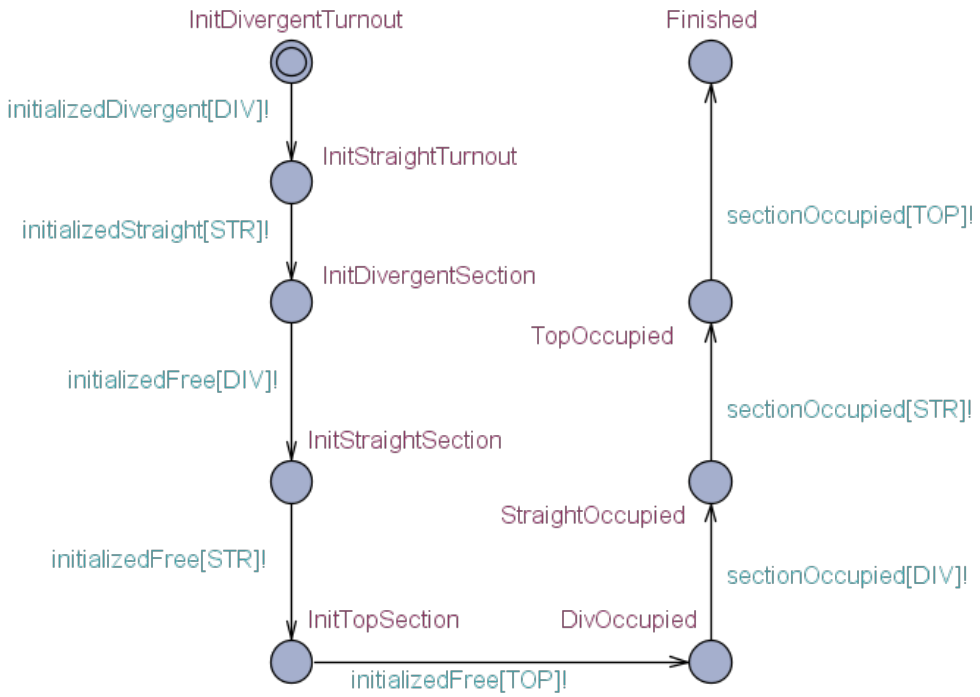
A biztonsági logika modellezésekor a modell részére szükséges külső információknak (pl. szakaszok foglaltsága) nem létezik BridgePoint modellje. Emiatt van szükség a *környezeti modellnek* megfelelő automatára az UPPAAL-ban, amely biztosítja az automaták működéséhez szükséges információkat valamint ezzel a verifikáció végrehajtása valósághű lesz.

A 4.5 szakaszban említett első fázis ellenőrzéséhez UPPAAL-ban, a 4.22 ábrán látható környezeti modellt készítettük el. Az automata viselkedést tekintve először inicializálja a váltóállást[3] és a váltórészeket, majd a kitérő irányból csatlakozó, az egyenes irányból csatlakozó és a váltócsúcs felől csatlakozó szakaszoknak (*Section*) küld egy-egy *foglalt* jelzést.

Egy váltó lokális döntésének verifikációhoz az alábbi öt esetet különböztettünk meg, melyekhez tartozó követelményeket formális, UPPAAL kifejezésekként fogalmaztuk meg.

1. Kitérő váltóállás (DIV), foglalt kitérő ág (DIV) és foglalt váltócsúcs (TOP) esetén a környezeti modell lefutása során a kitérő ághoz és a váltócsúcshoz tartozó szakasz (*Section*) előbb-utóbb zárolva lesz.

[3] A két váltóállás közül mindig csak az adott esetnek megfelelő lesz inicializálva.



4.22. ábra. Környezeti modell automatája

Ez a valóságban annak a veszélyes szituációnak felel meg, amikor a kitérő- és a váltócsúcs ágon is van egy-egy mozdony, amik között már csak egy szabad pályarész (váltó) van, ezért meg kell állítani mindkét vonatot.
Formálisan:

$$Environment.Finished \rightarrow OccupiedSectionDiv.BecomesLocked \\ \wedge\, OccupiedSectionTop.BecomesLocked$$

2. Egyenes váltóállás (STR), foglalt egyenes ág (STR) és foglalt váltócsúcs (TOP) esetén a környezeti modell lefutása során a kitérőághoz és a váltócsúcshoz tartozó *Section* előbb-utóbb zárolva lesz.
   Ez a valóságban annak a veszélyes szituációnak felel meg, amikor az egyenes- és a váltócsúcs ágon is van egy-egy mozdony, amik között már csak egy szabad pályarész (váltó) van, ezért meg kell állítani mindkét vonatot.
   Formálisan:

$$Environment.Finished \rightarrow OccupiedSectionStr.BecomesLocked \\ \wedge\, OccupiedSectionTop.BecomesLocked$$

3. Egyenes váltóállás (STR) és foglalt kitérő ág (DIV) esetén a kitérő ághoz tartozó *Section* előbb-utóbb zárolva lesz.
   Ez a valóságban annak a veszélyes szituációnak felel meg, amikor egy mozdony a váltót annak állásával ellentétes irányból közelíti meg. Emiatt a vonatot meg kell állítani, külön-

ben szabálytalanul menne rá a váltóra.
Formálisan:

$$Environment.Finished \rightarrow OccupiedSectionDiv.BecomesLocked$$

4. Kitérő váltóállás (DIV), foglalt egyenes ág (STR), foglalt kitérő ág (DIV) és foglalt váltócsúcs (TOP) esetén a környezeti modell lefutása során mindhárom váltórészhez tartozó *Section* előbb-utóbb zárolva lesz.
   Ez a valóságban annak a veszélyes szituációnak felel meg, amikor a váltó mindhárom irányából egy-egy mozdony közeledik és már csak egy szabad pályarész (váltó) van a mozdonyok között, ezért meg kell állítani mindhárom vonatot.
   Formálisan:

$$\begin{aligned} Environment.Finished \rightarrow & \; OccupiedSectionStr.BecomesLocked \\ & \wedge OccupiedSectionDiv.BecomesLocked \\ & \wedge OccupiedSectionTop.BecomesLocked \end{aligned}$$

5. Egyenes váltóállás (STR) és foglalt váltócsúcs (TOP) esetén a váltócsúcshoz tartozó *Section* soha nem lesz zárolva.
   A valóságban ilyenkor nem kell a biztonsági logikának beavatkoznia, a vonat szabadon mehet tovább.
   Formálisan:

$$A <> \; not\, OccupiedTop.BecomesLocked$$

A verifikáció eredményeket a 4.4 táblázat foglalja össze, melyben feltüntettük az egyes esetek verifikációja során használt memóriát és a verifikáció futásának időtartamát is.

Eset	**Memória**	**Idő**	**Teljesül**
1	10 928 KB	10 ms	✓
2	9 488 KB	16 ms	✓
3	10 120 KB	15 ms	✓
4	12 254 KB	20 ms	✓
5	10 660 KB	7 ms	✓

4.4. táblázat. Verifikációs eredmények lokális döntésre

A felírt kifejezések teljesülnek a formális modellen, azaz a modellellenőrző nem talál ellenpéldát. Ez alapján kijelenthetjük, hogy a váltók lokális döntései helyes működéshez vezetnek a vizsgált helyzetekben.

A váltók globális döntéseinek verifikációja az előzőekhez képest a kibővített, elosztott modell szerinti új állapotgépek és ezek közötti kommunikáció leképzését valamint bonyolultabb környezeti modellek megalkotását igényli. Ezek megvalósítása szerepel az 5. fejezetben bemutatandó jövőbeli terveink között.

5. fejezet

Összefoglalás és jövőbeli tervek

5.1. Elért eredmények

- *Mérnöki modellek formális ellenőrzése*: A TDK dolgozatunk során sikerült egy iparilag releváns modellező alkalmazás modelljeit, illetve Alf modelleket formális modellekre transzformáltunk automatikusan.
- *Komplex esettanulmány*: Elkészítettük a terepasztalt, melynek teljes hardver-szoftver részét sajátkezűleg valósítottuk meg, és a már meglévő vezérlési rendszerhez integráltuk ezt az architektúrát.
  - *Eszközök integrációja*: összekapcsoltuk a számítógépes rendszereinket a beágyazott rendszerekkel, így azok egy komplex visszajelzési és biztonsági rendszert alkotnak;
  - *Szükséges hardverek építése*: a projekthez szükséges hiányzó hardvereket elkészítettük és integráltuk a már meglévő rendszerünkbe;
  - *Biztonsági logika*: elkészítettük modell alapokon azt a biztonsági logikát, amely felügyeli a mozdonyok mozgását, folyamatos teszteléssel és verifikációval.
- *BridgePoint kódgenerátor alkalmazása*: Megismertük és használtuk a BridgePoint által adott generátort, konfiguráltuk azt a forráskód beágyazott rendszerekre való telepítéshez.
- *Modellellenőrzés*: Sikeresen verifikáltuk a már elkészült modelljeinket UPPAAL-lal, így a korai hibáinkat ki tudtuk javítani.
- *BME Kütyüpályázat*: Bemutattuk a terepasztalt a Villamosmérnöki és Informatikai Kar MIT tanszéke által szervezett versenyen, ahol már a hardver figyelte a rendszer állapotait és avatkozott be veszély esetén.
- *Kutatók éjszakája*: Az Ericsson Magyarország Kft. közreműködésével bemutattuk az érdeklődőknek, a modell alapú fejlesztés együttműködését a terepasztallal. Az eseményről készült Index videó [31] nagy része is ezt az asztalt mutatja be.

5.2. Problémák a fejlesztés során

A projekt kapcsán több érdekes probléma is felmerült, melyek közül többet is sikerült megoldanunk:

- *S88 kommunikáció visszafejtése*: Habár a kommunikációs szabvány dokumentációja elérhető, a gyártók gyakran nem teszik közzé a megvalósítás dokumentációját, így az ahhoz csatlakozó modul implementálása a kommunikáció megvalósításának visszafejtésével kezdődött. A fejlesztővel történő konzultáció során kapott segítséggel megoldottuk az időzítési problémákat.
- *Generált kód telepítése*: A BridgePoint modellekből generált C forráskódokat az Arduinokra telepítés során azt tapasztaltuk, hogy a program memóriahasználata túlhaladja az eszköz operatív memóriáját, így a futtatás nem volt megvalósítható. Ennek megoldása még további kontribúciót igényel.
- *BridgePoint rendszer hibái*: A BridgePoint rendszerben talált hibák gyakran olyan problémák voltak, amelyekre a hivatalos támogatás sem tudott megoldást nyújtani, valamint a tesztelések során a BridgePoint szimulációs környezetének terhelhetőségi problémái is előkerültek.
- *Hiányos dokumentáció*: A használt Arduino eszköz dokumentációja erősen hiányos, így félreértelmezett adattal számoltunk a hozzáillesztett eszközök áramfelvételét illetően. Kiegészítő áramkörökkel eszközöltük a problémát.

5.3. Tervek

A projekt folytatásaként az alábbi célokat tűztük ki:

- *Kódgenerátor további konfigurációja*: Szeretnénk finomítani a már meglévő konfigurációnkat a kódgenerátorhoz és megoldani a modellből való forráskód-származtatást.
- *Teljes verifikáció*: Szükséges a teljes rendszert ellenőrizni ahhoz, hogy kijelenthessük a hibamentességet, jelenleg csak egyes moduljainkat verifikáltuk.
- *Kommunikációk ellenőrzése*: A jelenleg használt kommunikációs protokollok formális ellenőrzésével felderíthetjük, hogy azokban hol lehetnek hibák, így ezek biztosításával tovább stabilizáljuk a rendszert.
- *Transzformált elemek nyomonkövethetősége*: A megvalósított modelltranszformációk kiegészítése nyomonkövethetőséggel (*traceability*); mellyel meghatározható, hogy a létrejövő példánymodellben lévő elemek melyik, a kiindulási példánymodellben szereplő elemekből jöttek létre. Így a létrejövő modellen végzett formális verifikáció eredményei a kiindulási példánymodellre könnyebben visszavezethetőek.

Köszönetnyilvánítás

Szeretnénk megköszönni a segítségét Lazányi Jánosnak, Szántó Péternek, Fehér Béla tanár úrnak és Cseh Róbertnek, akik segítették a hardver elkészülését mind technikai, mind tudásbeli hozzájárulással, továbbá Darvas Ádám (Ericsson Magyarország Kft.) segítségét és hozzájárulását a szoftveres oldalon. A vasútmodellezéssel kapcsolatos kérdéseinket a Vasútmodell-Centrum Kft. és a DigiTools Elektronika Kft. munkatársai válaszolták meg, köszönjük kitartó támogatásukat.

Irodalomjegyzék

[1] Radio Technical Commission for Aeronautics (RTCA). *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*, 2014.

[2] Radio Technical Commission for Aeronautics (RTCA). *DO-333 Formal Methods Supplement to DO-178C and DO-278A*, 2014.

[3] Jean-Louis Boulanger. *The new CENELEC EN 50128 and the used of formal method*, 2014.

[4] Friedenthal, Sanford and Moore, Alan and Steiner, Rick. *A practical guide to SysML: the systems modeling language*. Elsevier, 2011.

[5] Crisp, HE. Systems Engineering Vision 2020. *Seattle, Washington*, 2007.

[6] Radio Technical Commission for Aeronautics (RTCA). *DO-278A Software Integrity Assurance Considerations for Communication, Navigation, Surveillance and Air Traffic Management (CNS/ATM) Systems*, 2014.

[7] Radio Technical Commission for Aeronautics (RTCA). *DO-331 Model-Based Development and Verification Supplement to DO-178C and DO-278A*, 2014.

[8] V tervezési modell, 2014. URL: `http://www.waterfall-model.com/v-model-waterfall-model/`.

[9] Object Management Group. *OMG Unified Modeling Language (OMG UML) - Version 2.4.1*, August 2011.

[10] Object Management Group. *OMG Systems Modeling Language (OMG SysML) - Version 1.3*, June 2012.

[11] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models (fUML) - Version 1.1*, August 2013.

[12] Christian Ansorge, Ivica Skender. *Introduction to Unified Modeling Langauge (UML)*, December 2012.

[13] Stefano Cucchiella. Horizontal transformations for models reuse and tool chaining. 2010.

[14] Object Management Group. *Action Language for Foundational UML (ALF) - Concrete Syntax for a UML Action Language (Version 1.0.1)*, October 2013.

[15] Shlaer, Sally. The Shlaer-Mellor method. *Project Technology White paper*, 1996.

[16] Mellor, Stephen J and Balcer, Marc and Foreword By-Jacoboson, Ivar. *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[17] Object Action Language Reference Manual. URL: `http://www.ooatool.com/docs/OAL08.pdf`.

[18] Object Management Group. *Model Driven Architecture Overview*, 2014.

[19] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on UPPAAL 4.0*, 2006. URL: `http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf`.

[20] Luca Aceto, Augusto Burgueno, and Kim G Larsen. *Model checking via reachability testing for timed automata*. Springer, 1998.

[21] Kim G Laxsen, Paul Pettersson, and Wang Yi. *Diagnostic model-checking for real-time systems*. Springer, 1996.

[22] Zoltán Micskei, Raimund-Andreas Konnerth, Oszkár Benedek Horváth, András Vörös Semeráth, and Dániel Varró. On Open Source Tools for Behavioral Modeling and Analysis with fUML and Alf. 2014.

[23] Kleppe, Anneke G. and Warmer, Jos and Bast, Wim. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[24] DCC protokoll, 2014. URL: `http://www.opendcc.de/info/dcc/dcc.html`.

[25] Arduino. Arduino Uno specifikáció weboldal, 2014. `http://arduino.cc/en/Main/ArduinoBoardUno`.

[26] Assymetric Break Control, 2014. URL: `http://www.tonystrains.com/technews/lenz-asy-abc.htm`.

[27] David P. Reed. Udp protokol, 1980. URL: `https://www.ietf.org/rfc/rfc768.txt`.

[28] IEEE 802.3 (Ethernet) szabvány, 1983. URL: `http://standards.ieee.org/about/get/802/802.3.html`.

[29] I2C hivatalos specifikáció, 2014. URL: `http://www.nxp.com/documents/user_manual/UM10204.pdf`.

[30] S88-N szabvány, 2014. URL: `http://www.s88-n.eu/index-en.html`.

[31] Fiantok Dániel, Pándi Balázs. Az okospapíré és az intelligens kosárlabdáé a jövő, 2014. URL: `http://index.hu/video/2014/09/26/ilyen_lesz_a_halozatba_kapcsolt_tarsadalom_jovoje/`.

Függelék

F.1. Alf példakód

```
testBlock{
 let input: Integer = 5;
 let var1: Integer = 1;
 let var2: Integer = 1;
 let tmp: Integer = 0;

 if (input == 1){
        return 1;
 } else if(input == 2) {
        return 1;
 }

 let i: Integer = 3;

 while (i <= input){
        tmp = var1 + var2;
        var1 = var2;
        var2 = tmp;
        i++;
 }

 return tmp;
}
```

F.1.1. Fibonacci-sorozat implementáció

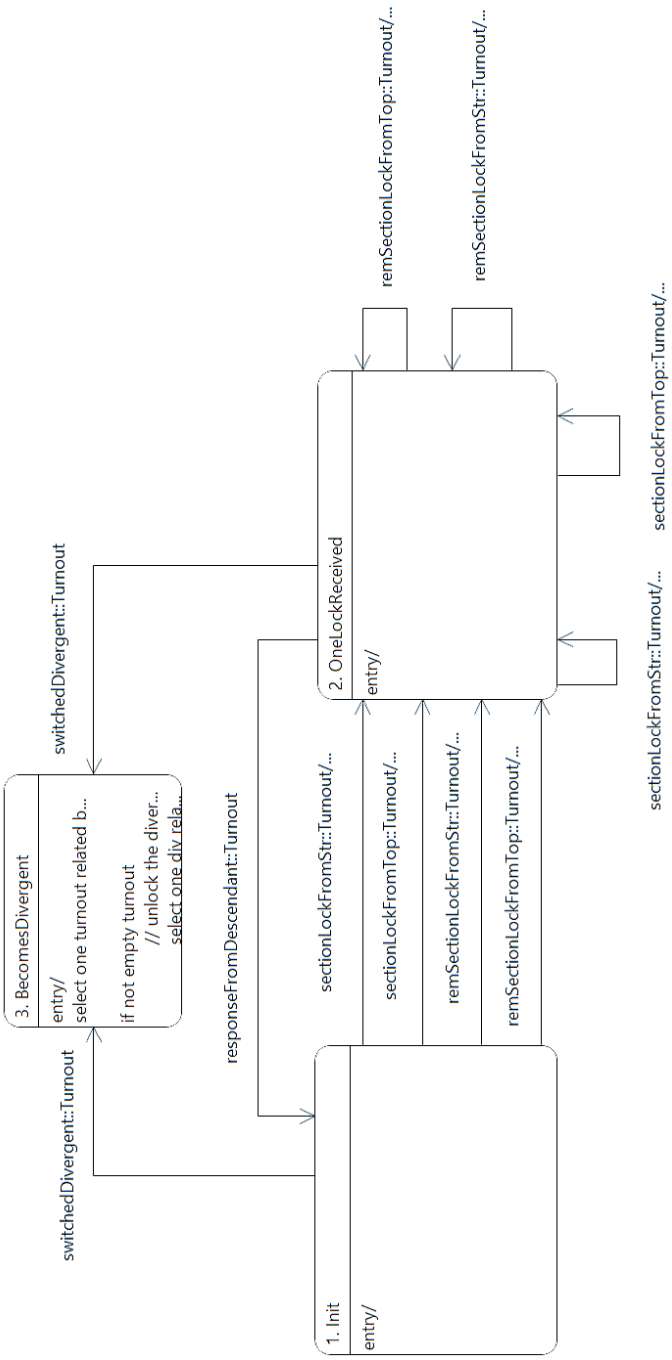
F.2. Modellek példányosítása

```
1  // initialize sections
2  create object instance straight of Section;
3  straight.id = 9;
4  create object instance fStraight of FreeSection;
5  relate fStraight to straight across R1;
6  create object instance divergent of Section;
7  divergent.id = 13;
8  create object instance fDivergent of FreeSection;
9  relate fDivergent to divergent across R1;
10
11 // initialize turnout
12 create object instance turnout of Turnout;
13 turnout.id = 129;
14 turnout.index = 1;
15 relate turnout to divergent across R3.'divergent';
16 relate turnout to straight across R4.'straight';
17
18 // create the descendants of the turnout
19 create object instance ds of DispatchStraight;
20 create object instance hs of HandlerStraight;
21 create object instance ps of ProtocolStraight;
22 relate turnout to ds across R2;
23 relate ds to hs across R7;
24 relate hs to ps across R9;
25
26 // create remote turnouts
27 create object instance divRemote of RemoteTurnout;
28 divRemote.remoteId = 131;
29 divRemote.localDirection = Direction::Top;
30 divRemote.remoteDirection = Direction::Divergent;
31 divRemote.isPrior = false;
32 create object instance topRemote of RemoteTurnout;
33 topRemote.remoteId = 134;
34 topRemote.localDirection = Direction::Divergent;
35 topRemote.remoteDirection = Direction::Top;
36 topRemote.isPrior = true;
37 create object instance strRemote of RemoteTurnout;
38 strRemote.remoteId = 134;
39 strRemote.localDirection = Direction::Divergent;
40 strRemote.remoteDirection = Direction::Straight;
41 strRemote.isPrior = true;
42
43 relate turnout to divRemote across R6;
44 relate turnout to topRemote across R6;
45 relate turnout to strRemote across R6;
46
47 // generate the initializer events
48 generate Section5:sectionInitialized to fDivergent;
49 generate Section5:sectionInitialized to fStraight;
50 generate Turnout8:turnoutInitialized to turnout;
51 generate HandlerStraight31:HSinitialized to hs;
52
53 // initialize the realized component
54 send Port1::initialize(turnoutId: turnout.id);
```

F.2.1. Váltóhoz tartozó modellek példányosítása

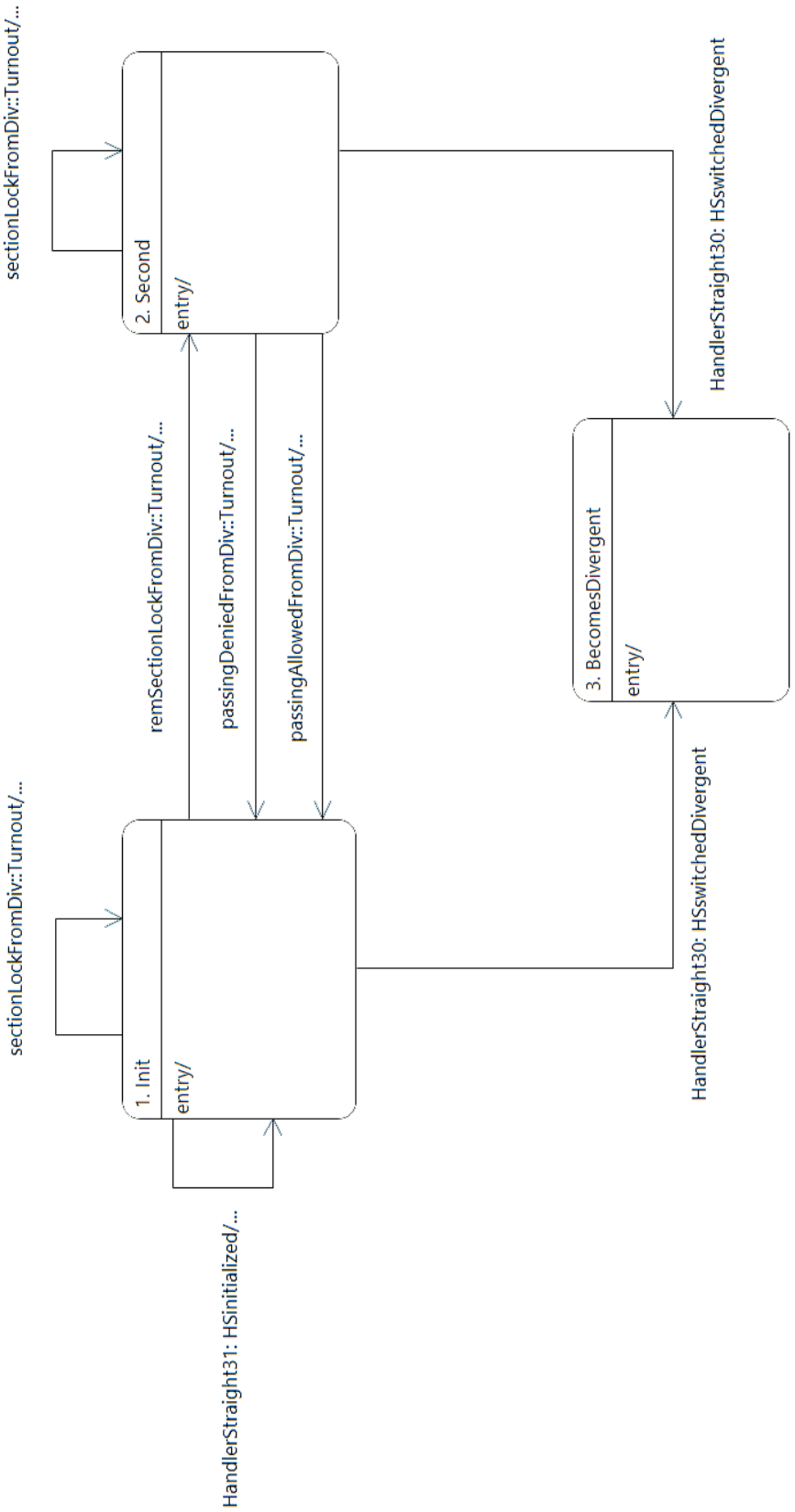
F.3. Állapotgép részletek

A *DispatchStraight* osztályhoz tartozó példányszintű állapotgép, melynek feladata a beérkező kérések transzformálása új, a hierarchia alsó szintjein lévő állapotgépek által feldolgozható jelzésekké, és egy kérés feldolgozása közben, a váltó állásával megegyező irányból érkező újabb kérések automatikus visszautasítása.



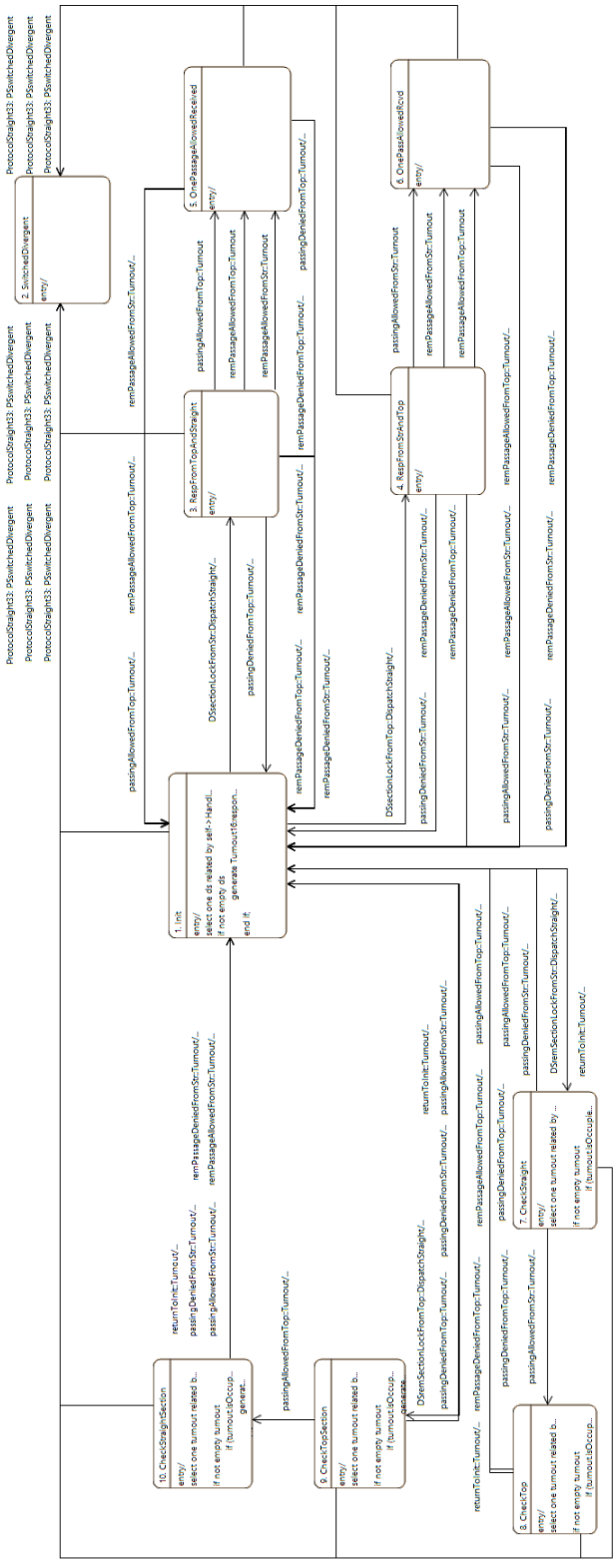
F.3.1. ábra. DispatchStraight állapotgép

A *HandlerStraight* osztályhoz tartozó példányszintű állapotgép, melynek feladata a váltó állásával ellentétes irányból érkező kérések feldolgozása.



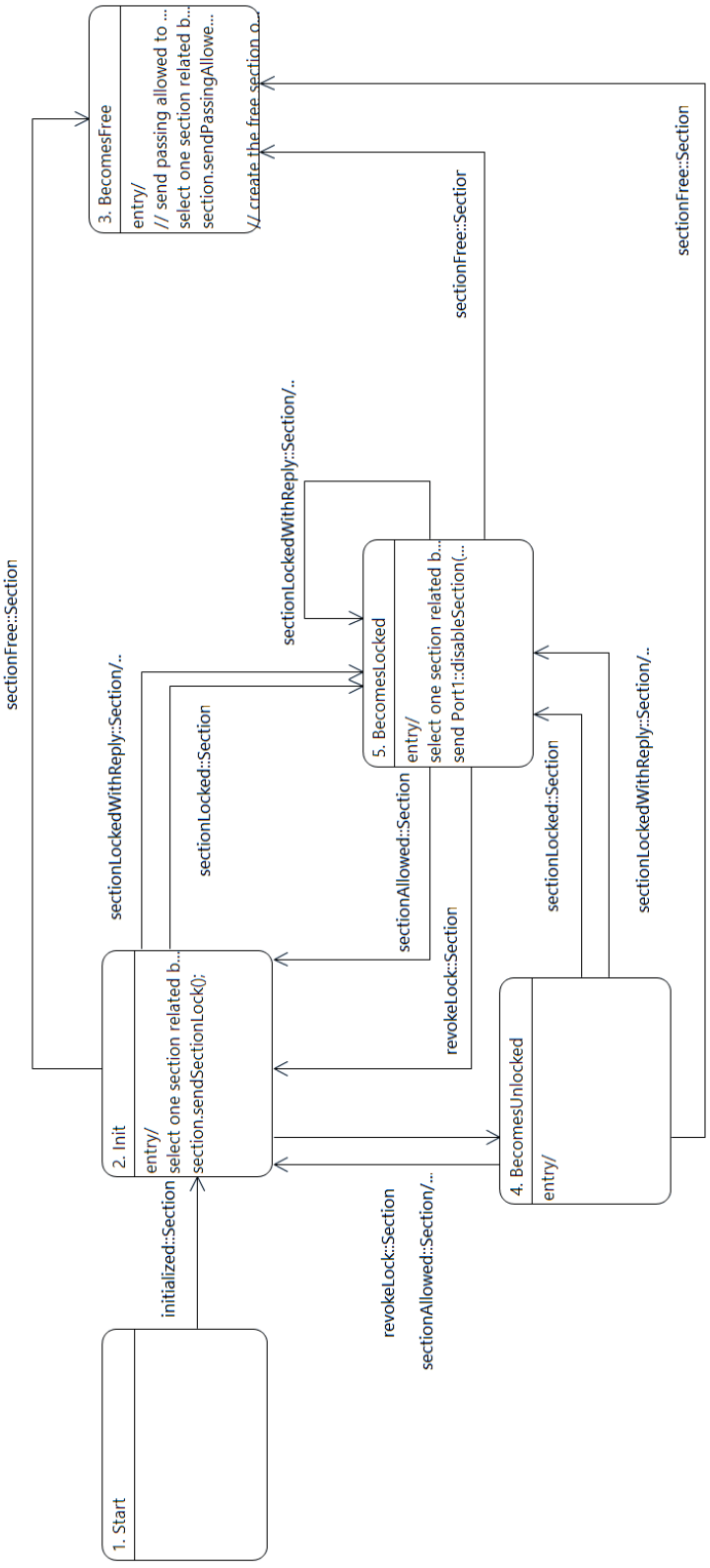
F.3.2. ábra. HandlerStraight állapotgép

A *ProtocolStraight* osztályhoz tartozó példányszintű állapotgép, melynek feladata az adott váltóhoz tartozó lokális döntés, és a szomszédos váltókkal közös globális döntés meghozása, és végrehajtása.



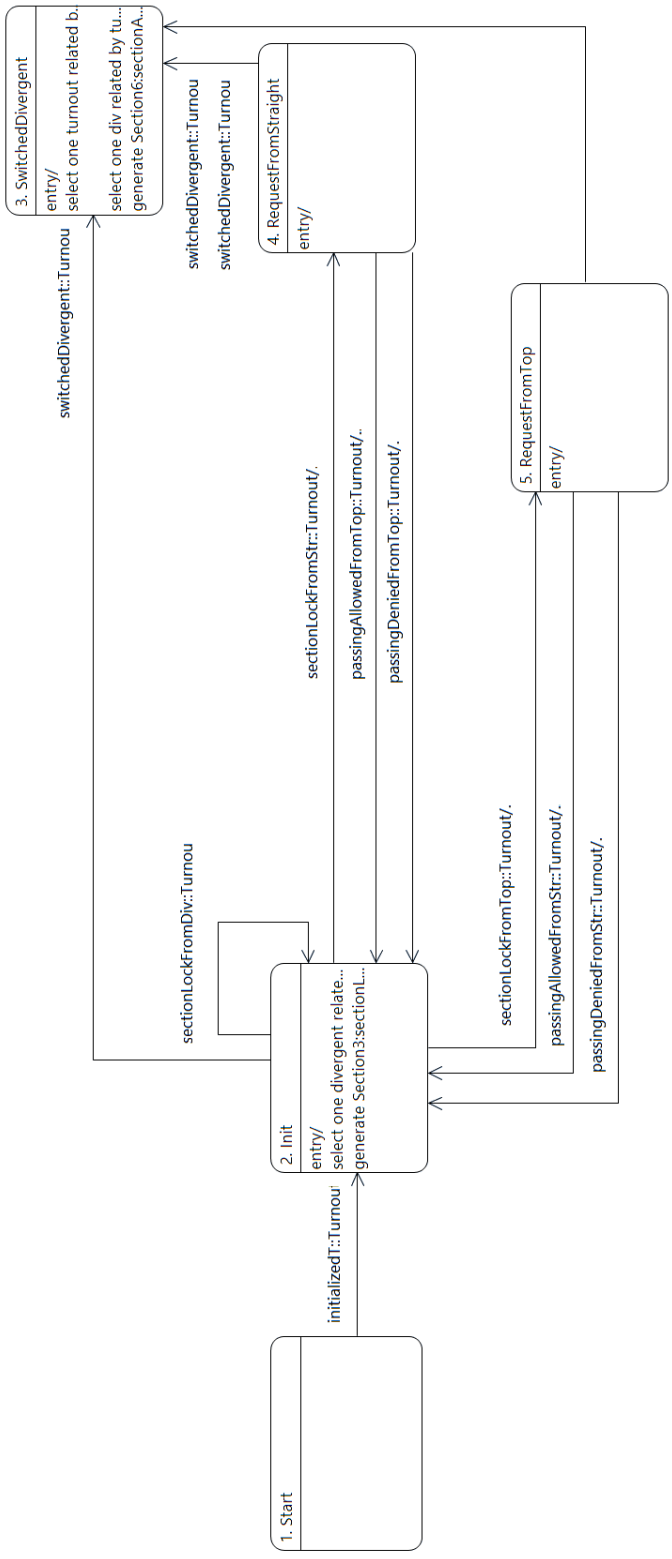
F.3.3. ábra. ProtocolStraight állapotgép

Az *OccupiedSection* osztályhoz tartozó példányszintű állapotgép, melynek feladata a foglalt szakaszhoz tartozó viselkedés megvalósítása a *szakasz lezárási protokoll* alapján.



F.3.4. ábra. Foglalt szakasz állapotgép

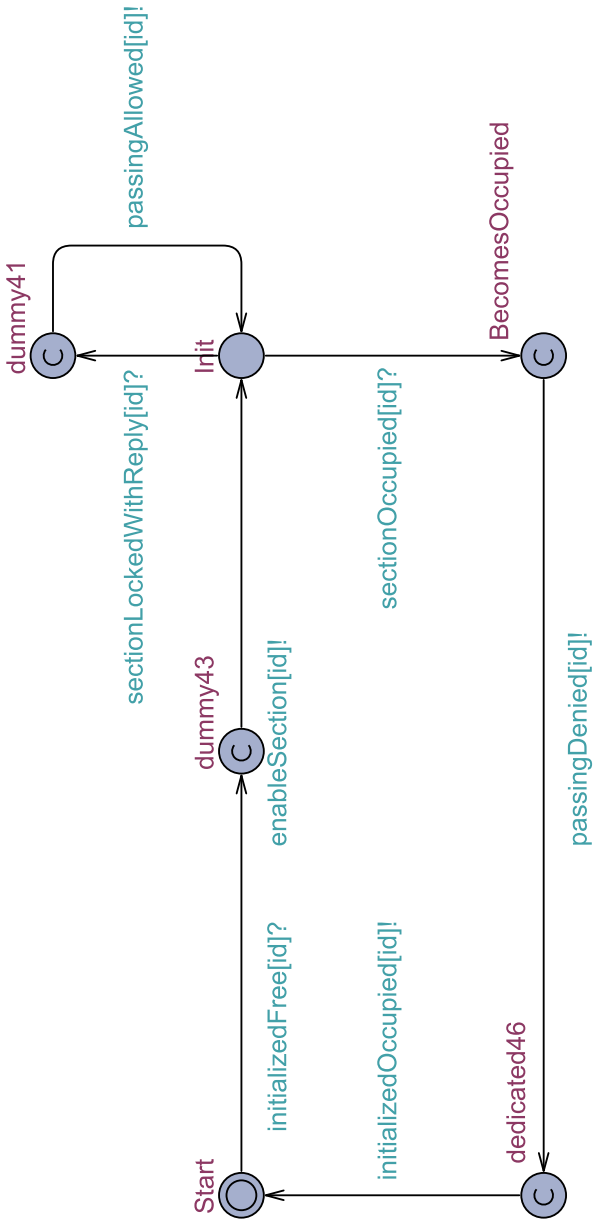
A *StraightTurnout* osztályhoz tartozó példányszintű állapotgép, melynek feladata a hagyományos váltóhoz tartozó, lokális döntést tartalmazó viselkedés megvalósítása a *szakasz lezárási protokoll* alapján.



F.3.5. ábra. Egyenes állású váltó állapotgép

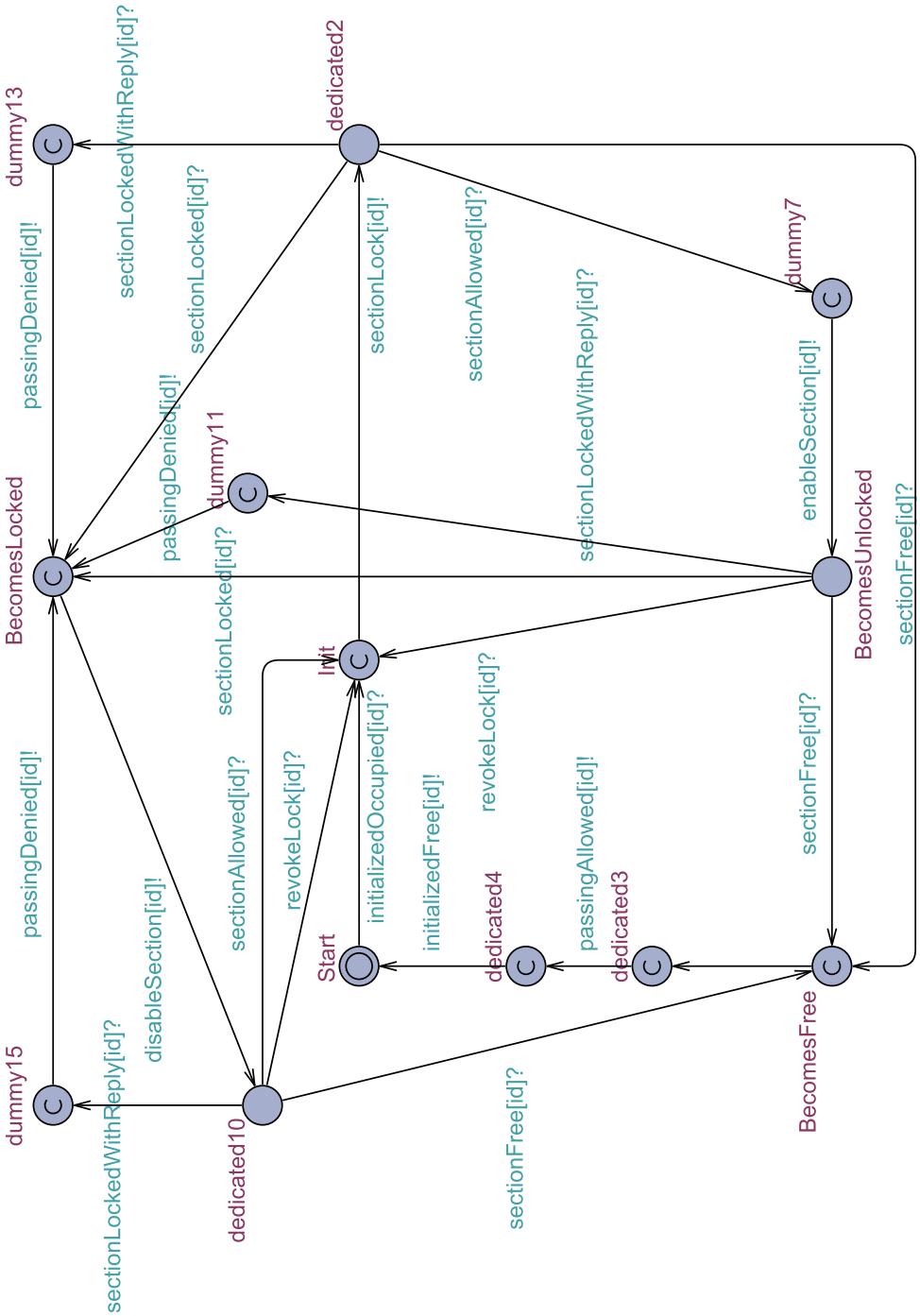
F.4. UPPAAL automata részletek

A *FreeSection* állapotgéphez tartozó formális automata, melynek feladata a szabad szakaszhoz tartozó viselkedés megvalósítása a *szakasz lezárási protokoll* alapján.



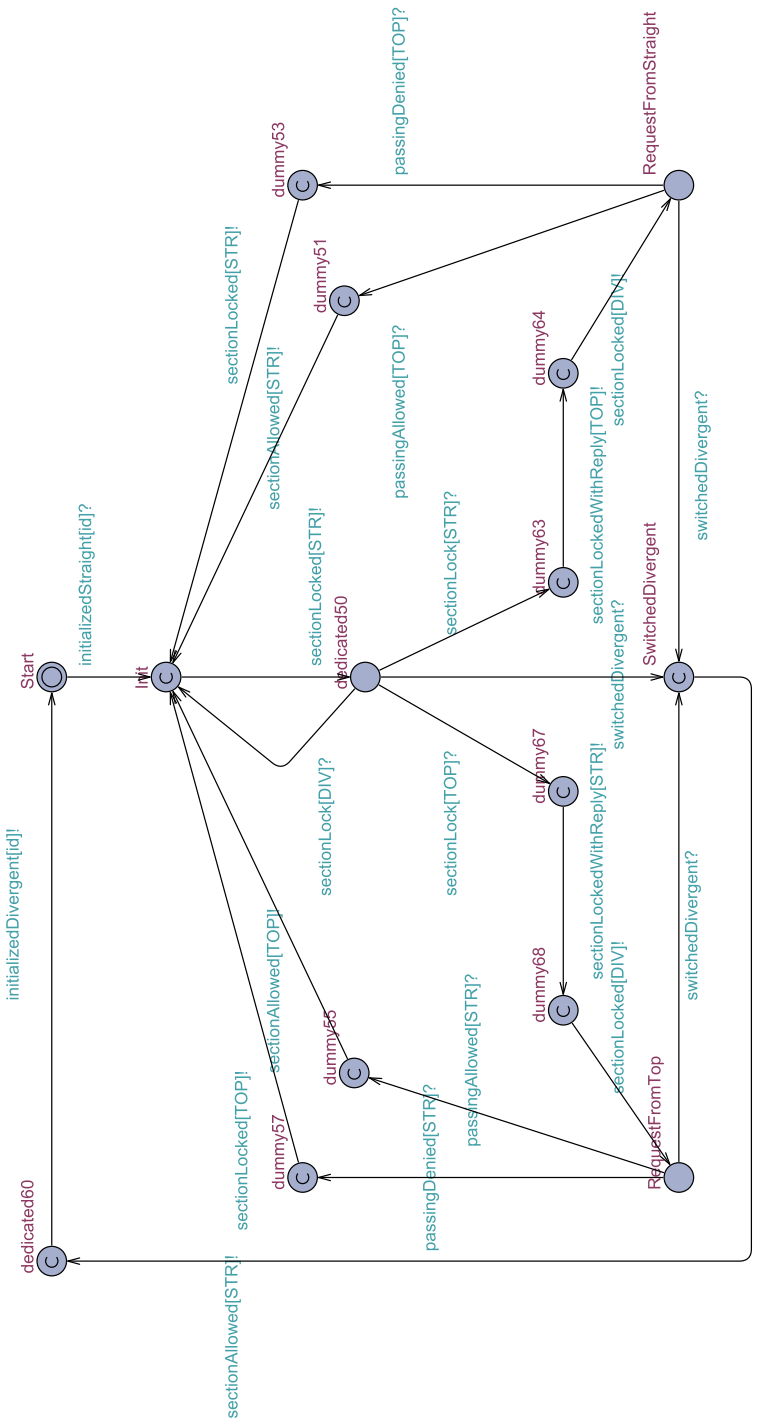
F.4.1. ábra. Szabad szakasz automata

Az *OccupiedSection* állapotgéphez tartozó formális automata, melynek feladata a foglalt szakaszhoz tartozó viselkedés megvalósítása a *szakasz lezárási protokoll* alapján.



F.4.2. ábra. Foglalt szakasz automata

A *StraightTurnout* állapotgéphez tartozó automata, melynek feladata az egyenes állású váltó viselkedésének megvalósítása.



F.4.3. ábra. Egyenes állású váltó automata