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

Elosztott biztonságkritikus rendszerek xtUML alapú modellvezérelt fejlesztése

Szakdolgozat

Készítette
Horváth Benedek

Konzulens
Vörös András

2014. december 11.

Tartalomjegyzék

1. Bevezetés **7**

2. Háttérismeretek **8**

2.1. Modell alapú rendszertervezés 8

2.2. Modell alapú rendszertervezés folyamata 9

2.3. Szoftverrendszerek modell alapú fejlesztése 10

2.3.1. Shlaer-Mellor módszer 11

2.3.2. Executable UML 12

2.3.3. Executable and Translatable UML 13

2.4. Mérnöki modellek tervezése BridgePoint-ban 13

2.4.1. Iparban elterjedt modell alapú fejlesztőeszközök 15

2.5. Verifikáció 15

2.5.1. Véges automata formalizmus 16

2.5.2. Automaták hálózata 16

2.5.3. Az UPPAAL modellellenőrző eszköz 17

2.5.4. Véges automaták UPPAAL-ban 17

2.5.5. Időzítések UPPAAL-ban 17

2.5.6. Szinkronizáció UPPAAL-ban 17

2.5.7. Követelmények specifikációja UPPAAL-ban 18

2.5.8. Elérhetőségi követelmények 19

2.5.9. Biztonsági követelmények 19

2.5.10. Élőségi követelmények 19

2.5.11. Állapotkifejezések 19

3. xtUML elemek **20**

3.1. Követelmény analízis 20

3.1.1. Használati eset diagram 21

3.1.2. Aktivitás diagram 21

3.1.3. Szekvencia diagram 22

3.2. Rendszerterv modell 22

3.3. Komponens modellek 22

3.4. Komponenseket felépítő egységek modelljei 23

3.5. Modellek szimulációja 25

3.6. Modellekből forráskód származtatása . . . 25
3.7. xtUML alapú tervezés értékelése . . . 26
3.7.1. xtUML előnyei . . . 26
3.7.2. xtUML hátrányai . . . 26

4. Demonstrátor eszköz biztonsági logikájának modell alapú tervezése 27
4.1. Esettanulmány rövid bemutatása . . . 27
4.2. Konfiguráció a BridgePoint-tal való együttműködéshez . . . 28
4.3. Biztonsági logika modell alapú tervezése . . . 29
4.3.1. Magasszintű követelmények . . . 29
4.3.2. Specifikáció . . . 30
4.3.3. Tervezési feltételezések és kényszerek . . . 30
4.3.4. Szakasz lezárási protokoll . . . 31
4.3.5. Szakasz engedélyezési protokoll . . . 34
4.3.6. Terminológia definiálása . . . 35
4.3.7. Biztonsági logika modell alapú tervezése . . . 36
4.3.8. Váltóhoz tartozó biztonsági logika modellszintű kompozíciója . . . 41
4.3.9. Biztonsági logika szimulációja . . . 42
4.4. Rendszerszintű modell értékelése . . . 42

5. Modell alapú szoftverfejlesztés támogatása formális ellenőrzésekkel 44
5.1. Modellek transzformációja . . . 44
5.2. Mérnöki modellek formális verifikációja . . . 45

6. Biztonsági logika verifikációja 51
6.1. Formális verifikáció folyamata . . . 51
6.2. Biztonsági logika állapotgépeinek formális verifikációja . . . 51
6.2.1. Lokális döntések verifikációja . . . 52

7. Összefoglalás és jövőbeli tervek 55
7.1. Gyakorlati eredmények . . . 55
7.2. Elméleti eredmények . . . 55
7.3. Jövőbeli tervek . . . 56

Ábrajegyzék 58

Irodalomjegyzék 60

Függelék 61
F.1. Modellek példányosítása . . . 61
F.2. Állapotgép részletek . . . 62
F.3. UPPAAL automata részletek . . . 67

SZAKDOLGOZAT-FELADAT

Horváth Benedek (CEGJKA)
szigorló mérnök informatikus hallgató részére

Elosztott biztonságkritikus rendszerek xtUML alapú modellvezérelt fejlesztése

Napjainkban az elosztott rendszerek egyre nagyobb teret hódítanak beágyazott bizonságkritikus környezetben is. Ez annak köszönhető, hogy az összetett rendszerek elvárt funkcionalitásának teljesítéséhez, a bonyolultságukból adódóan, elosztott működés szükséges. Ezen rendszerek tervezése komoly eszköztámogatást igényel, fontos feladattá vált a modellezés és a tervek ellenőrzése.

A mérnöki modelleket napjainkban sokszor UML alapú szerkesztő eszközökben tervezik meg, míg az ellenőrzést jellemzően modellellenőrző eszközök végzik formális modelleken.

Az xtUML és a hozzá tartozó eszközkészlet is az UML kiterjesztéseként fogható fel, hatékony támogatást adva a modell alapú tervezéshez.

A hallgató feladata megvizsgálni és bemutatni az xtUML modellezési nyelvet és egy elosztott vasút-irányítási biztonságkritikus demonstrátor esettanulmányon keresztül bemutatni alkalmazhatóságát.

A hallgató feladatának a következőkre kell kiterjednie:

- Tekintse át és mutassa be az xtUML nyelvet.
- A tanszéki demonstrátor szoftver komponenseit tervezze meg xtUML segítségével.
- Értékelje az xtUML alapú tervezést és vizsgálja meg ellenőrizhetőség szempontjából a rendszerterveket.

Tanszéki konzulens: Vörös András, tudományos segédmunkatárs

Budapest, 2014. október 11.

.....
Dr. Jobbágy Ákos
tanszékvezető

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

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

HALLGATÓI NYILATKOZAT

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

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

Budapest, 2014. december 11.

Horváth Benedek
hallgató

Kivonat

Az elmúlt évek során az informatikai és a biztonságkritikus rendszerek komplexitása rohamosan növekedett: például egy Airbus A380-as repülőgépen is több millió sornyi kód felelős az utasok biztonságáért. Az összetett rendszerek elvárt funkcionalitásának teljesítéséhez, a bonyolultságukból adódóan, elosztott működés szükséges, melynek megtervezése a hagyományos mérnöki módszerekkel csak nehezen lehetséges.

A problémának egy lehetséges megoldásaként az elmúlt években meghatározó paradigmává vált a modell alapú megközelítés. A metodika célja, hogy a rendszer elkészítése során már a korai fázisoktól, magasszintű modellekből kiindulva származtassuk a konfigurációt, dokumentációt és forráskódot is. Az előálló megoldások analízise és helyes működésének tesztelése a konvencionális módszerekkel jellemzően csak nagy erőforrás-ráfordítással lehetséges. Azonban a modellvezérelt megközelítés közvetlenül lehetőséget nyújt formális módszerek alkalmazására is, melyekkel a hibák a fejlesztés korai fázisában felfedezhetőek.

Dolgozatomban áttekintem a modellvezérelt rendszerfejlesztés módszertanát, ezen belül kiemelve az állapotgép alapú megközelítést támogató xtUML nyelvet. A nyelv alkalmazhatóságát egy elosztott vasút-irányítási biztonságkritikus demonstrátor esettanulmányon keresztül igazolom, melyben a biztonsági logikát modell alapon terveztem meg. Az elkészített modell helyességének matematikai precizitású bizonyításához formális modelleket származtattam, melyeken temporális logikai nyelven fogalmaztam meg követelményeket. A követelmények teljesülését egy iparilag releváns formális modellellenőrző keretrendszerben (UPPAAL) vizsgáltam.

Abstract

The complexity of IT and safety-critical systems has increased rapidly in recent years. In the field of software engineering, this led to the increasing number of lines of source code. For example, on an Airbus A380 airplane several million lines of code are responsible for the safety of the passengers. In addition, the rich functional requirements necessitates the development of distributed systems to be able to handle complexitiy and provide efficiency.

Therefore in the latest years the model-based approach became an important paradigm in the field of distributed and safety-critical systems. The goal of this approach is to provide a means to develop models and automatize the the generation of the implementation from the high level models.

Testing and analyzing the correct behavior of such systems with the conventional approaches usually requires big amount of manual effort. However, the model-driven approach can also provide means to apply formal methods that allows the discovery of both design and behavioral errors in an early stage of development.

In my thesis I overview the model based systems engineering, focusing on the xtUML language that provides a state-machine based approach. The usability of the language is demonstrated with a distributed railway-control safety-critical case study. The design models are transformed into formal models, so the correctness of the models are verified using temporal logic expressions by a widely used formal model checker framework called UPPAAL.

1. fejezet

Bevezetés

A biztonságkritikus rendszerek komplexitása az elmúlt évek során rohamosan növekedett. 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.

Ennek következtében számos modellezési nyelv terjedt el, köztük a végrehajtható és futtatható modellek tervezésére használt xtUML (*Executable and Translatable UML* [4]).

Szakdolgozatomban bemutatom az xtUML modellezési nyelvet és egy elosztott vasút-irányítási biztonságkritikus demonstrátor esettanulmányon keresztül igazolom alkalmazhatóságát. Dolgozatomban kitérek az xtUML nyelv fejlődési folyamatára, valamint a nyelv használatával elkészített mérnöki modellt formális ellenőrzés szempontjából megvizsgálom.

A 2. fejezetben ismertetem a modell alapú rendszer- és szoftvertervezés módszertanát, illetve a formális verifikációhoz szükséges háttérismereteket. A 3. fejezetben bemutatom az xtUML nyelvet és módszertant, melynek áttekintem főbb elemeit, majd értékelem az xtUML alapú tervezést. A 4. fejezetben bemutatom az általam modell alapon megtervezett tanszéki demonstrátor szoftver komponenseit, valamint értékelem az xtUML alapú rendszerterveket.

Az 5. fejezet az xtUML alapú modellek formális modellekre történő transzformációjának folyamatát tartalmazza. A létrejövő formális modellek ellenőrzését a 6. fejezet mutatja be.

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. A technológiai folyamatok magukban foglalják a rendszer specifikációját, megtervezését, elkészítését és az elkészült rendszer tesztelését, ellenőrzését, illetve verifikációját is [5].

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.*

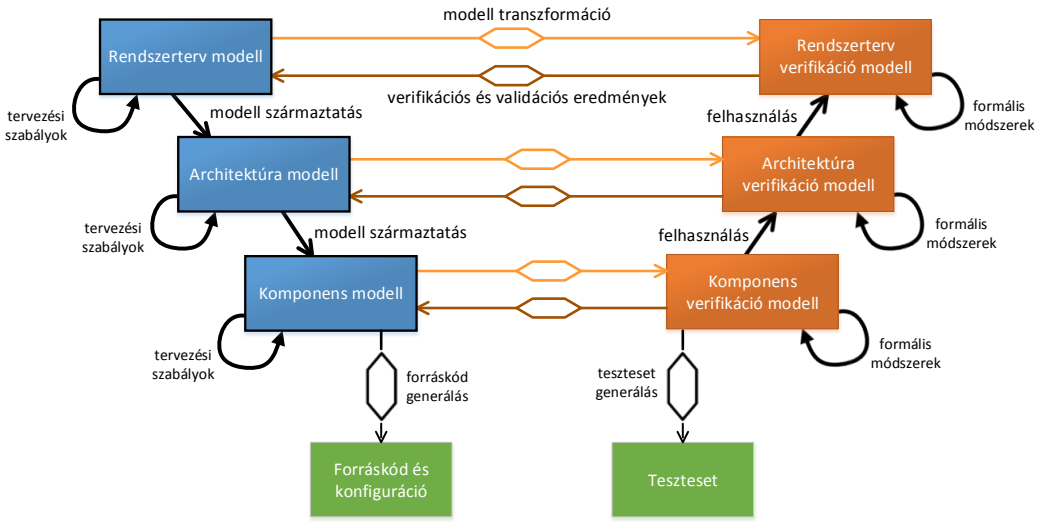
2. Definíció (Modell alapú rendszertervezés). *A modell alapú rendszertervezés egy formalizált alkalmazása a modellezésnek a rendszer követelményekre, tervezésre, analízisre, verifikációs és validációs tevékenységekre vonatkozóan. Az elsőszámú artifaktum a modell, amely a fejlesztés minden fázisában a legfőbb információ hordozó médium [6].*

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 biztosíthatóvá és bizonyíthatóvá válik.

A korábbi megközelítésben alkalmazott nagymennyiségű dokumentáció mellett 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, automatikusan megtörténik. Ezáltal nem lesz ellentmondás a szöveges dokumentumok és a rendszer



2.1. ábra. Modell alapú rendszertervezés Y-modell

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 [7], DO-331 [8]) 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 megfelelés ellenőrzése), a tervezés minden fázisában. Ezért a rendszertervezés folyamata során tipikusan a szoftver- és rendszerfejlesztésben elterjedt V-modellnek [9] 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 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.

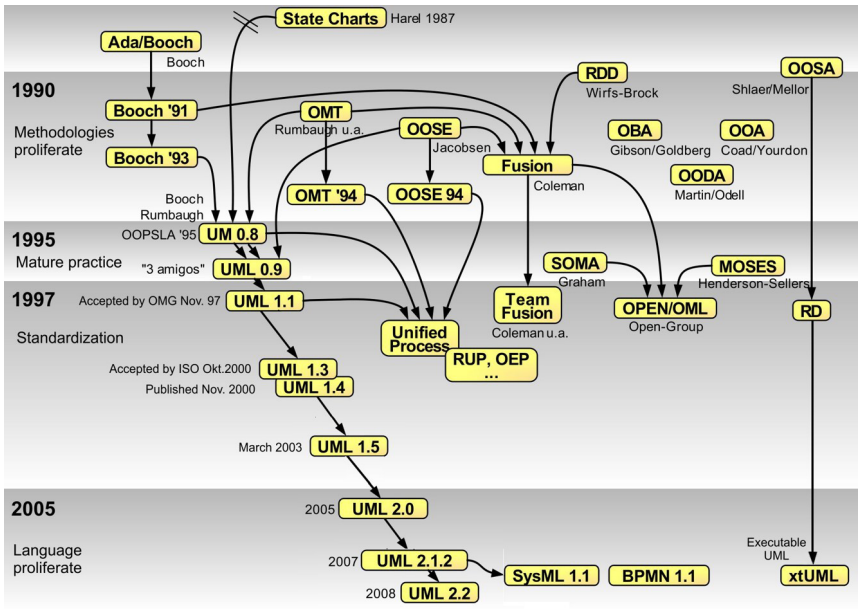
2.3. Szoftverrendszerek modell alapú fejlesztése

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* [10]), vagy a rendszerek modell alapú tervezésére használt SysML (*Systems Modeling Language* [11]), 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 Facility*). A konkrét szintaxist az egyes UML diagram reprezentációk jelentik. Jólformáltsági kényszereket a nyelvhez kapcsolódóan OCL-ben (*Object Constraint Language* [12]) lehet megadni. Az UML formális szemantikáját nem definiálták, helyette egy, az UML-ből származtatott nyelv, az fUML [13] tartalmazza ezt a leírást.



2.2. ábra. Modellezési nyelvek fejlődése

A 2.2 ábrán [14] 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 2.3.1 szakaszban mutatok be.

2.3.1. Shlaer-Mellor módszer

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

1. Objektum-orientált rendszeranalízis (*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 [16]) 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. Azonban a *Shlaer-Mellor módszer* konkrét megoldást nem kínál a kódgenerálási fázisra, azt egy későbbi módszerben, az *xtUML* nyelvben vezették be, melyet a 2.3.3 szakaszban mutatok be röviden.

2.3.2. Executable UML

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

Az Executable UML (*xUML*) mérnöki modellek készítésére definiált modellezési nyelv, amely a *Shlaer-Mellor módszerből* fejlődött tovább. A nyelv elemkészletét a 2.1 táblázat tartalmazza [17].

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 osztály 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 xUML nyelv segítségével megvalósítható egy rendszer platformfüggetlen modelljének analízise és a platformfüggetlen modell működése a viselkedést leíró nyelv alapján egy szimulációs környezet segítségével végrehajtható. Ezáltal megvalósítható egy komplex rendszer modell alapú analízise, tervezése és szimulációja, mellyel hamarabb felderíthetőek a tervezés során elkövetett hibák, rossz döntések.

Ezenkívül az xUML módszer lehetőséget biztosít a platformfüggetlen modellekből platformfüggő modellek transzformációjára, ezáltal a rendszert felépítő egyes komponensek tervei modellszinten karbantarthatóak maradnak, de a konkrét megvalósításról már több részletet tartalmaznak, így a valósághoz már közelebb állnak. Azonban a konkrét transzformációt a módszertan nem tartalmazza, csak mint lehetőséget vetíti előre.

Az Executable UML metodikát követő, mérnöki modellek tervezésére ipari környezetben használt első eszköz a *Kennedy Carter* által fejlesztett *iUML* volt (lásd a 2.4.1. szakaszban).

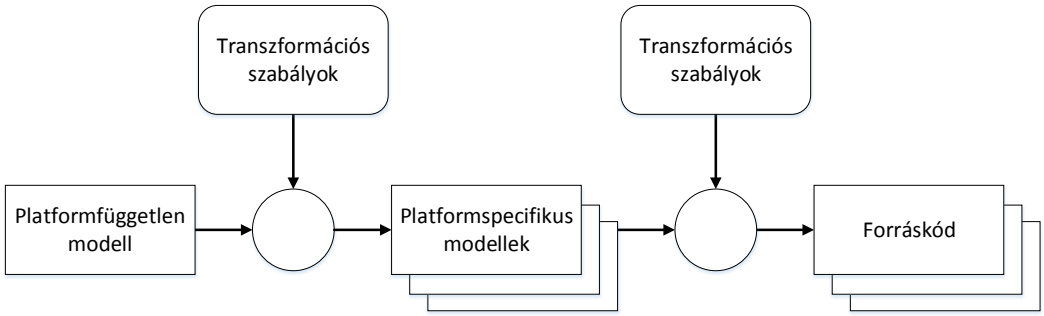
2.3.3. Executable and Translatable UML

Az Executable and Translatable UML (*xtUML*) az xUML-ből származtatott módszertan [4][18]. A nyelv elemkészletét a 3. fejezetben mutatom be.

xtUML-ben lehetőség van a megtervezett platformfüggetlen modelleket egy szimulációs környezetben végrehajtani a modellekhez tartozó, szintén platformfüggetlen viselkedésleíró nyelven megfogalmazott akciók alapján. Ezáltal a létrehozott modellek viselkedése szimulálható, így a tervezés és a megvalósítás során elkövetett hibák hamarabb felderíthetők és javíthatók.

A modellek szimulációján, végrehajtásán túl a modellhez csatolt platformspecifikus kódgenerátor segítségével a platformfüggetlen vagy platformfüggő modellekből és a hozzájuk tartozó viselkedésleíró nyelvből, transzformációs szabályok segítségével (a 2.3 ábrán látható módon) automatikusan származtatható a platformspecifikus implementáció. Ezáltal a megtervezett modellekkel konzisztens forráskód és konfiguráció automatizáltan előállítható. Ennek következtében – amennyiben a kódgenerátor helyes működését ellenőrizték –, elkerülhetőek a manuális forráskód-előállítás során elkövetett programozói hibák, tévedések, illetve a forráskódnak a modellekkel való inkonzisztenciája is.

Az Executable and Translatable UML metodikát követő, mérnöki modellek tervezésére ipari környezetben használt eszköz a BridgePoint, melyet a 2.4 szakaszban mutatok be röviden.



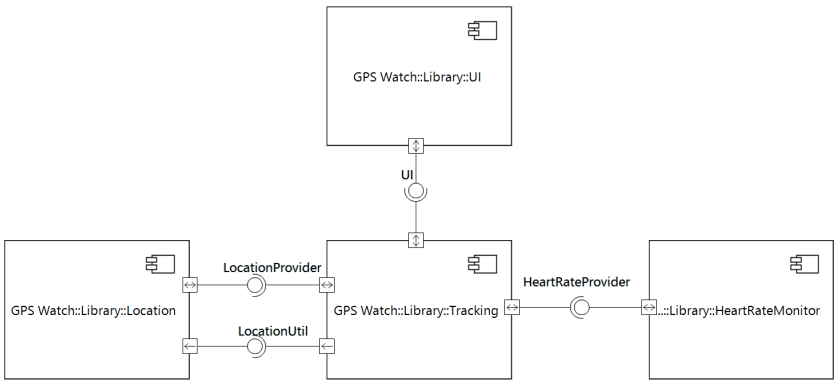
2.3. ábra. xtUML modellekből forráskód származtatása

2.4. Mérnöki modellek tervezése BridgePoint-ban

A korábban a *Mentor Graphics*® által fejlesztett, jelenleg nyílt forráskódú, szabadon hozzáférhető BridgePoint egy mérnöki modellek tervezésére, szimulációjára, a modellekből forráskód automatikus származtatására használt alkalmazás [19]. A szoftverrel a modellvezérelt szoftver- és rendszerfejlesztés metodikája alapján lehet komplex szoftverek és rendszerek szakterület-specifikus mérnöki modelljét elkészíteni [18].

A rendszert felépítő egyes komponensek külön megtervezhetőek és interfészeken keresztül egymáshoz kapcsolhatóak, ezáltal megvalósítva a komponensek felelősségének szeparációját a 2.4 ábrán látható módon. A négy komponensből (*Location, Tracking, UI, HeartRateMonitor*) álló

rendszer egyes komponensei különböző, jól definiált interfészeken (pl. a *Tracking*, *Location* komponensek a *LocationProvider*, *LocationUtil*) keresztül kommunikálnak egymással, ezáltal az egyes komponensek felelősségének szeparációja megvalósítható.



2.4. ábra. Komponens diagram példa

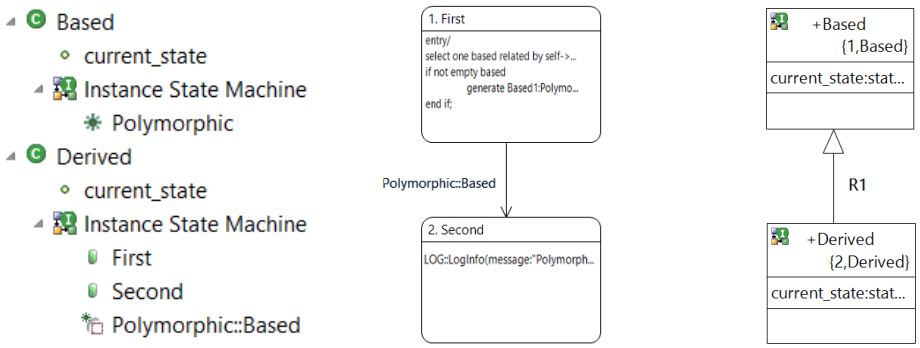
A komponenseken belül UML osztálydiagramszerű struktúrákkal lehet leírni az adott komponens belső szerkezetét. Az osztálydiagramon ábrázolt egyes osztályok viselkedése állapotgépekkel írható le.

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 [20].

BridgePoint-ban lehetőség van a komponensen belüli osztályok között specializációs viszonyt definiálni. Ezáltal bevezethető a polimorf eseménykezelés, mellyel csökkenthető a komponensen belül az osztályok egymástól való függősége.

Polimorf eseménynek neveznek egy leszármaztatási hierarchiában egy olyan eseményt, amely az ősosztály állapotgépében nem, de a leszármazott osztály állapotgépében legalább egy állapotátmenethez kötve van a 2.5 ábrán látható módon.



2.5. ábra. Polimorf eseménykezelés példa

Az ábrán a *Based* ősosztály és a belőle származtatott *Derived* osztály látható. A *Based* osztály rendelkezik egy *Polymorphic* eseménnyel, amely a leszármazott osztály kétállapotú 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.

Az egyes komponenseket lehetőség van interfészeken keresztül olyan komponensekhez csatolni, melyeket nem lehet modellszinten megalkotni, hanem közvetlenül platformspecifikus forráskódot tartalmaznak. Így megvalósítható, hogy a közvetlenül nem modellezhető részek az egyes komponensek modelljeiből is elérhetőek legyenek, ezáltal hozzájárulva heterogén rendszerek integrálhatóságához.

2.4.1. Iparban elterjedt modell alapú fejlesztőeszközök

A *Mentor Graphics*® által fejlesztett BridgePoint-on kívül számos modell alapú fejlesztőkörnyezet létezik, melyek közül néhányat ismertetek 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 használt belső modellezési nyelvük nem szabványos, és a zárt működés következtében formális szemantikájuk sem ismert. Azonban fontos megjegyezni, hogy ezen zárt eszközöknek a kódgenerátora 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 akciónyelvként. Ezzel szemben a BridgePoint-ban OAL kódot alkalmaznak, amely egy magasabb szintű nyelv, mellyel lehetőség van platformfüggetlen modellek viselkedésének definiálására, amelyből 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* [21]) és xUML metodikát. Azonban a BridgePoint-hoz képest kevés, nyilvánosan elérhető dokumentációval rendelkezik, illetve az xtUML elődjét követi, ezért nem ezt a megoldást választottam.

2.5. Verifikáció

Munkám során kiemelt figyelmet fordítottam arra, hogy olyan tervezési eljárásokat és módszereket alkalmazzak, amelyek biztosítják, hogy az elkészült rendszer helyesen működjön.

Azonban a tervezési hibák kiszűrésére új módszert kellett alkalmaznom: formális verifikációt használtam. A formális verifikáció olyan módszer, amely a rendszertervek formális modelljein képes bizonyítani azok helyességét, vagy megtalálni az esetleges tervezési hibákat. Munkám során állapotgép alapú modellezési megközelítést alkalmaztam, ezért a lehetséges formális megközelíté-

sek közül a modellellenőrzésre esett a választásom. A modellellenőrzés során annak a vizsgálata történik, hogy egy adott formális modell megfelel-e a vele szemben támasztott formális követelményeknek.

Ebben a fejezetben bemutatom az általam használt formális modellező nyelvet, a temporális logikai specifikációs nyelvet, továbbá ezt ellenőrzés céljából felhasználó, az iparban is elterjedt modellellenőrző keretrendszert.

2.5.1. 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 $\mathcal{A}$ automata 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, az eseményvezérelt véges automatákat használják. 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 $l \xrightarrow{g,a} l'$ jelölés akkor használható, ha $\langle l, g, a, l' \rangle \in E$, azaz vezet l állapotból átmenet l' állapotba, amelyhez g őrfeltétel és a akció van rendelve.

2.5.2. Automaták hálózata

Elosztott, időzített rendszerek verifikációjára több időzített automata hálózata jelentheti a megoldást. 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.

2.5.3. Az UPPAAL modellellenőrző eszköz

Az UPPAAL egy valósidejű rendszerek modellezésére, szimulációjára és verifikációjára használt keretrendszer, melyet az Uppsalai Egyetemen és az Aalborgi Egyetemen közösen fejlesztenek. Első változata 1995-ben jelent meg, jelenleg a 4.0-s verzió a legfrissebb [22].

Az UPPAAL képes automaták hálózatát leírni, ahol az automata jelentése a fent definiáltaknak felel meg. Ezenfelül pedig további bővítéseket is illeszt az automata formalizmushoz, ezeket a továbbiakban mutatom be [23].

2.5.4. Véges automaták UPPAAL-ban

Az UPPAAL-ban definiálható véges automaták felépítése hasonló az UML állapotgépekhez. Egy automata két alapeleme a következő:

- *állapotok*: névvel rendelkeznek, 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 tulajdonságok rendelhetők. Ilyenek például *helyinvariánsok* definiálása, *szinkronizációk* megadása az átmenetekhez vagy *óraváltozók* definiálása.

2.5.5. 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 viselkedés is modellezhető a rendszerben. Általában ezen óraváltozók pontos értéke nem ismert, viszont összehasonlíthatóak konstansokkal, ezáltal intervallumokat képezve. Az óraváltozók két fontos tulajdonsággal rendelkeznek:

- Az óraváltozók értéke vagy monoton növekszik vagy explicit módon nullázódik.
- 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 szabályozható az állapotokban eltöltött időt.

Mivel a 4.3.7 fejezetben később bemutatott modell időzítéseket nem tartalmaz, ezért a dolgozatban ezzel a területtel nem foglalkoztam, azonban az 5.2 fejezetben később bemutatott leképezés segítségével időzített analízisre is lehetőség lenne.

2.5.6. Szinkronizáció UPPAAL-ban

Elosztott, időzített rendszerek működése UPPAAL-ban az előzőekben említett időzített automaták hálózatával modellezhető. 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 akció csatolható: 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 az `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.

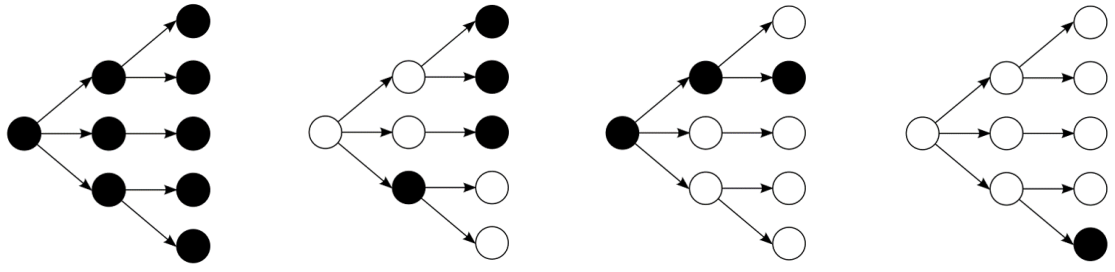
Broadcast szinkronizáció egy küldő $\mathcal{A}$ és (0..*) $\mathcal{B}_i$ fogadó automata között valósulhat meg. A fogadó fél viselkedése megegyezik az egyszerű szinkronizációban leírtakkal, viszont a küldő fél az összes $\mathcal{B}_i$ fogadó automatának elküldi a szinkronizációs üzenetet, még akkor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

2.5.7. 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 nyelven, temporális logikával van kifejezve. 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. Ezek a követelmények három csoportba sorolhatók:

- elérhetőségi követelmények,
- biztonsági követelmények,
- élőségi követelmények.

A 2.6 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

2.6. ábra. UPPAAL-ban használt CTL operátorok

2.5.8. Elérhetőségi követelmények

Az elérhetőségi követelmények során azt vizsgálja a modellellenőrző, hogy egy adott ϕ kifejezés teljesül-e a jövőben valamely állapot elérésekor. Az ilyen követelményeket CTL-ben az `EF`ϕ, míg UPPAAL-ban az `E<>`ϕ kifejezéssel lehet megfogalmazni.

2.5.9. 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 később említendő holtpontmentesség, de a kölcsönös kizárást és az adatbiztonságot is fel lehet hozni példának.

Ha ϕ egy állapotkifejezés, és ϕ-nek minden elérhető állapotban igaznak kell lennie, akkor az ezt kifejező formula CTL-ben `AG`ϕ, míg UPPAAL-ban `A[]`ϕ.

2.5.10. 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 `AF`ϕ. Legegyszerűbb esetben UPPAAL-ban az élőségi kifejezéseket a `A<>`ϕ kifejezéssel fogalmazhatóak meg.

Emellett az UPPAAL-ban definiálható egy másik élőségi követelmény 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.

2.5.11. Állapotkifejezések

Egy állapotkifejezés egy olyan kifejezés, melynek kiértékelése az adott állapoton történik. A szintaxisa pedig az őrfeltételekéhez hasonló, azaz a kifejezések mellékhatásmentesek, de a diszjunkció használata nem tiltott.

Ugyanakkor *P.loc* típusú kifejezések segítségével, ahol *P* a folyamat és *loc* az adott állapot, lehetőség van leellenőrizni, hogy egy folyamat (automata) egy adott állapotban tartózkodik-e vagy sem. Az UPPAAL a holtpont ellenőrzésére egy speciális állapotkifejezést használ, ami a `deadlock` kulcsszóból áll és mindig teljesül, ha egy automata holtpontban van. Például, a modell holtpontmentességének ellenőrzéséhez, az `A[] not deadlock` kifejezést kell megadni az eszköz verifikációs moduljának.

3. fejezet

xtUML elemek

Ebben a fejezetben bemutatom a 2.3.3 fejezetben bevezetett xtUML nyelvet és módszertant, melynek áttekintem főbb elemeit [4].

Az Executable and Translatable UML (*xtUML*) az xUML-ből származtatott módszertan. xtUML-ben lehetőség van a megtervezett platformfüggetlen modelleket egy szimulációs környezetben végrehajtani a modellekhez tartozó, szintén platformfüggetlen viselkedésleíró nyelven megfogalmazott akciók alapján. Ezáltal a létrehozott modellek viselkedése szimulálható, így a tervezés és a megvalósítás során elkövetett hibák hamarabb felderíthetővé és javíthatóvá válnak.

A módszer alkalmazásával lehetőség van komplex rendszerek mérnöki modelljének iteratív, szisztematikus elkészítésére, amely folyamat a következő lépésekből áll:

1. Követelmény analízis
2. Rendszerterv modell elkészítése
3. Komponens modellek elkészítése
4. Komponenseket felépítő egységek modelljeinek elkészítése
5. Modellek szimulációja
6. Modellekből forráskód származtatása

Az egyes nyelvi, módszertani elemeket Stephen J. Mellor és Marc J. Balcer *Executable UML: A Foundation for Model-Driven Architecture* című könyve alapján mutatom be [18]. Bár az elemek az xUML-hez vannak definiálva, xtUML esetén ugyanúgy használhatóak, hiszen a két módszer között az egyetlen eltérés, hogy xtUML esetén a modellekből automatikusan előállítható a viselkedésüket megvalósító platformfüggő forráskód és konfiguráció.

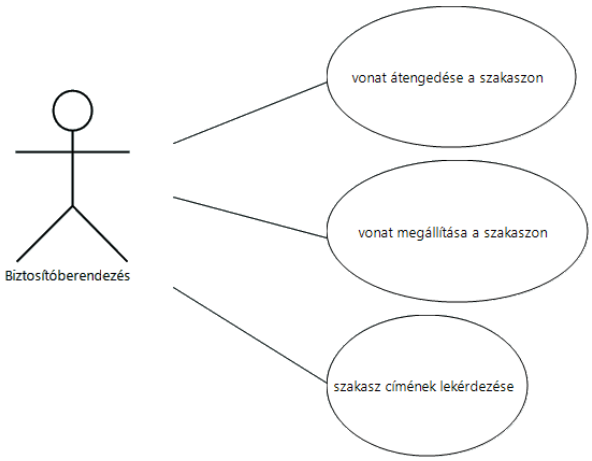
3.1. Követelmény analízis

Követelmény analízis során a rendszerrel szemben állított követelményeket vizsgálják meg, melyek alapján előállítják a rendszer működését leíró specifikációt, amely mentén a tervezés során teljesíteni lehet a definiált követelményeket.

A követelmények modellszintű ábrázolásához több, az UML-ben definiáltakhoz hasonló diagram és modelltípus is használható, köztük a használati eset és aktivitás diagramok, valamint a szekvencia diagramok is.

3.1.1. Használati eset diagram

Használati eset diagramokon aktorok és használati esetek jelennek meg, melyek bemutatják, hogy az aktorok milyen célokra akarják a rendszert, vagy annak egyes részeit igénybe venni.

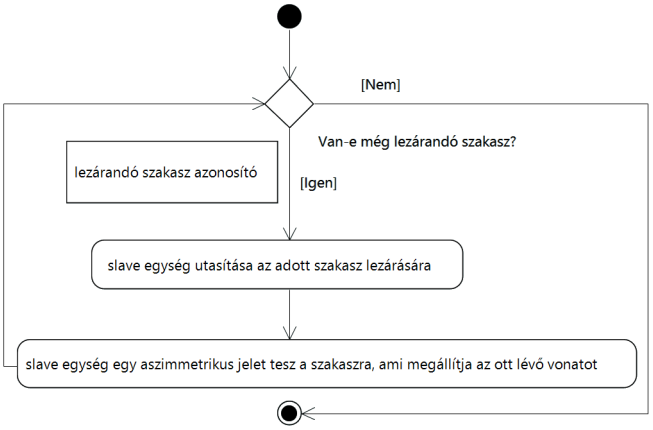


3.1. ábra. Használati eset diagram példa

A 3.1 ábrán látható, hogy a biztosítóberendezés a vonatok átengedésében, megállításában, illetve a szakasz címének lekérdezésében vesz részt, ezekhez kapcsolódik működése során.

3.1.2. Aktivitás diagram

Aktivitás diagramokon cselekvések, aktivitások sorrendisége jelenik meg, amely egy nagyobb, komplexebb viselkedés megvalósításához szükséges. Ezáltal a rendszer egyes részeinek komplex működése magasszinten vizualizáltan leírható, az adott aktivitás szempontjából irreleváns részek elhagyhatóak, ezáltal könnyebben áttekinthetővé válnak az összetett folyamatok.

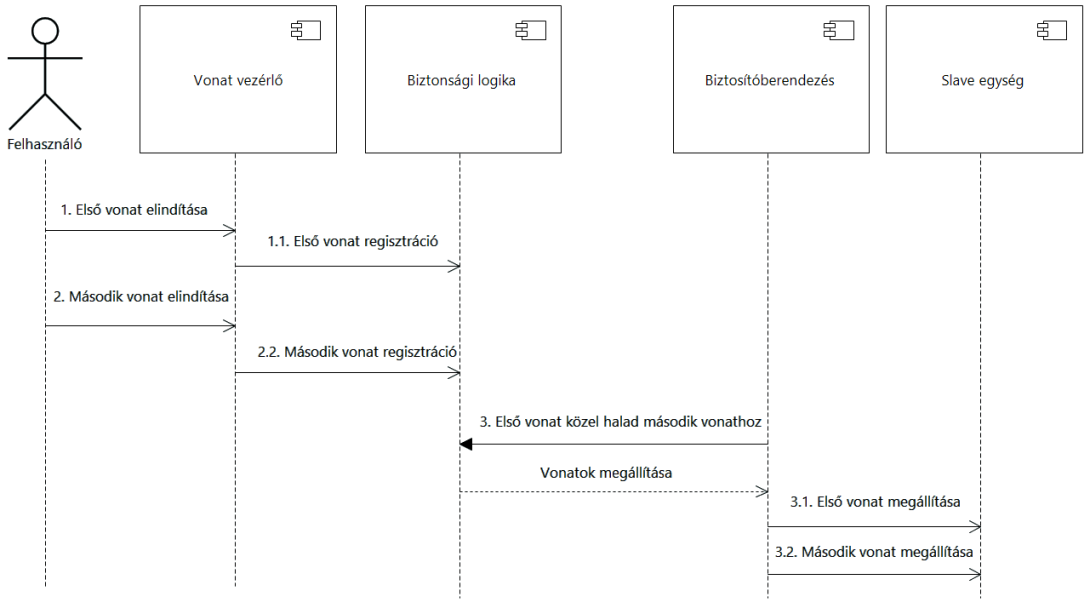


3.2. ábra. Aktivitás diagram példa

A 3.2 ábrán látható egy biztosítóberendezés működésének leírása: ameddig van a biztosítóberendezés által lezárandó szakasz, addig a biztosítóberendezés utasítja az adott szakaszhoz tartozó szolga (*slave*) egységet, hogy zárja le a szakaszt, melynek hatására a *slave* egység megállítja a rajta lévő vonatot. Ha már nincsen több lezárandó szakasz befejeződik a cselekvés.

3.1.3. Szekvencia diagram

A szekvencia diagram két, vagy több elem interakciójának, kommunikációjának sorrendiségét mutatja, melyen időbeliségi és időzítési kritériumok is megadhatók. Ezáltal a rendszer komponenseinek együttműködése egyszerűen vizualizálható és áttekinthető.



3.3. ábra. Szekvencia diagram példa

A 3.3 ábrán látható, hogy a felhasználó elindít két vonatot, melyeket a biztosítóberendezés a *slave* egységeken keresztül megállít, mert túlságosan megközelítették egymást.

3.2. Rendszerterv modell

Az xtUML alapú tervezés fontos koncepciója, hogy a rendszer megtervezése előtt pontosan definiálva legyen a szakterület (*domain*) aminek az elemeit a mérnökök modellszinten meg akarják tervezni. Ez alapján a teljes rendszerterv modell a rendszert felépítő egyes területek (*domain*) modelljeinek egymáshoz kapcsolásából épül fel.

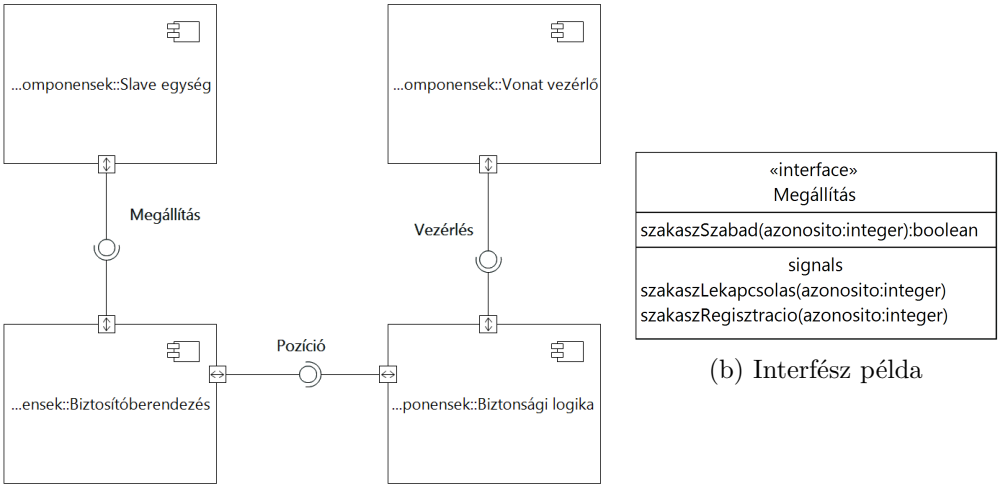
3.3. Komponens modellek

A rendszert felépítő egyes domainek szisztematikus leírásához szükséges, hogy a tervezőmérnökök különböző felelősségi körökkel és funkcionalitással rendelkező komponenseket hozzanak létre, melyeket egymáshoz és a külvilághoz jól definiált interfészeken keresztül kapcsoljanak.

A külvilághoz való kapcsolódás történhet olyan komponenseken keresztül, amelyek modellszinten nincsenek leképezve, hanem közvetlenül platformspecifikus forráskódból épülnek fel, így a jól definiált interfészeken keresztül a modell számára transzparensen viselkednek.

Az interfészek egyrészt tartalmazhatnak műveleteket, függvényhívásokat, másrészt tartalmazhatnak jelzéseket, eseményeket, melyeket az egyes komponensek egymásnak küldhetnek.

A platformfüggetlen viselkedésleíró nyelv használatával, és az interfészeken keresztüli jelzésküldések segítségével az egyes komponensek kollaborációja modellszinten, platformfüggetlenül leírható, melyből tetszőleges platformfüggő modell és forráskód származtatható.



(a) Komponens modell példa

(b) Interfész példa

3.4. ábra. Komponens modell és interfész példa

A 3.4a ábrán látható négy komponens (*Slave egység, Biztosítóberendezés, Biztonsági logika, Vonat vezérlő*), melyek jól definiált interfészeken (*Megállítás, Pozíció, Vezérlés*) kapcsolódnak egymáshoz.

A 3.4b ábrán látható *Vezérlés* interfész tartalmaz egy metódust, mellyel egy megadott szakasz szabad állapota lekérdezhető, illetve két szignált (jelzést), melyekkel egy adott azonosítóval rendelkező szakasz lekapcsolható, illetve beregisztrálható.

3.4. Komponenseket felépítő egységek modelljei

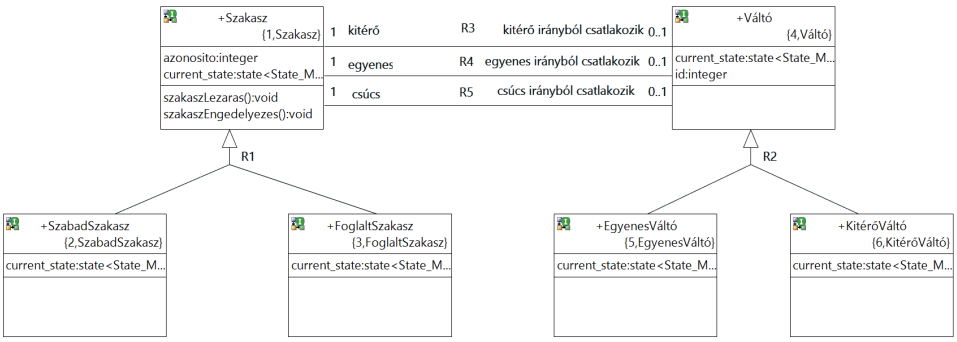
Az egyes komponenseken belül a modell statikus leírását UML osztálydiagramszerű struktúrában lehet megadni. A struktúra tartalmazza egyrészt az osztályokat, másrészt az osztályokat összekötő asszociációkat is, melyek különböző kardinalitással rendelkezhetnek.

Az egyes osztályokból más osztályok származhatnak le. A leszármazott osztályokra megkötés, hogy azok az ősosztály speciális változatai kell, hogy legyenek, illetve a leszármazottaknak nem lehet közös tulajdonsága (az ősosztályon kívül), különben a közös tulajdonságot egy közös ősben kell bevezetni.

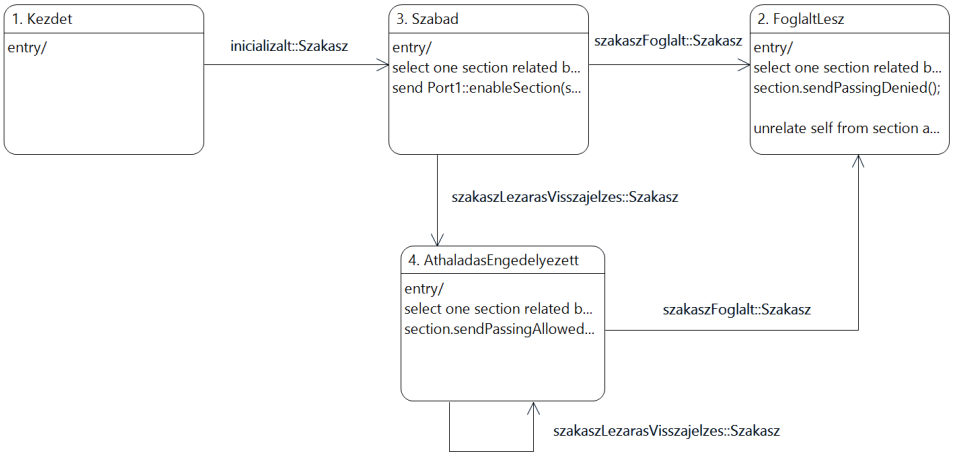
Az egyes osztályok viselkedését az UML-ben definiált állapotgépekhez hasonló, azoknál egyszerűbb szerkezetű állapotgépekben lehet megadni. Az állapotgépek állapotokból, különböző

események hatására bekövetkező állapotátmenetekből, illetve az egyes állapotokba vagy állapotátmenetekre viselkedésleíró nyelven írt akciókból épülnek fel.

Az akciók segítségével az állapotgépek dinamikus viselkedése megadható. Események hozhatók létre melyek elsüthetőek egy már létező állapotgéppel rendelkező objektumnak, vagy egy interfészen keresztül szignálként továbbíthatóak egy másik komponensnek.



(a) Osztálydiagram példa



(b) Állapotgép példa

3.5. ábra. Osztálydiagram és állapotgép példa

A 3.5a ábrán a 3.4 ábrán bemutatott *Biztonsági logika* nevű komponens egyszerűsített belső szerkezete látható. A biztonsági logika egyrészt szakaszokból, másrészt váltóból áll.

A szakaszoknál a foglaltságuk alapján megkülönböztettem *szabad* vagy *foglalt* szakaszokat. A váltó aktuális állása alapján lehet *egyenes* vagy *kitérő* állású. Egy szakasz *egyenes*, *kitérő*, vagy *csúcs* irányból csatlakozhat egy váltóhoz.

Modellszinten statikusan nem lehet garantálni, hogy egy szakasz egy váltóhoz csak egy irányból csatlakozhasson. xtUML-ben az UML-hez bevezetett OCL kényszerleíró nyelvhez hasonló nyelv nem áll rendelkezésre, így a modellre vonatkozó kényszerek érvényesítésére a tervezőmérnöknek kell figyelnie.

A 3.5b ábrán a *SzabadSzakasz* osztály állapotgépe látható. Az állapotgép kezdőállapota a *Kezdet* nevű állapot, melyből az *inicializalt* esemény (melyet a *Szakasz* ősosztálytól örökölt) hatására átmegy *Szabad* állapotba. További viselkedése az állapotoktól és állapotátmenetektől függ. A nem kezelt eseményeket az állapotgép figyelmen kívül hagyja és eldobja.

3.5. Modellek szimulációja

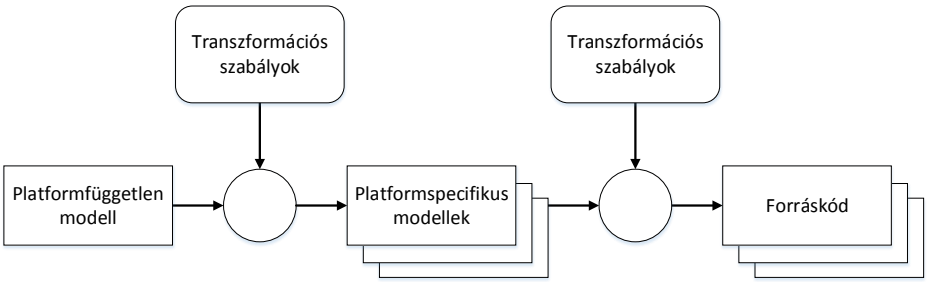
Az UML korai változatai nem támogatták a modellek közvetlen végrehajthatóságát, mert a viselkedésleíró elemeknek (jelzések küldése, objektumok létrehozása és megszüntetése) csak egy korlátozott halmazát tartalmazták. A nyelvet az évek során kiterjesztették a viselkedésleíró elemek szemantikájának definiálásával, ezáltal magasszintű leírási lehetőséget adva a modellek viselkedésére.

Az xtUML-ben – az xUML-ben bevezetett módon – definiálták a viselkedésleírás dinamikus szemantikáját. Dinamikusan minden objektum a többi objektumhoz képest aszinkron, konkurens módon fut. Minden objektum egy adott pillanatban vagy egy folyamatot hajt végre, vagy egy esemény bekövetkezésére várakozik. Az objektumok által végrehajtott események sorrendje egymástól független, nincsen egy közös, globális idő, így minden szinkronizációt az objektumok között explicit modellezni kell [18].

A dinamikus szemantika definíciójának következtében a modellek viselkedése magasszinten, a konkrét implementációtól függetlenül végrehajtható, szimulálható. Ezáltal a platformspecifikus forráskód előállítása előtt észlelhetők és javíthatók a tervezés során elkövetett tévedések, melynek következtében a modell hibamentessége javítható és csökkenthető a hibák későbbi javításából adódó magas költségek.

3.6. Modellekből forráskód származtatása

Az Executable and Translatable UML, az Executable UML-en túlmutatva lehetőséget biztosít a platformfüggetlen modellekből és viselkedésleíró nyelvből platformfüggő modellek származtatására, melyeken keresztül egy kódgenerátor segítségével, automatikusan előállítható a platformspecifikus forráskód és konfiguráció.



3.6. ábra. xtUML modellekből forráskód származtatása

A 3.6 ábrán látható a forráskód előállítást megvalósító általános transzformációs folyamat. Először a platformfüggetlen, általános modellekből transzformációs szabályok alapján származtatják a platformspecifikus részletekkel ellátott platformfüggő modelleket, melyekből újabb transzformációs szabályokkal előállítható a modellek működését megvalósító tetszőleges programnyelvű forráskód és konfiguráció.

A transzformációs lánc automatizáltságának következtében – amennyiben a kódgenerátorok helyes működését ellenőrizték – elkerülhetőek a manuális forráskód-előállítás során elkövetett programozói hibák, tévedések, és az implementáció modellekkel való inkonzisztenciája is.

3.7. xtUML alapú tervezés értékelése

Szakdolgozatom egyik feladataként értékelem az xtUML alapú tervezést, a 3.7.1 szakaszban bemutatva az előnyeit, illetve a 3.7.2 szakaszban bemutatva a hátrányait is.

3.7.1. xtUML előnyei

Az xtUML alapú tervezés és fejlesztés legnagyobb előnye a szimulációs keretrendszer adta korai ellenőrizhetőség, mellyel a tervezés korai fázisában már ellenőrizhető a platformfüggetlen modellek specifikációnak megfelelő működése, ezáltal a hibák késői kijavításából adódó jelentős költséget lehet megtakarítani.

További előnye, hogy a platformfüggetlen modellekből transzformációs szabályok felhasználásával automatizáltan előállítható a platformspecifikus modell, amely egy szinttel közelebb van a megvalósításhoz, mint a platformfüggetlen megfelelője. Ezáltal még modellszinten, de a valósághoz közelebb elhelyezkedve lehet a rendszer működését vizsgálni, ellenőrizni. Továbbá a platformspecifikus modellekből újabb transzformációs szabályok felhasználásával automatizáltan származtatható a modellt megvalósító forráskód és implementáció, amely ha a kódgenerátor helyességét ellenőrizték, konzisztens a modellel, ezáltal elkerülhetőek a manuális implementáció során elkövetett fejlesztői hibák, tévedések is.

Előnyei közé tartozik még, hogy támogatja a beágyazott rendszerek tervezésekor általánosan használt állapotgép alapú modellezést és megvalósítást. Ennek során a rendszer csak az előre definiált eseményekre reagál, és a mindenkori állapotától függ, hogy milyen cselekvést fog végrehajtani. Ezáltal a beágyazott rendszer viselkedése viszonylag egyszerűen megadható, mert szinte mindig egy jól definiált célfeladat teljesítésére szokták igénybe venni.

3.7.2. xtUML hátrányai

Az xtUML alapú tervezés legnagyobb hátránya, hogy az UML-hez bevezetett OCL kényszerleíró nyelvhez hasonló nyelv nem áll rendelkezésre, így a modellre vonatkozó kényszerek érvényesítésére a tervezőmérnöknek kell figyelnie, az automatikus vizsgálat hiánya miatt.

További hátránya, hogy az állapotgép alapú megközelítés nem minden rendszer tervezésekor a legcélszerűbb megoldás, mert a viselkedés állapotalapú leírása modellszinten rendkívül bonyolult lehet. Ebből adódóan mindig a feladatnak leginkább megfelelő absztrakciót kell választani a modell megtervezése során, különben jelentős többletköltség adódik a tervezéskor és a fejlesztéskor a nem megfelelő absztrakciós szint miatt.

4. fejezet

Demonstrátor eszköz biztonsági logikájának modell alapú tervezése

A Méréstechnika és Információs Rendszerek Tanszéken 2014 nyarán elkészült egy elosztott vasútirányítási biztonságkritikus demonstrátor esettanulmány terepasztal, melyhez szakdolgozatomban egy biztonsági logikát terveztem meg modell alapon. A biztonsági logikát egy modellszimulációs környezeten keresztül kapcsoltam a terepasztalhoz. Ezáltal lehetővé tettem, hogy emberi beavatkozás nélkül megálljanak a vonatok, ha veszélyes szituáció alakul ki a vonatok terepasztalon való közlekedése során.

A 4.1 alfejezetben röviden bemutatom az esettanulmányt, a modellvasút terepasztalt, és a hozzá tartozó elosztott, beágyazott mikrokontroller alapú hálózatot. A 4.2 szakaszban ábrázolom a rendszerhierarchiát, és a tanszéki demonstrátor szoftver (*biztonsági logika*) kapcsolódását a terepasztalhoz. A 4.3 alfejezetben részletesen bemutatom a szakdolgozatomban modellalapon megtervezett biztonsági logikát.

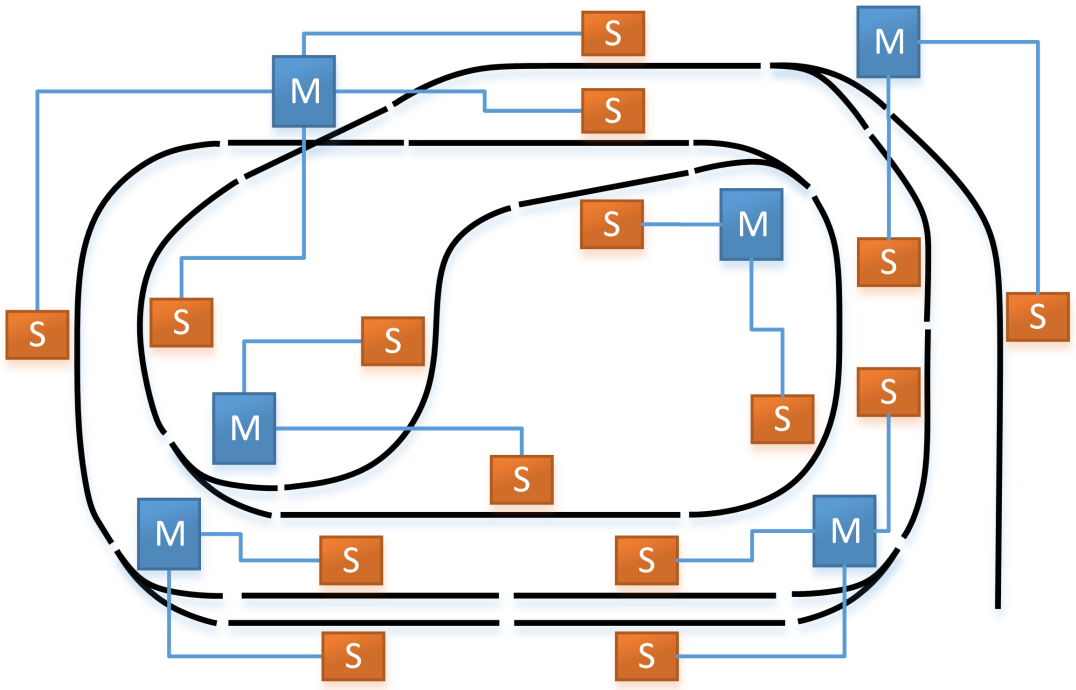
A BridgePoint-ban megtervezett és megvalósított modellekből az 5.1 fejezetben később bemutatott módon automatizáltan származtattam formális modelleket, melyeken a 6. fejezetben később leírt verifikációt hajtottam végre. Ezáltal szakdolgozatomban ellenőrizhetőség szempontjából megvizsgáltam az xtUML alapú rendszerterveket.

4.1. Esettanulmány rövid bemutatása

A terepasztal 15 db sínszakaszból, és 6 db váltóból épül fel a 4.1 ábrán látható módon. A vonatok haladását a digitális vezérlésű modellvasutaknál használt, úgynevezett DCC (*Digital Command Control* [24]) protokoll alapján történik. A protokollba kívülről beavatkozni nem lehet, a szigorú időzítésalapú kódolás miatt. Azonban a mozdonyokat megállásra lehet kényszeríteni, ha a bennük lévő dekóderek egy aszimmetrikus jelet érzékelnek a sínpárban (ABC mód [25]).

Ennek következtében a vezérléstől függetlenül elkészült egy elosztott, beágyazott rendszereket tartalmazó beavatkozó hálózat, melynek feladata, hogy az említett ABC mód segítségével megállítsa a vonatokat. A hálózatban – a 4.1 ábra szerint – megkülönböztetnek *master* és *slave* egységeket, melyek egymástól eltérő feladatkörrel rendelkeznek:

- *master*: a biztonsági logika által utasított egységek;



4.1. ábra. Beavatkozó egységek elhelyezkedése a hálózatban

- *slave*: a fizikai beavatkozást az ABC mód segítségével végző egységek.

Minden szakaszhoz tartozik egy *slave* egység, melynek feladata az ABC mód segítségével a vonatok megállítása a hozzá tartozó szakaszon. Minden váltóhoz tartozik egy *master* egység, amely vezérli a váltóhoz csatlakozó szakaszokhoz illesztett *slave* egységeket. Ezáltal lehetőség van a vonatok haladásába a DCC protokoll szerinti vezérlésétől függetlenül beavatkozni.

A modellvasút terepasztalt és a hozzá tartozó elosztott, mikrontroller-alapú beavatkozó hálózatot Mázló Zsolt tervezte és valósította meg.

4.2. Konfiguráció a BridgePoint-tal való együttműködéshez

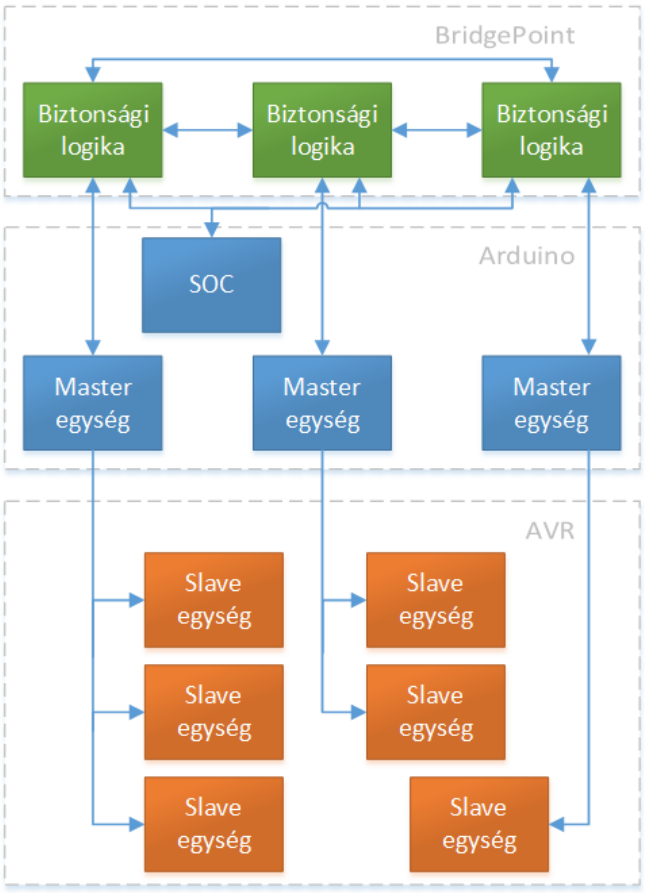
A hardver architektúrát a BridgePoint-ban általam létrehozott biztonsági logikával a 4.2 ábrán látható módon kapcsoltam össze [26]. A biztonsági logikát a BridgePoint modellek valósítják meg saját állapotgépeikkel, amelyek a 4.3.4 szakaszban leírtak alapján viselkednek.

A BridgePoint lehetőséget biztosít arra, hogy a modell úgynevezett realizált komponensen keresztül legyen összekapcsolva a környezetével, melyben platformspecifikus C++ vagy Java kód futtatására van lehetőség. Ezáltal a modellekből közvetlenül kommunikálni lehet a terepasztalhoz illesztett *master* egységekkel.

Minden *master* egységhez egy, a BridgePoint szimulátorában futtatott példánymodellt társítottam. A *master* egységek statikus, parancsokon alapuló funkcionalitással rendelkeznek, és a realizált komponenseken keresztül egy általam definiált formátum alapján kommunikálnak a BridgePoint állapotgépekkel.

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

Ezáltal megvalósítottam a 4.3.7 szakaszban modellszinten megalkotott biztonsági logika összekapcsolódását a hardver architektúrával, így a modellvasút terepasztallal.



4.2. ábra. A hálózat összeköttetése a BridgePoint állapotgépekkel

4.3. Biztonsági logika modell alapú tervezése

A következő alfejezetekben bemutatom az általam xtUML alapon megtervezett biztonsági logikát, a hozzá tartozó követelményekkel, specifikációval, tervezéssel és szimulációval együtt.

A modellekben szerepelő fogalmak, a 3. fejezetben bemutatott példákhoz képest szándékosan angol nyelvűek, mert ezáltal akár nemzetközi célközönség számára is érthetőek lesznek az egyes elemek.

4.3.1. 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 dolgoztam 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 közlekedhetnek. 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 modellvasút terepasztalhoz elektronikai okok miatt nem lehetett 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 4.1 ábrán) folytonos vonallal jelölt szakasz vagy váltó. Ha ez a követelmény nem teljesíthető, akkor a közvetlenül szomszédos szakaszokon lévő mozdonyokat feltétlenül meg kell állítani.

4.3.2. Specifikáció

A követelmények alapján elkészítettem a rendszer specifikációját. A biztonsági logika vonatok vezérléstől való szeparációjának teljesítéséhez egy teljesen független alrendszert hoztam 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 indexeknek felelnek meg. Az adott helyiértékű bit jelöli a hozzá tartozó egység foglaltságát. Egy pályaegységet foglaltnak tekintek, 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 kell parancsot adnia 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 megakadályozására és a feladat komplexitásának kezelése érdekében szükséges, hogy a biztonsági logikát elosztottan, több szintre bontva valósítsam meg. Az elosztottságot a pályához tartozó váltók mentén teljesítettem.

4.3.3. Tervezési feltételezések és kényszerek

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

- 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 ott 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.

- 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 általam kidolgozott biztonsági alrendszer nem biztosítja, annak helyes működésének szükséges, de nem elégséges feltételei.

4.3.4. Szakasz lezárási protokoll

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

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

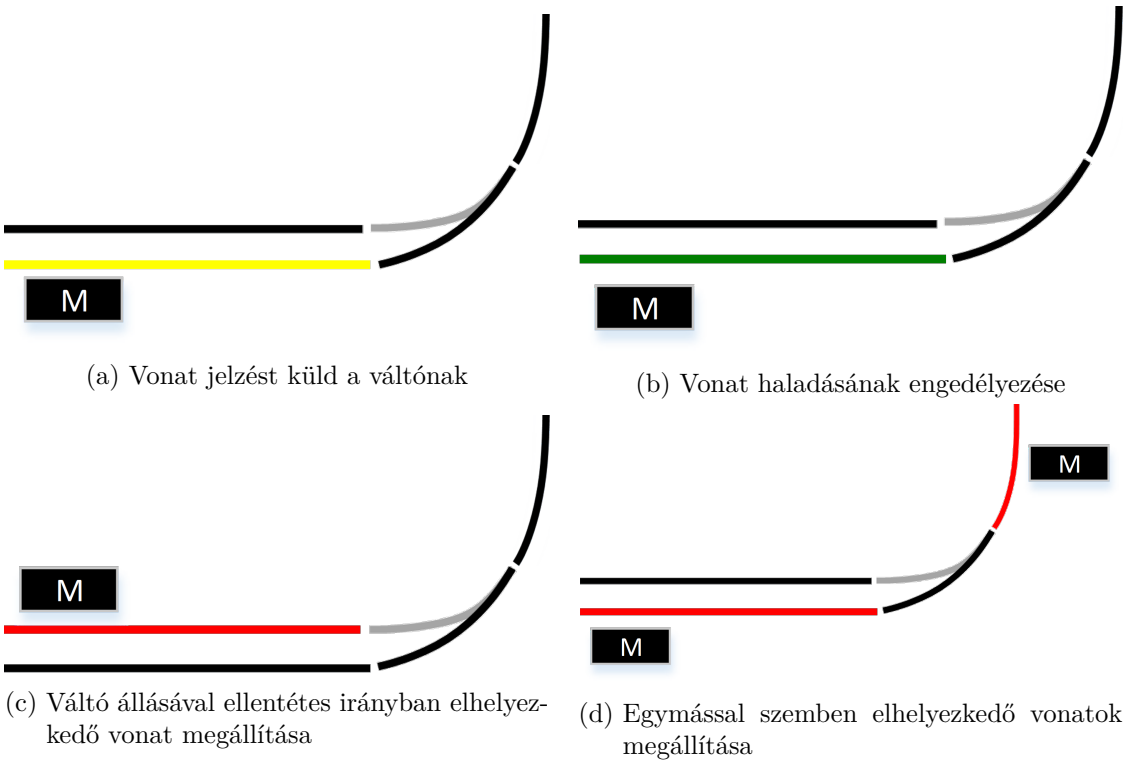
A biztonságkritikus szituációk elkerülésének szükséges, de nem elégséges feltételei a lokális döntések. Az elégségesség teljesítéséhez a szomszédos váltókhoz tartozó szakaszok állapotait is ismerni kell, ezért az egyes váltókhoz tartozó biztonsági logika modelleknek kommunikálniuk kell egymással. Az elosztott kommunikációhoz tartozó döntéseknek és a lokális döntéseknek együtt kell érvényre jutniuk. Ha bármelyik döntésben egy szakasz lekapcsolása szerepel, akkor annak a döntésnek a 4.3.3 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.

4.3.4.1. 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 a 4.3 ábrán látható módon, 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 jelzés.



(a) Vonat jelzést küld a váltónak

(b) Vonat haladásának engedélyezése

(c) Váltó állásával ellentétes irányban elhelyezkedő vonat megállítása

(d) Egymással szemben elhelyezkedő vonatok megállítása

4.3. ábra. Szakasz lezárási protokoll lokális döntés

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

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, engedélyezi a mozdony áthaladását. Amennyiben van a túlsó

oldalon vonat, mindkét szakaszhoz tartozó mikrokontrollernek tiltó üzenetet küld a vonatok megállítása céljából.

Váltóhoz tartozó *lokális döntés* esetén a váltó foglaltságát nem vizsgálom, mert feltételezem, 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 ér.

4.3.4.2. Globális döntés

Globális döntés esetén a *szomszédos* váltók a *lokális* információikat felhasználva – a 4.4 ábrán látható módon – eldöntik, 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 a vonatnak a továbbhaladást. Ennek hatására az említett szomszédos váltó megállítja a területén lévő vonatot, a szakaszhoz tartozó mikrokontrolleren keresztül.

A 4.1 ábrán látható pályarajz bal felső részén elhelyezkedő, két egymást metsző szárral rendelkező váltót nevezik 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 elkerüljem, hogy két szomszédos váltó területén, de egymástól még legalább két szabad szakasz és váltó távolságra lévő vonatok megálljanak, prioritásokat rendeltem 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 4.4b ábrán).

(a) Vonatok haladásának engedélyezése mindkét váltóra

(b) Vonat megállítása prioritás alapján

4.4. ábra. Szakasz lezárási protokoll globális döntés

4.3.4.3. Lokális és globális döntések kiértékelése

Egy adott váltóhoz tartozó *lokális* és *globális* döntések kiértékelése során az ellentmondások elkerülése érdekében prioritásokat rendeltem 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.

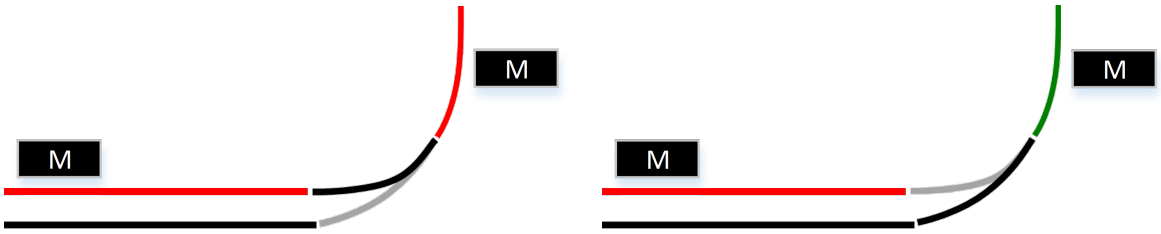
4.3.5. Szakasz engedélyezési protokoll

A modellvasút-terepasztal működésének demonstrálása során el akartam 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 kidolgoztam és megvalósítottam 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, a 4.5 ábrán látható módon. 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.



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

4.5. ábra. Szakasz engedélyezési protokoll

4.3.6. Terminológia definiálása

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

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 megszakítóberendezés	Biztonságkritikus szituáció kialakulása esetén megállítja a vonatokat.

4.1. táblázat. Vasúti terminológia leképzése

Vasúti szakkifejezésben két állomás közti térközök összességét nevezik szakasznak. Ettől eltérően, szakasznak én azt a legkisebb granularitású pályaelemet neveztem, ami nem váltó. Erre azért volt szükség, mert a terepasztalon nincsenek állomások, így nem szükséges a térköz fogalmát definiálni.

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

Azért volt szükség a fogalom két részre bontására, 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 lévő vonatok haladásának engedélyezéséért felelős. Ezáltal a *master* egységeket (biztonsági) vezérlőknek, a *slave* egységeket megszakító-berendezéseknek nevezem.

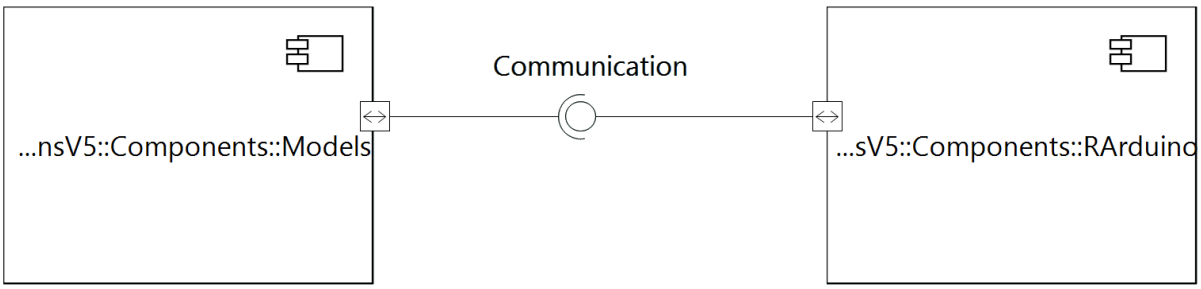
A vezérlő és megszakító-berendezések modellszinten nem kerültek ábrázolásra, hozzájuk a biztonsági logika egy interfészen keresztül kapcsolódik.

4.3.7. Biztonsági logika modell alapú tervezése

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

A korábban a *Mentor Graphics*® által fejlesztett, jelenleg nyílt forráskódú BridgePoint nevű, mérnöki modellek készítésére alkalmas tervezőszoftverben két nagy komponensre bontottam a logikát a 4.6 ábrán látható módon. Az egyik komponens (*Models*) a modelleket, a másik, realizált komponens (*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 foglaltsága, váltó állása), és a modellből a külvilág részére küldött üzenetek (pl. szakaszok letiltása, engedélyezése) továbbítása.



4.6. ábra. Modellezett és nem modellezett komponensek

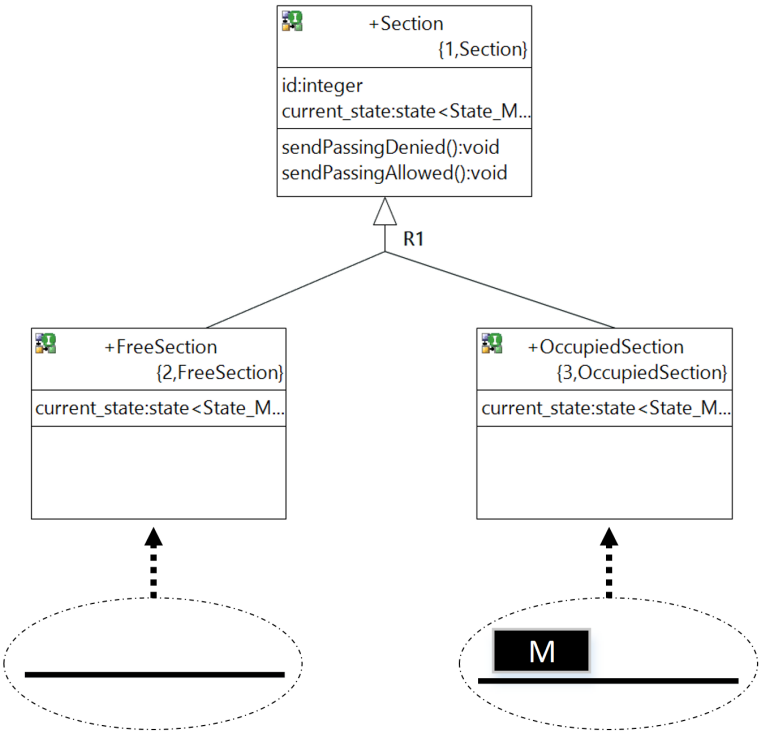
A szakasz lezárási protokollban előforduló fogalmakat a modelleket tartalmazó komponensen belül leképeztem egy osztályhierarchiába. Az egyes osztályokból példányosított objektumok viselkedését állapotgépekben, a protokollbeli üzeneteket a modellekben az egyes osztályok közti jelzésekben (*signalokban*)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

4.3.7.1. Szakasz dekompozíció

Egy modellvasút terepasztali szakaszt a modellben egy egyedileg azonosítható *Section* osztály reprezentál, a 4.7 ábrán látható módon. A szakasz, foglaltsága alapján lehet szabad

(*FreeSection*), vagy foglalt (*OccupiedSection*). Foglalt szakasznak egy olyan szakaszt nevezek, amin legalább egy mozdony tartózkodik.



4.7. ábra. Szakaszok absztrakciója

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

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

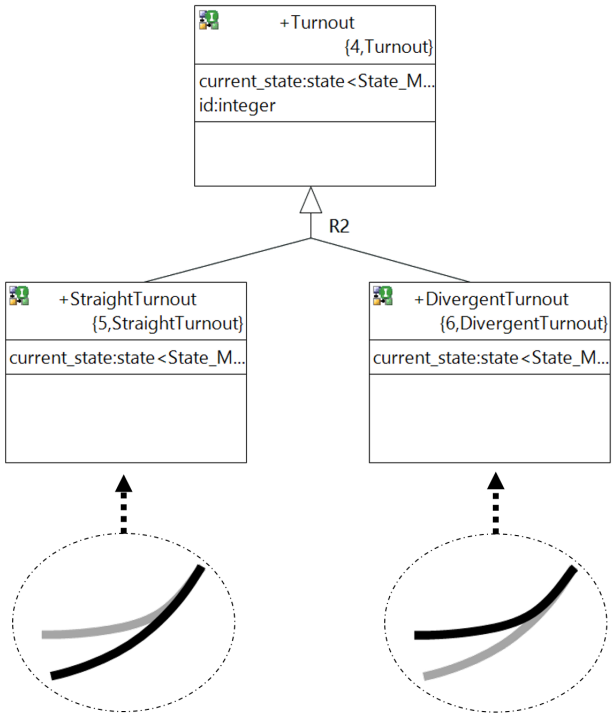
- Ha foglalt, akkor egy jelzést kell küldenie a váltónak a lezárási protokoll elindítására.
- Ha foglalt és egy státusz kérést kap a váltótól a protokoll alapján, akkor vissza kell utasítania, így jelezve, hogy rajta vonat tartózkodik.
- Ha szabad és egy státusz kérést kap a váltótól, akkor engedélyező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

A foglalt szakasz állapotgépe a Függelékek között, az F.2.4 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.

4.3.7.2. Váltó dekompozíció

Egy modellvasút terepasztali váltót a modellben egy egyedileg azonosítható *Turnout* osztály reprezentál, a 4.8 ábrán látható módon. A szakasz lezárási protokoll *lokális döntésének* mo-

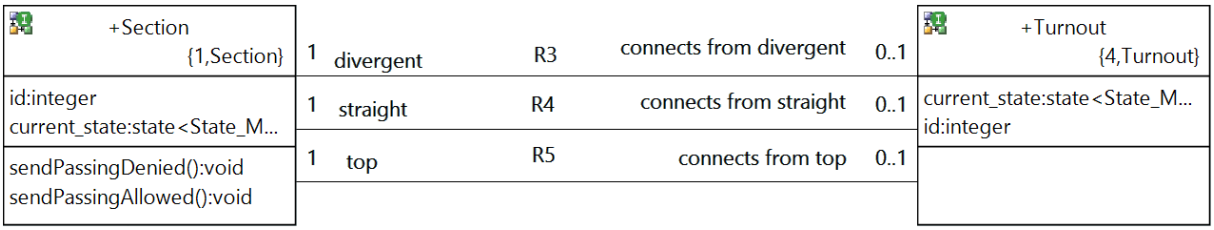
dellezéséhez elegendő a váltó egyenes (*StraightTurnout*), és kitérő (*DivergentTurnout*) állásbeli viselkedését megkülönböztetni, új osztályokra nincsen szükség.



4.8. ábra. Váltó absztrakció

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.



4.9. ábra. Szakaszok és a váltó kapcsolódása

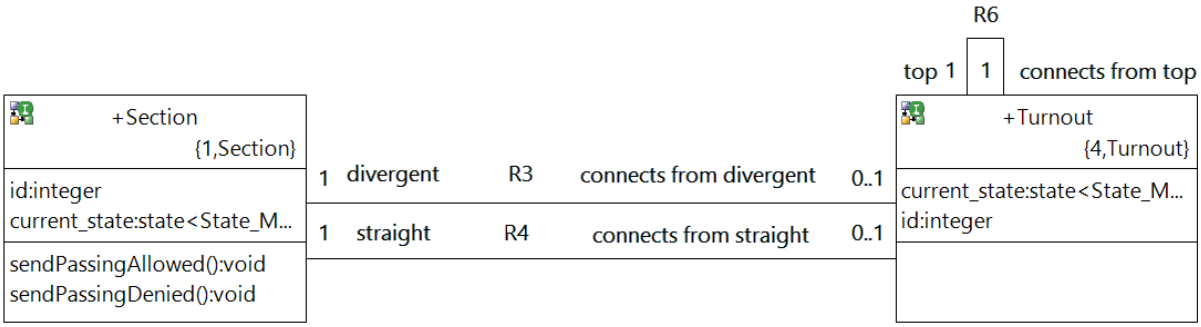
A váltóhoz tartozó szakaszok modellszinten a szakasz váltóhoz való csatlakozási irányától függően tárolódnak a 4.9 ábrán látható módon.

Az egyenes állású váltó lokális döntéseit leíró állapotgép a Függelékek között az F.2.5 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.

4.3.7.3. Angolváltó dekompozíció

Egy modellvasút terepasztali angolváltó a modellben két hagyományos váltó csúcs felőli egymáshoz csatlakoztatásának feleltethető meg a 4.10 ábrán látható módon.

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.



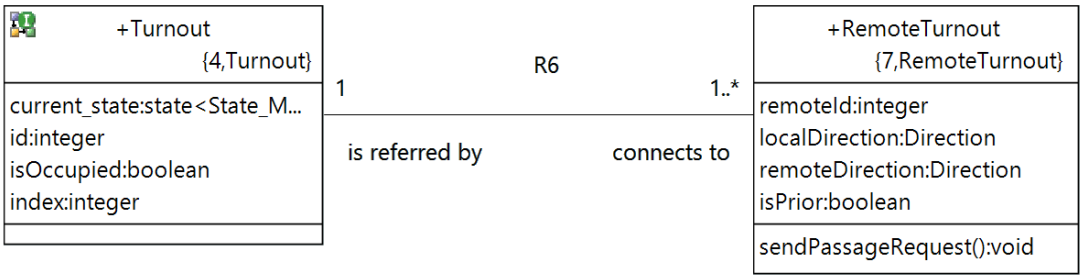
4.10. ábra. Szakaszok és az angolváltó kapcsolódása

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

4.3.7.4. Szakasz lezárási protokoll globális döntése

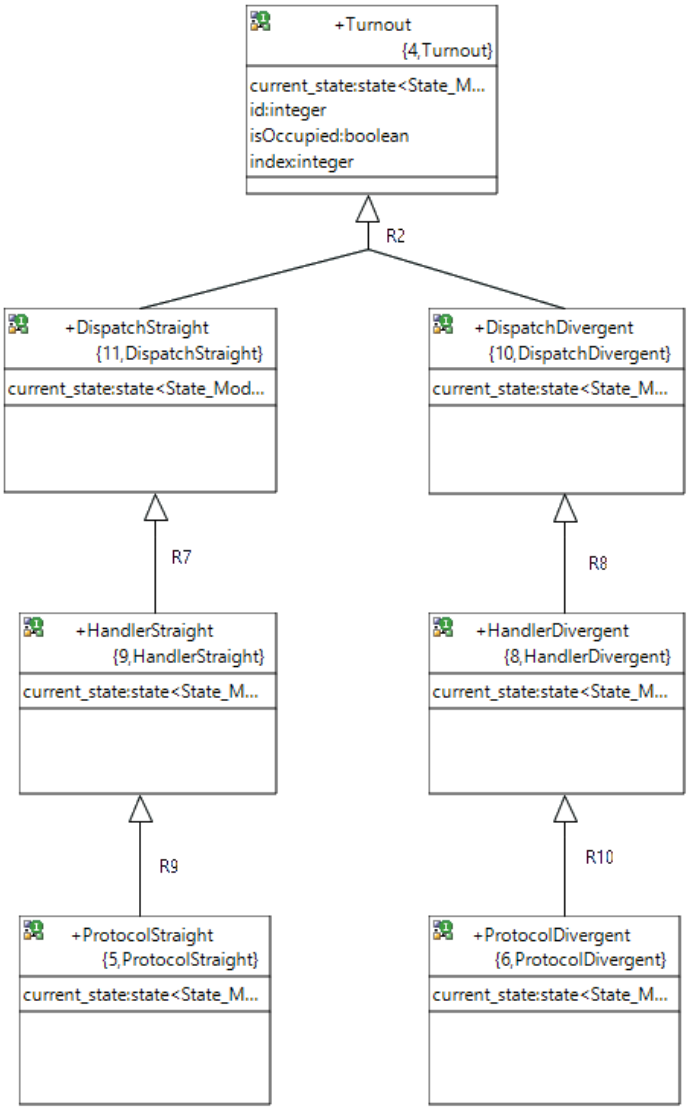
A szakasz lezárási protokoll működésében nagy szerepe van a szakaszfoglalási kérések érkezési irányának. Emiatt a globális döntés meghozása során az egyes lokális váltókra nézve szükséges tárolni, hogy hozzájuk milyen irányból csatlakoznak a szomszédos váltók.

Ezért a modellben minden váltó objektumhoz tartozik egy szomszédos váltót reprezentáló távoli váltó (*RemoteTurnout*) példány – a 4.11 ábrán láthatóan –, melynek segítségével a váltó állapotgépében a szomszédos váltóknak küldött irányfüggő kérések megadhatóak.



4.11. ábra. Távoli váltó és lokális váltó modellszintű kapcsolata

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 4.12 ábra.



4.12. ábra. Globális döntéshez tartozó többszintű osztályhierarchia

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.

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,
- 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.

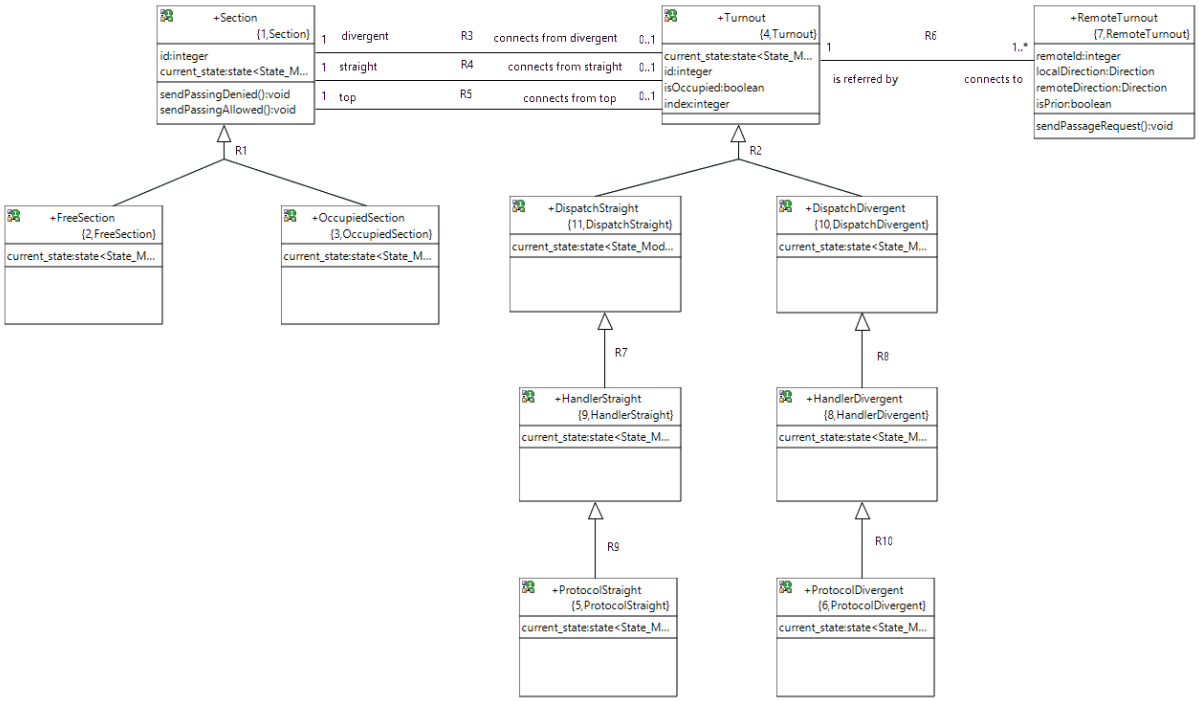
Az egyes szintekhez tartozó állapotgép részletek a Függelékben, az F.2 fejezetben megtalálhatók.

4.3.8. Váltóhoz tartozó biztonsági logika modellszintű kompozíciója

Az előző szakaszokban vázolt modellelemek kompozíciójával létrehoztam 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 4.13 ábrán látható, melyen lévő osztályok állapotgépei közül néhány megtalálható a Függelékek között az F.2 fejezetben.

Az angolváltó modelljének strukturális kompozíciója a hagyományos váltó modelljéhez hasonló, egyedüli különbség a 4.10 ábrán látható váltó (*Turnout*) osztály önmagával vett asszociációja.



4.13. ábra. Hagyományos váltóhoz tartozó modell kompozíciója

A modellezett és a nem modellezett (*master* egységekkel való kommunikációhoz szükséges) komponensek összekapcsolásával megvalósítottam 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 ha-

gyományos váltó és angolváltó esetén megegyezik, míg a lehetséges állapotátmenetek száma a hagyományos váltó esetén magasabb, mint az angolváltónál.

A többi esetben az angolváltónál és a hagyományos váltónál eltérő állapotgépeknél a komplexitás (állapotok, illetve állapotátmenetek száma) az angolváltónál magasabb.

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

4.3.9. Biztonsági logika szimulációja

Az elkészített modelleket az egyes váltókhoz tartozóan specializáltan példányosítva (melyre példa OAL kód a Függelékek között az F.1.1 fejezetben található), a BridgePoint szimulátorában futtatva szimuláltam a biztonsági logika lokális döntésének működését.

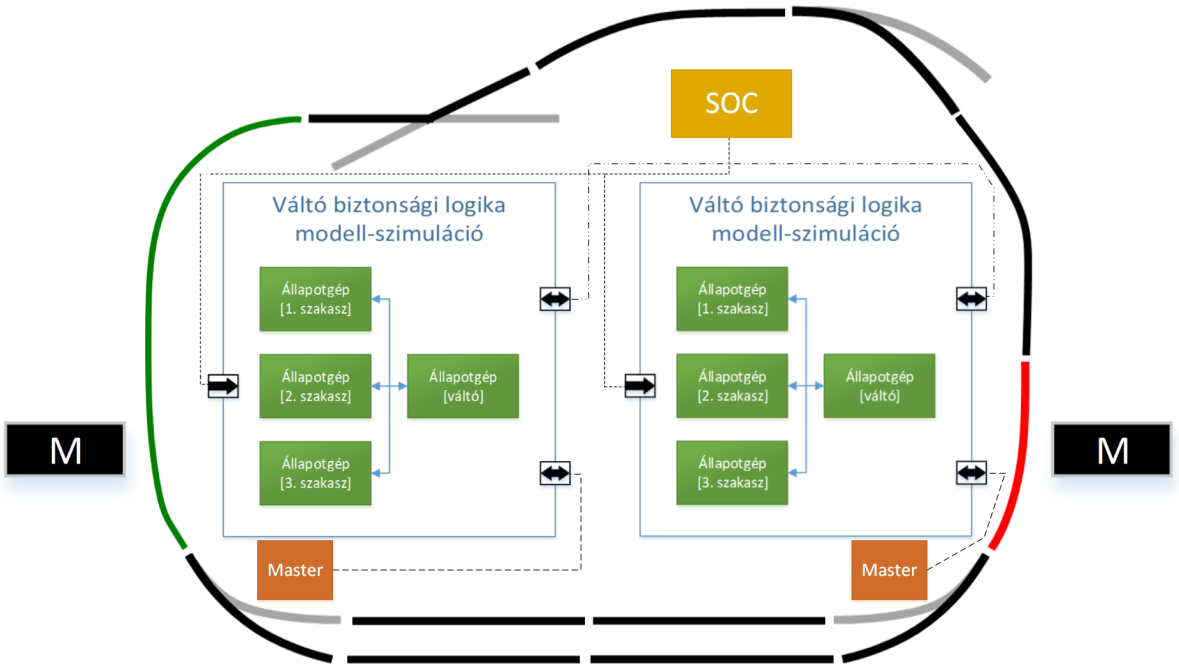
Több példányban futtatva a BridgePoint szimulátorát, az egyes szimulátorokat futtató számítógépek és a terepasztal lokális, vezetékes hálózatának összekapcsolásával az egyes biztonsági logika modellpéldányok egymással és a hozzájuk tartozó *master* egységekkel kommunikálva megvalósítják a biztonsági logika elosztott működését, és a 4.14 ábrán látható módon megakadályozzák a vonatok összeütközését a modellvasút terepasztalon, a 4.3.3 szakaszban írt feltételezések figyelembevételével.

4.4. Rendszerszintű modell értékelése

Az általam megtervezett *szakasz lezárási és engedélyezési protokollt* megvalósító *biztonsági logika* rendszerszintű modellje meglehetősen összetett lett annak ellenére, hogy a komplexitás csökkentése érdekében szeparáltam az egyes egységek felelősségi köreit és feladatait.

Azonban a 4.3.4 szakaszban bemutatott protokoll működéséből adódóan a váltó rendelkezik a legnagyobb felelősséggel a biztonsági logikában. A hozzá tartozó állapotgép kapcsolja össze az egyes szakaszokat, és ő továbbítja állapotától függően a jelzéseket.

Bár az összes lehetséges eseménykombinációt csökkentendő, nem válaszkritikus eseményekre a váltó modellje bizonyos állapotokban nem reagál (eldobja az adott eseményt), illetve az egymás-



4.14. ábra. Elosztott modell alapú biztonsági logika szimulációja

tól függetlenül kezelhető eseményeket (például váltóval ellentétes irányból csatlakozó szakaszról érkező szakasz lezárási kéréseket) külön állapotgépekben kezeltem (lásd a 4.12 ábrán).

Ennek ellenére a szakasz lezárási protokoll *globális* döntését meghozó állapotgépek (*ProtocolStraight*, *ProtocolDivergent*) meglehetősen összetettek lettek, ebből adódóan további szeparációs lépéseket lehet itt még elvégezni.

A modelleket egyrészt ellenőriztem a BridgePoint keretrendszert által biztosított szimulációs környezetben, másrészt formális ellenőrzést is végeztem a modelleken, az 5. fejezetben később bemutatott modelltranszformáció felhasználásával, melynek eredményéről a 6. fejezetben számolok be. Ezáltal többszintű ellenőrizhetőséget definiáltam az xtUML alapú modellekhez.

5. fejezet

Modell alapú szoftverfejlesztés támogatása formális ellenőrzésekkel

Szakdolgozatom egyik feladata az xtUML alapú rendszertervek ellenőrizhetőség szempontjából való vizsgálata volt. Ennek céljából egy olyan folyamatot hoztam létre Konnerth Raimund Andreas hallgatótársammal közösen, amely támogatja a mérnöki modellek formális analízisét ([27]).

Ehhez modelltranszformációt fejlesztettünk, amely a BridgePoint-ban elkészített mérnöki modelleket formális modellekké képzi le. A leképezést meghatározó transzformációs szabályokat és folyamatot én definiáltam, a leképezési folyamat forráskódszintű implementációját Konnerth Raimund Andreas végezte.

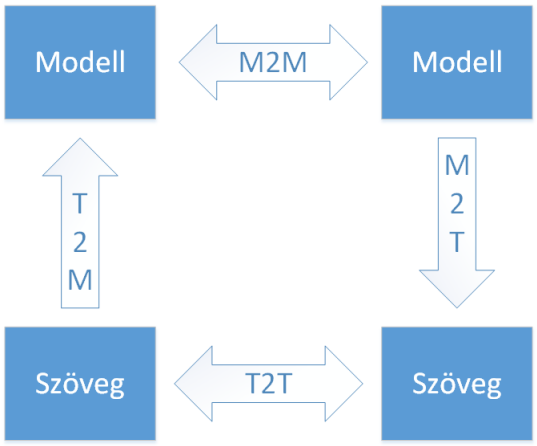
Az 5.1 fejezetben röviden bevezetem a modelltranszformáció fogalmát és változatait. Az 5.2 fejezetben bemutatom a BridgePoint-ban készített mérnöki modellek formális modellekre történő leképezésének folyamatát, a transzformációs szabályokkal együtt, majd egy példán keresztül illusztrálom a működést.

5.1. Modellek transzformációja

Annak érdekében, hogy a magasszintű mérnöki modellek helyességét vizsgálni lehessen, szükség van formális modellek származtatására. A származtatás alatt rendszerint valamilyen transzformáció értendő, ami lehet manuális, részben automatizált vagy teljesen automatizált.

4. Definíció (Modelltranszformáció). *A modelltranszformáció egy olyan folyamat, melynek 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 [28].*

A kiinduló- és a célformátum alapján, az 5.1 ábrán látható módon négy csoportba sorolhatók a modelltranszformációk: modell-modell (M2M), modell-szöveg (M2T), szöveg-szöveg (T2T), illetve szöveg-modell (T2M) közötti transzformáció.

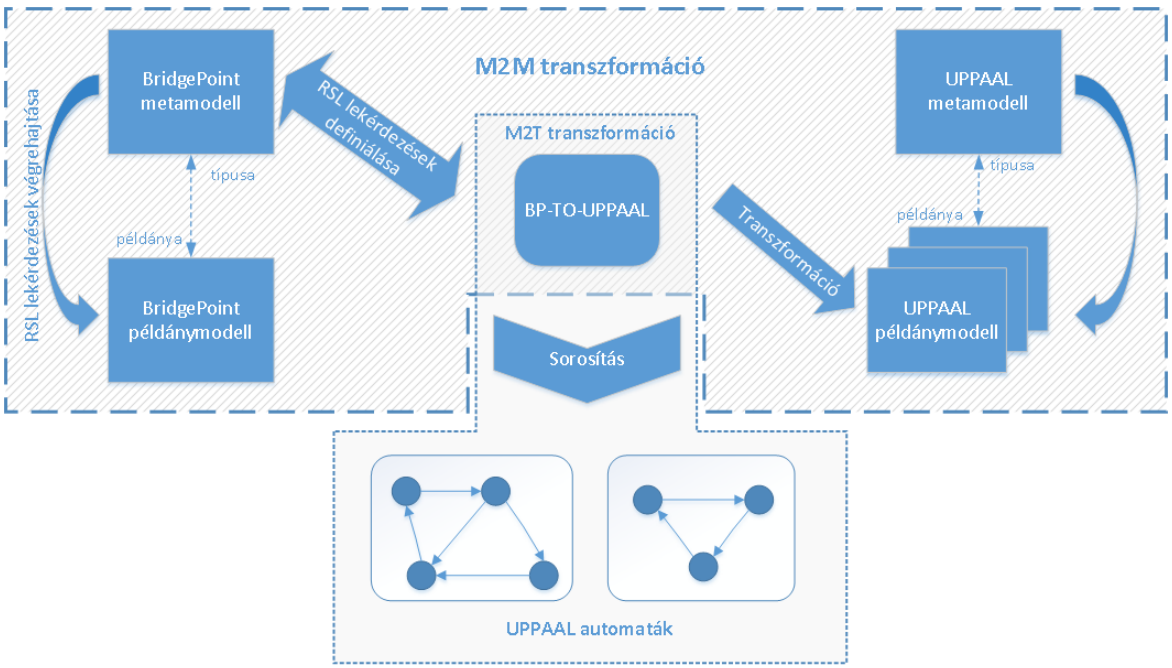


5.1. ábra. Transzformáció típusok

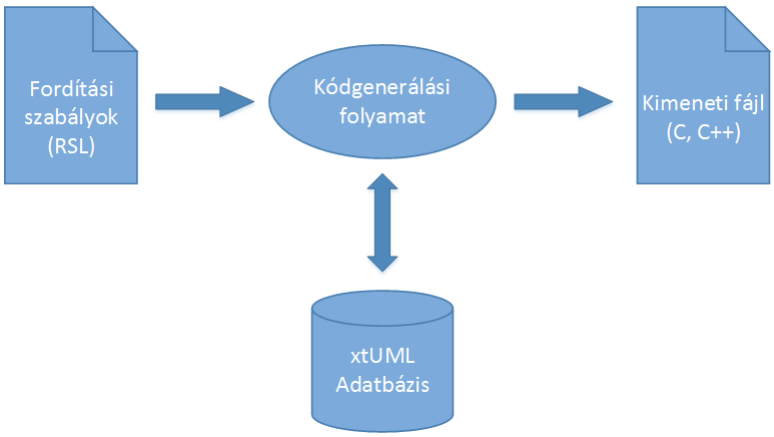
Modell-modell transzformáció esetén a kiinduló- és a célformátumot a modell alapú tervezőeszközök számára értelmezhető modelleknek tekintik. Modell-szöveg transzformációnál a kiindulási modellnek tekintett formátumból egy szöveges reprezentáció készül, ami a nem modell alapú tervezőeszközök számára értelmezhető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

5.2. Mérnöki modellek formális verifikációja

A BridgePoint-ban elkészített mérnöki modellek transzformálása formális modellekre két fő fázisból áll: az első egy modell-modell (M2M) transzformáció, míg a második egy modell-szöveg (M2T) transzformáció. A teljes folyamatot az 5.2 ábra szemlélteti. A formális modellbe való átalakításnak a lényegi része az első fázisban történik.



5.2. ábra. BridgePoint - UPPAAL transzformációs folyamat



5.3. ábra. BridgePoint kódgenerálási folyamata

A BridgePoint-ban elkészített modellből 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. Az 5.3 ábra ezt a folyamatot szemlélteti.

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

- szöveges literálok,
- vezérlőszerkezetek,
- 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 nyerhetőek ki adatok, melyek a kimeneti fájlba írás előtt adott formátumra hozhatóak.

A lekérdezések szintaxisa nagyon hasonló az SQL nyelven írt lekérdezésekhez. Például az 5.1 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 nyelv kifejezőereje nem teszi lehetővé a kódgenerálási folyamatban inkrementalitás alkalmazását.

```
1  .select many objects from instances of O_OBJ    //vezérlőszerkezet
2          .for each object in objects
3                  ...
4          Név             // szöveges literál
5          ${object.Name}  // helyettesítő változó,
6                          // mely kiírja az objektum név attribútumát
7                  ...
8          .end for
```

5.1. kódrészlet. RSL példakód

Ezenkívül a BridgePoint metamodellje 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 – az 5.2 ábrán láthatóan – az 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. 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

A továbbiakban bevezetem 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.

A BridgePoint állapotgépek és az UPPAAL automaták közti modelltranszformációt meghatározó, általam definiált transzformációs szabályok:

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

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

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 kifejezésekből 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.*

$$\frac{\underline{\text{BridgePoint}}}{\begin{array}{c}A, B \in N \\ \exists\, \{A, B\} = e \in E \\ OAL\{e\} = 0 \\ \text{trigger}\end{array}} \quad \Rightarrow \quad \begin{array}{c}\underline{\text{UPPAAL}} \\ A', B' \in L \\ \{A', B'\} = t \in T \\ \text{trigger?}\end{array}$$

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.*

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:*

$$\begin{aligned}\forall\, X \in L\text{−re, ha} \exists\, \{X, A\} &= t \in T \,\text{él} \Rightarrow \{X, A_{kezdő}\} = t \\ \forall\, X \in L\text{−re, ha} \exists\, \{A, X\} &= t \in T \,\text{él} \Rightarrow \{A_{végső}, X\} = \{A, X\} \,\text{és}\, TÖRÖL(\{A, X\})\end{aligned}$$

BridgePoint		UPPAAL
$A, B \in N$		$\exists\, A', q_0, q_1, \ldots, q_{OAL\{e\}}, B' \in L,\ ahol\ q_i \in C$
$\exists\, \{A, B\} = e \in E$		$\{A', q_0\} = t \in T$
$OAL\{e\} > 0$	$\Rightarrow$	$\{q_{i-1}, q_i\} = OAL_i\{e\}, 1 \leq i \leq OAL\{e\}$
		$\{q_{OAL\{e\}}, B'\} = t_0 \in T$
trigger		trigger?
event		event!

5.1. táblázat. 4. transzformációs szabály

Mivel a BridgePoint-ban létrehozott osztályok egyedi azonosítóval rendelkeznek, így a példányokhoz tartozó állapotgépek számossága is legfeljebb egy lesz. Viszont, míg BridgePoint-ban futás közben jöhetnek létre új- és törlődhetnek meglévő objektumok (és így állapotgépek), addig UPPAAL-ban verifikáció közben ez nem tehető meg. Ezért 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 végállapotban van), akkor UPPAAL-ban a neki megfelelő automatában ezt a helyet vissza kell csatolni a kezdőállapotba.*

A fenti szabályokat RSL kifejezésekkel megfogalmazva, majd a Konnerth Raimund Andreas által implementált alkalmazás (BP-TO-UPPAAL) segítségével sikerült automatikusan létrehozni az adott BridgePoint modelleknek megfelelő UPPAAL példánymodelleket, melyekből generáltam az UPPAAL által értelmezhető, automatákat tartalmazó bemeneti fájlt. Az automatákon formális kifejezéseket felírva (lásd 6.2.1 alfejezet) verifikáltam azok működését, ezáltal a kiindulási mérnöki modellek működését is.

Példa: Szabad szakasz állapotgépének transzformációja automatára

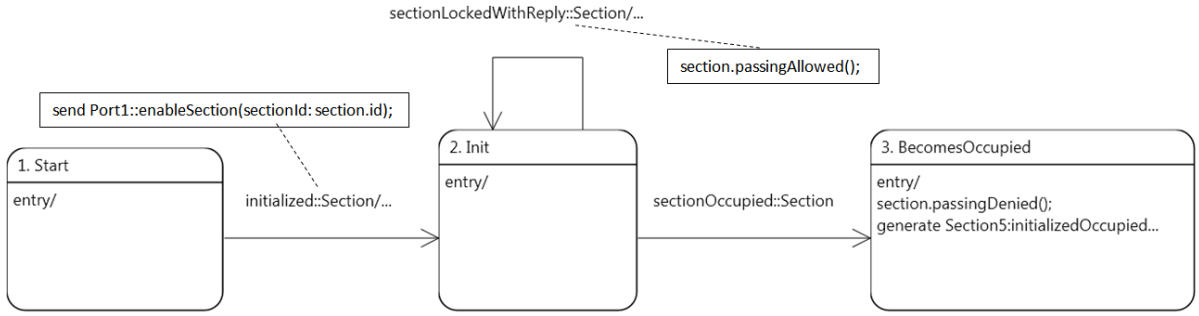
Az 5.4 ábra a fenti szabályok alkalmazását mutatja be a BridgePoint-ban elkészített *szabad* szakasz (*FreeSection*) állapotgépének transzformációjára. A szabad szakasz működése a következő:

A *Start* kezdőállapotból egy *initialized* üzenet hatására az *Init* állapotba kerül, miközben a külső interfészen küld egy *enableSection* üzenetet. Ha a szakasz foglalt lesz, azaz *sectionOccupied* üzenet érkezik, akkor továbblép 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 látható, hogy az *Init* állapothoz tartozik egy hurokél is, mely szintén tartalmaz OAL kódot és a *sectionLockedWithReply* üzenet hatására triggerelődik.

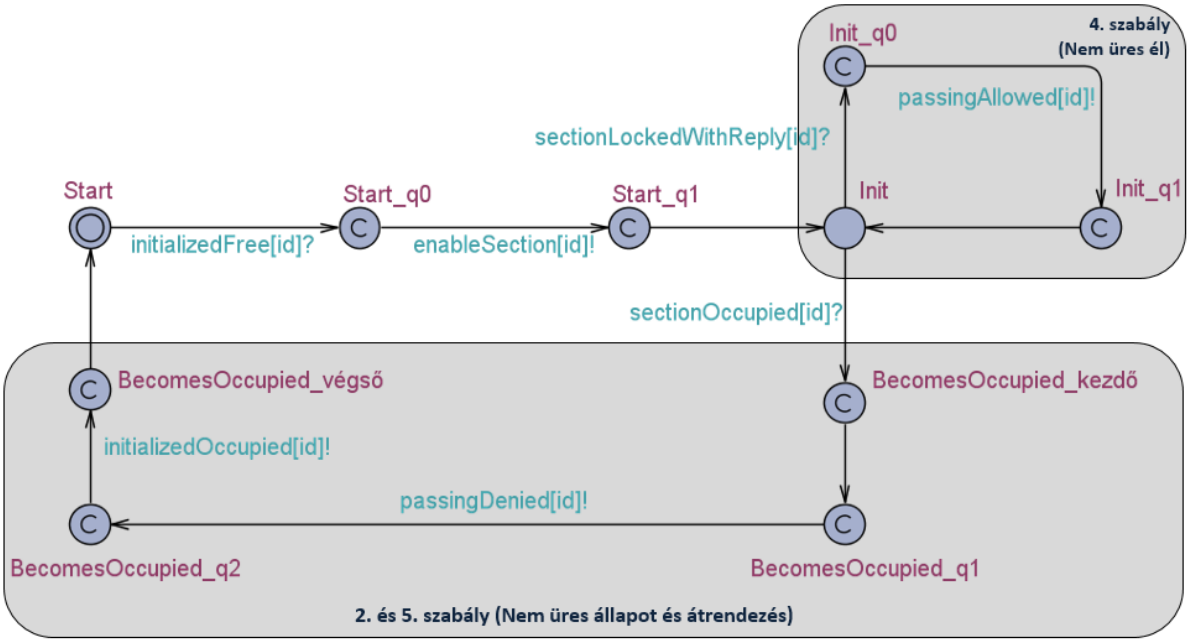
Az alsó ábrán látható a fent definiált szabályok alkalmazásával generált UPPAAL automata. Az alkalmazott szabályok a következők: a *Start* és *Init* állapotok esetén az üres állapotra vonatkozó 1. szabály értelmében megfeleltettem egy-egy helyet. Mivel a két állapot között menő él nem üres, ezért itt a 4. szabályt kell alkalmazni, mely további két *commitált* helyet – *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.



(a) BridgePoint állapotgép



(b) UPPAAL automata

5.4. ábra. BridgePoint állapotgép transzformációja UPPAAL automatára

6. fejezet

Biztonsági logika verifikációja

Annak ellenére, hogy a 4. fejezetben elkészített 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 biztonságkritikus rendszereket nemcsak kizárólag szimulációnak alávetni, hanem formálisan is verifikálni.

Ebből a célból szakdolgozatom egyik feladataként formálisan ellenőriztem az 5.2 fejezetben ábrázolt transzformációs folyamat alapján előállított UPPAAL automatákat, ezáltal ellenőrzést adva az xtUML módszertan szerint megvalósított rendszertervek helyességére vonatkozóan.

A 6.1 szakaszban röviden bemutatom a formális verifikáció folyamatát, a 6.2.1 szakaszban beszámolok az elért verifikációs eredményekről, végül a további vizsgálati lehetőségeket a 7. fejezetben mutatom be.

6.1. Formális verifikáció folyamata

A formális verifikáció során a verifikálni kívánt modellekre temporális logikát felhasználva meghatároznak követelményeket, majd az összes lehetséges esetet (állapotkonfigurációt) megvizsgálva ellenőrzik ezek 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.

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.

6.2. Biztonsági logika állapotgépeinek formális verifikációja

A BridgePoint-ban elkészített, xtUML alapú állapotgépekből származtatott UPPAAL automaták állapotainak számát a 6.1 táblázat mutatja. A transzformáció során létrejött automaták közül néhány a Függelékek között, az F.3 fejezetben megtekinthető.

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

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

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

modell áll elő. Továbbá a keletkezett automaták kompozíciója esetén az elosztott rendszerek modellezésére jellemzően könnyen előkerülhet az *állapottér robbanás* problémája is, amely az állapotok számával exponenciálisan növekvő formális állapottér bejárását jelenti.

Az általam elkészített biztonsági logika formális ellenőrzését három fázisra bontottam:

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.

Dolgozatomban a biztonsági logika ellenőrzésének első fázisát, egy váltó lokális döntésének verifikációját végeztem el a 6.2.1 fejezetben leírtak szerint.

6.2.1. Lokális döntések 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 az UPPAAL-ban szükség van a *környezeti modellnek* megfelelő automatára, amely biztosítja az automaták működéséhez szükséges információkat.

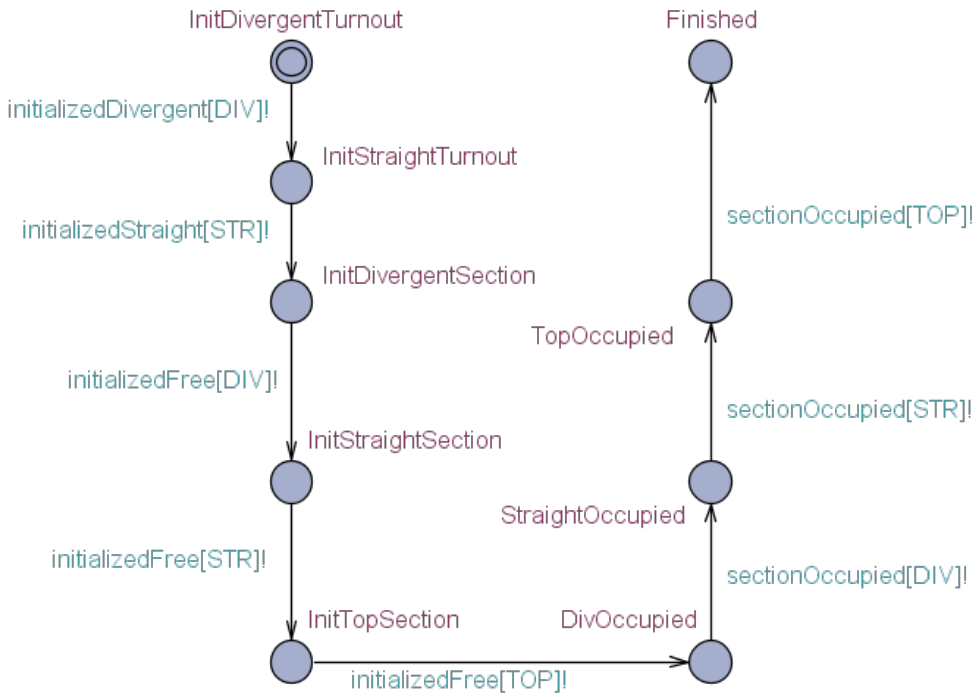
A 6.2. alfejezetben említett első fázis ellenőrzéséhez UPPAAL-ban, a 6.1 ábrán látható környezeti modellt készítettem el. Az automata viselkedést tekintve először inicializálja a váltóállást[1] és a váltórészeket, majd a kitérő irányból, az egyenes irányból, és a váltócsúcs felől csatlakozó szakaszoknak (*Section*) küld egy-egy *foglalt* jelzést.

Egy váltó lokális döntésének verifikációhoz az alábbi öt esetet különböztettem meg, melyekhez tartozó követelményeket CTL kifejezésekként fogalmaztam 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.

   Ez a valóságban annak a veszélyes szituációnak felel meg, amikor a kitérő- és a váltócsúcs

[1] A két váltóállás közül mindig csak az adott esetnek megfelelő részek lesznek inicializálva.



6.1. ábra. Környezeti modell automatája

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 kitérő ághoz 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, különben 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 verifikációs eredményeket a 6.2 táblázat foglalja össze, melyben feltüntettem az egyes esetek verifikációja során használt memóriát és a verifikáció futásának időtartamát is. A verifikációt egy HP EliteBook Folio 9470m, Intel Core i5 1.8 GHz-es processzorral és 8 GB DDR3 típusú memóriával rendelkező számítógépen végeztem.

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	✓

6.2. táblázat. Verifikációs eredmények lokális döntésre

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

7. fejezet

Összefoglalás és jövőbeli tervek

Szakdolgozatomban több gyakorlati és elméleti eredményt is elértem, ezeket a 7.1 és a 7.2 szakaszokban tekintem át. Munkámat a 2014-es TDK konferencia beágyazott rendszerek szekciójában, hallgatótársaimmal (Mázló Zsolt, Konnerth Raimund Andreas) közösen bemutattuk [27].

7.1. Gyakorlati eredmények

- *Biztonsági logika megtervezése*: Az xtUML módszertant egy iparilag releváns modellalapú tervezőrendszerben használva modellalapon kidolgoztam és megvalósítottam a tanszéki demonstrátor szoftver (biztonsági logika) komponenseit.
- *Eszközök integrációja*: A megvalósított modelleket egy modellszimulációs környezetben futtatva, interfészeken keresztül integráltam a modellvasút terepasztal elosztott hálózatához, ezáltal a terepasztalon közlekedő vonatok haladásába közvetlen befolyást szerzett a biztonsági logika, és veszélyes szituációk esetén megakadályozta a vonatok összeütközését.

7.2. Elméleti eredmények

- *xtUML nyelv bemutatása*: Dolgozatom elején bevezetést adtam a modellvezérelt szoftver- és rendszerfejlesztés folyamatába. Ezen belül áttekintettem és bemutattam az Executable and Translatable UML nyelv és módszer elemkészletét, amely segítséget nyújthat rendszerek modell alapú tervezése során.
- *xtUML alapú tervezés értékelése*: Dolgozatomban az xtUML nyelv, illetve a biztonsági logika bemutatása után értékeltem a módszertan alkalmazhatóságát, illetve a módszertan alapján készített rendszerterv és modellek komplexitását.
- *Mérnöki modellek transzformációja*: Szakdolgozatomban sikerült egy iparilag releváns mérnöki modellek tervezésére használt alkalmazás modelljeit formális modellekre transzformálni automatikusan, melynek során modelltranszformációs szabályokat határoztam meg.
- *Formális modellellenőrzés*: Sikeresen verifikáltam egy iparilag releváns modellellenőrző keretrendszer segítségével az elkészült modellek közül az egy váltó lokális döntéséért fele-

lős állapotgépeket. Így a modellszimuláción kívül egy újabb ellenőrizhetőséget adtam az xtUML alapú rendszertervekhez.

7.3. Jövőbeli tervek

Természetesen az általam elkészített megoldások nem teljesek, számos továbbfejlesztési lehetőség létezik, melyek közül néhányat az alábbiakban sorolok fel.

- *Verifikáció kiterjesztése*: A formális ellenőrzés során, csak az egy váltó lokális döntésében együttműködő állapotgépek működését verifikáltam. Azonban a biztonsági logika számos egyéb komponensből áll, melyek helyes működésének bizonyítása alapvető biztonsági kérdést jelent a terepasztalon közlekedő vonatok biztosítása szempontjából.
  - *Globális döntések ellenőrzése*: A váltók globális döntéseinek verifikációja az előzőekhez képest kibővített, elosztott modell szerinti új állapotgépek leképzését, valamint bonyolultabb környezeti modellek megalkotását igényli.
  - *Kommunikációk ellenőrzése*: A jelenleg használt kommunikációs protokollok formális ellenőrzésével felderíthető, hogy azokban hol lehetnek hibák, majd ezek javításával tovább növelhető a rendszer megbízhatósága.
- *Modellek egyszerűsítése*: A biztonsági logika globális döntéséért felelős állapotgépek jelenleg bonyolult felépítésűek, ami jelentősen megnehezíti a helyes működésük bizonyítását. Ennek következtében célszerű ezeket több részre dekomponálni, ezáltal könnyebben áttekinthetőek és verifikálhatóak lesznek.
- *Kódgenerátorok alkalmazása*: A modellszimulációs környezet használata helyett célszerűbb az elkészített modellekből a viselkedésüket megvalósító, platformspecifikus forráskódot automatizáltan származtatni, ezáltal a szimulációs környezet használata nélkül, közvetlenül a beágyazott rendszereken futtathatóvá válnának a modellek, így a biztonsági logika is.
- *Transzformált elemek nyomonkövethetősége*: A megvalósított modelltranszformáció 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 közvetlenül visszavezethetőek.

Ábrák jegyzéke

2.1. Modell alapú rendszertervezés Y-modell 9
2.2. Modellezési nyelvek fejlődése 11
2.3. xtUML modellekből forráskód származtatása 13
2.4. Komponens diagram példa 14
2.5. Polimorf eseménykezelés példa 14
2.6. UPPAAL-ban használt CTL operátorok 18

3.1. Használati eset diagram példa 21
3.2. Aktivitás diagram példa 21
3.3. Szekvencia diagram példa 22
3.4. Komponens modell és interfész példa 23
3.5. Osztálydiagram és állapotgép példa 24
3.6. xtUML modellekből forráskód származtatása 25

4.1. Beavatkozó egységek elhelyezkedése a hálózatban 28
4.2. A hálózat összeköttetése a BridgePoint állapotgépekkel 29
4.3. Szakasz lezárási protokoll lokális döntés 32
4.4. Szakasz lezárási protokoll globális döntés 34
4.5. Szakasz engedélyezési protokoll 35
4.6. Modellezett és nem modellezett komponensek 36
4.7. Szakaszok absztrakciója 37
4.8. Váltó absztrakció 38
4.9. Szakaszok és a váltó kapcsolódása 38
4.10. Szakaszok és az angolváltó kapcsolódása 39
4.11. Távoli váltó és lokális váltó modellszintű kapcsolata 39
4.12. Globális döntéshez tartozó többszintű osztályhierarchia 40
4.13. Hagyományos váltóhoz tartozó modell kompozíciója 41
4.14. Elosztott modell alapú biztonsági logika szimulációja 43

5.1. Transzformáció típusok 45
5.2. BridgePoint - UPPAAL transzformációs folyamat 45
5.3. BridgePoint kódgenerálási folyamata 46
5.4. BridgePoint állapotgép transzformációja UPPAAL automatára 50

6.1. Környezeti modell automatája 53

F.2.1 DispatchStraight állapotgép 62
F.2.2 HandlerStraight állapotgép 63
F.2.3 ProtocolStraight állapotgép 64
F.2.4 Foglalt szakasz állapotgép 65
F.2.5 Egyenes állású váltó állapotgép 66
F.3.1 Szabad szakasz automata 67
F.3.2 Foglalt szakasz automata 68
F.3.3 Egyenes állású váltó automata 69

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] S Flint and C Boughton. Executable/Translatable UML and Systems Engineering. *Practical Approaches for Complex Systems (SETE 2003)*, 2003.

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

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

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

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

[9] V tervezési modell. URL: `http://www.waterfall-model.com/v-model-waterfall-model/`, hozzáférés: 2014. december 9.

[10] Object Management Group. *OMG Unified Modeling Language (OMG UML) - Version 2.4.1*, 2011. URL: `http://www.omg.org/spec/UML/2.4.1/`, hozzáférés: 2014. december 9.

[11] Object Management Group. *OMG Systems Modeling Language (OMG SysML) - Version 1.3*, 2012. URL: `http://www.omg.org/spec/SysML/1.3/`, hozzáférés: 2014. december 9.

[12] Object Management Group. *OMG Object Constraint Language (OMG OCL) - Version 2.3.1*, 2012. URL: `http://www.omg.org/spec/OCL/2.3.1/`, hozzáférés: 2014. december 9.

[13] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models (fUML) - Version 1.1*, 2013. URL: http://www.omg.org/spec/FUML/, hozzáférés: 2014. december 9.

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

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

[16] Soley, Richard and others. Model Driven Architecture. *OMG white paper*, 308:308, 2000.

[17] Selo Sulistyo and Warsun Najib. Executable uml. *Dept of Information and Communications Technology Agder University College, Norway*, 2002.

[18] 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.

[19] xtUML homepage, 2014. URL: http://www.xtuml.org/, hozzáférés: 2014. december 9.

[20] Object Action Language Reference Manual. URL: http://www.ooatool.com/docs/OAL08.pdf, hozzáférés: 2014. december 9.

[21] Object Management Group. *Model Driven Architecture Overview*, 2014. URL: http://www.omg.org/mda/specs.htm, hozzáférés: 2014. december 9.

[22] UPPAAL homepage, 2014. URL: http://www.uppaal.org/, hozzáférés: 2014. december 9.

[23] 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, hozzáférés: 2014. december 9.

[24] DCC protokoll. URL: http://www.opendcc.de/info/dcc/dcc.html, hozzáférés: 2014. december 9.

[25] Asymmetric Break Control. URL: http://www.tonystrains.com/technews/lenz-asy-abc.htm, hozzáférés: 2014. december 9.

[26] Arduino. Arduino Uno specifikáció weboldal. URL: http://arduino.cc/en/Main/ArduinoBoardUno, hozzáférés: 2014. december 9.

[27] Horváth Benedek, Konnerth Raimund Andreas, Mázló Zsolt. *Elosztott biztonságkritikus rendszerek modellvezérelt fejlesztése*, 2014. BME TDK konferencia, Beágyazott rendszerek szekció. URL: http://tdk.bme.hu/VIK/DownloadPaper/Elosztott-biztonsagkritikus-rendszerek, hozzáférés: 2014. december 9.

[28] 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.

Függelék

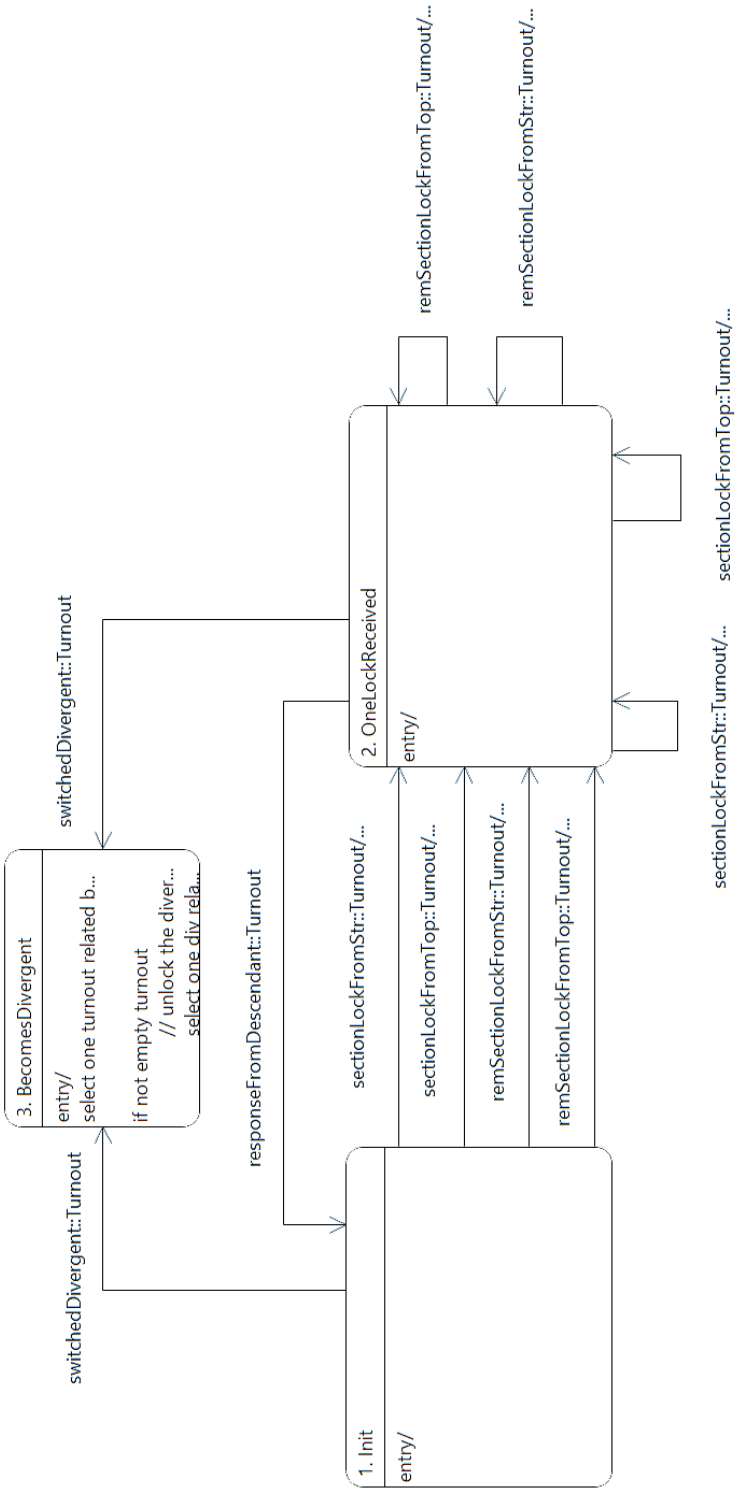
F.1. Modellek példányosítása

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

F.1.1. kódrészlet. Váltóhoz tartozó modellek példányosítása

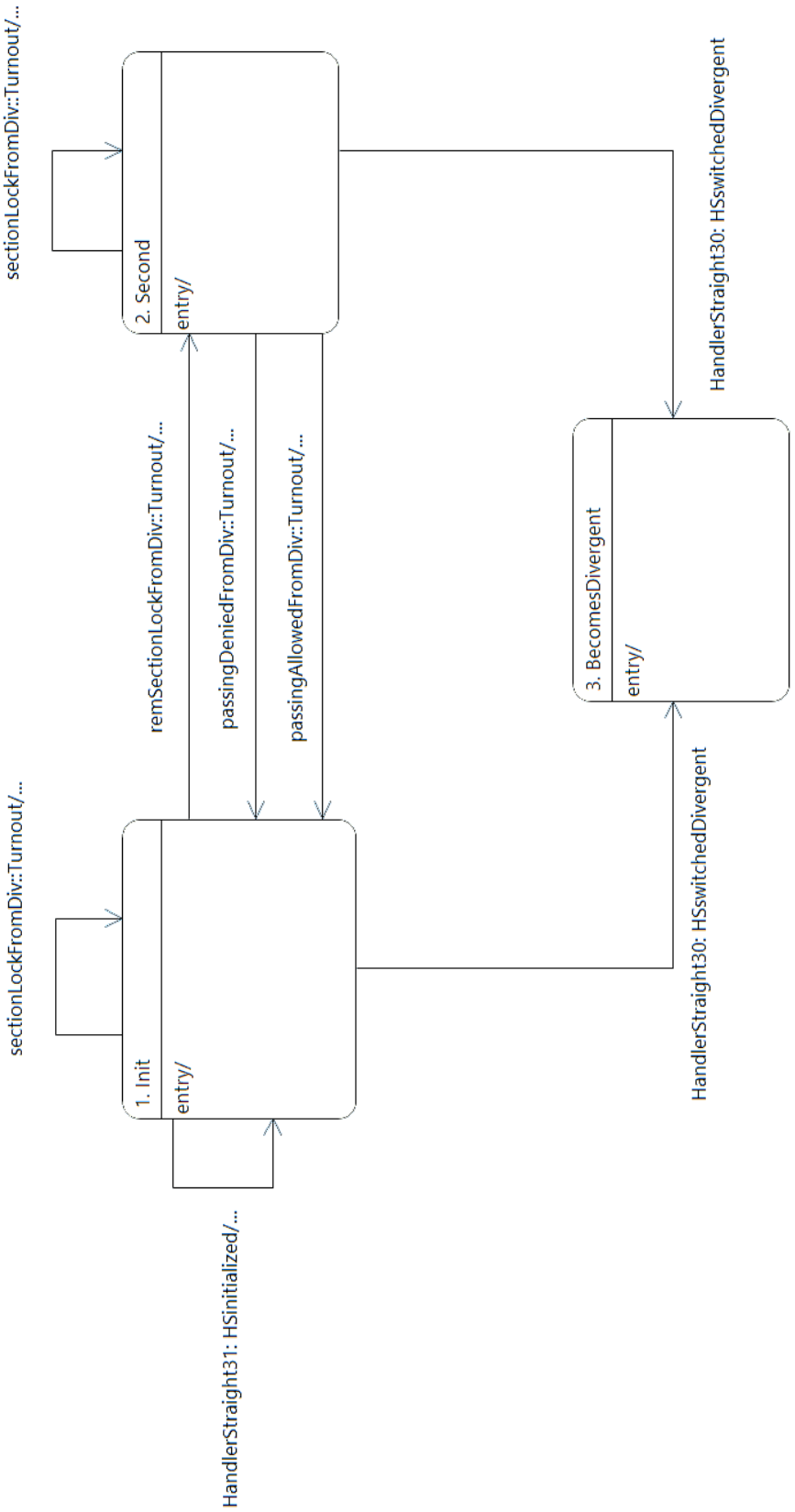
F.2. Állapotgép részletek

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.



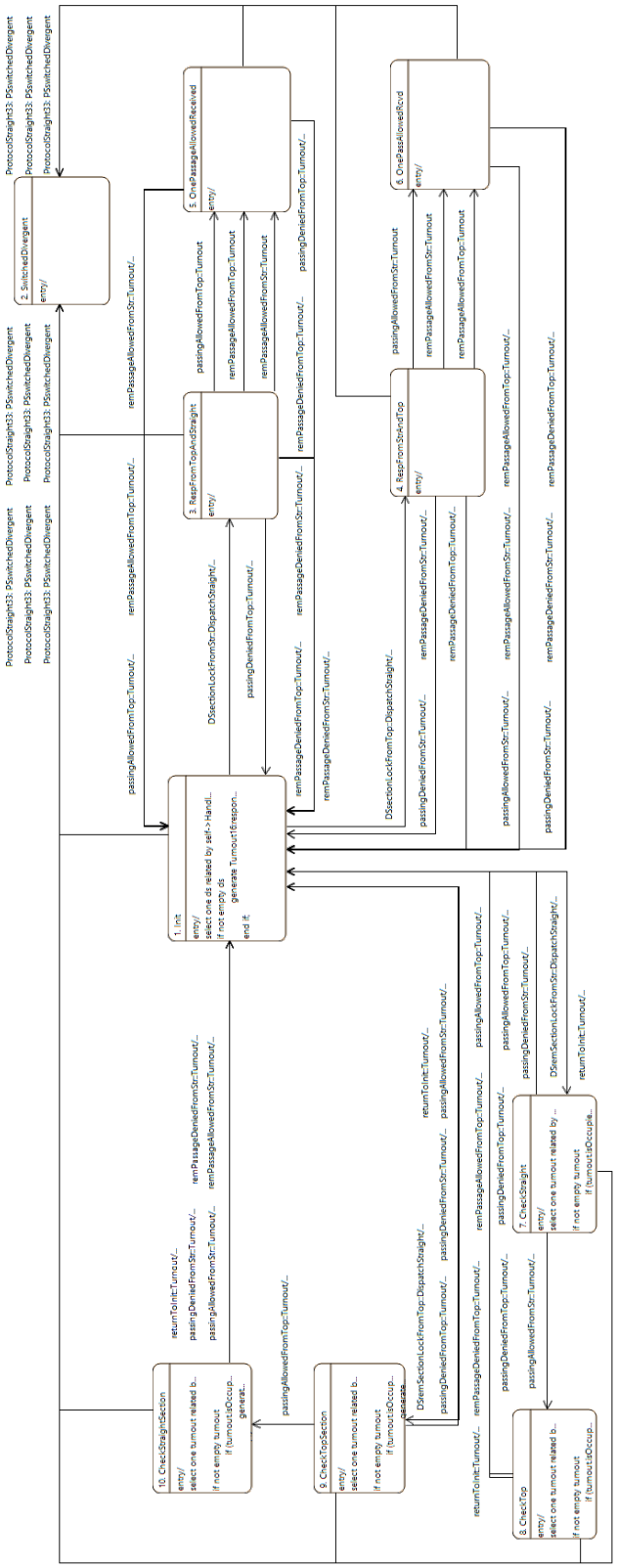
F.2.1. ábra. DispatchStraight állapotgép

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



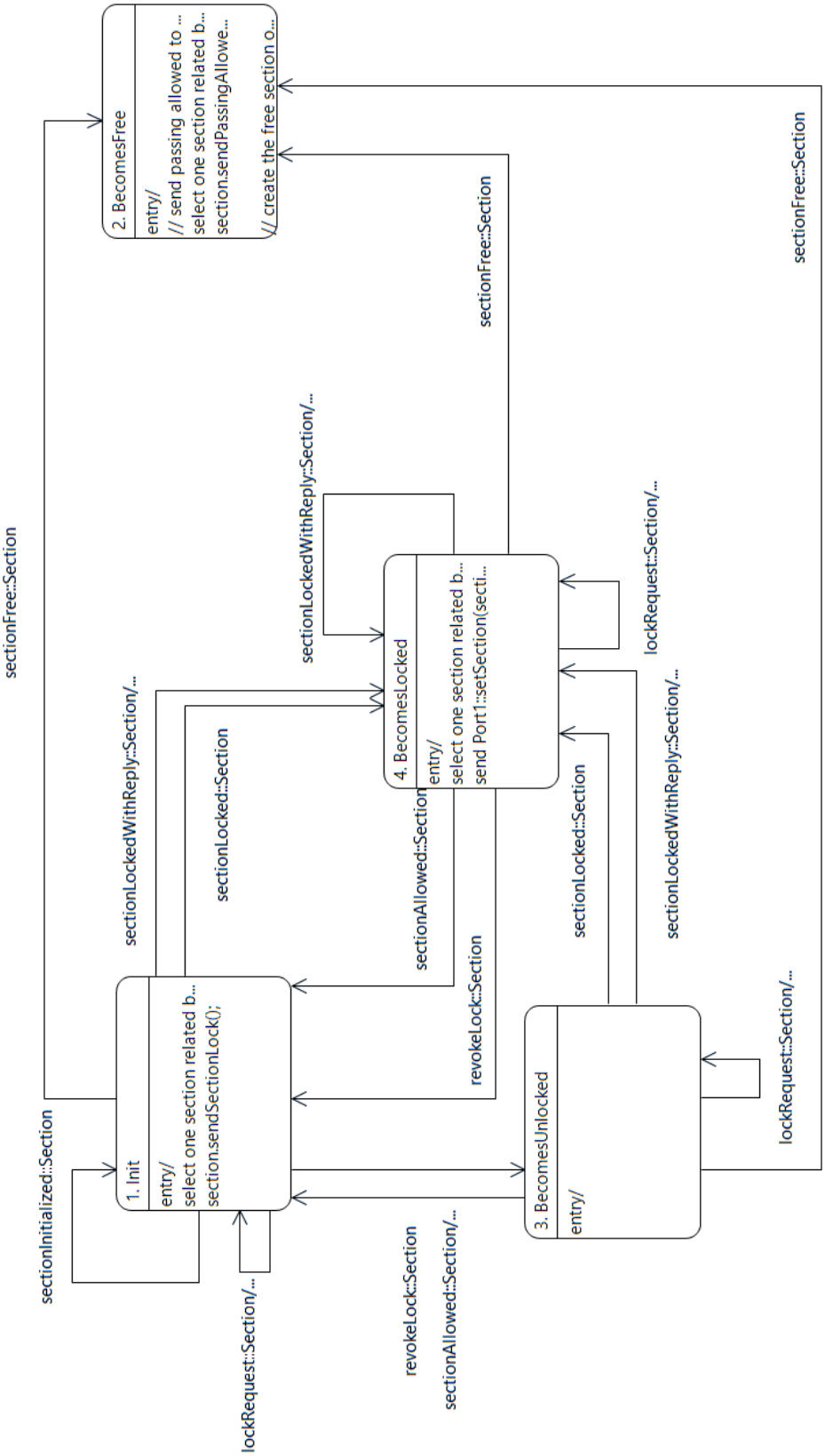
F.2.2. ábra. HandlerStraight állapotgép

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



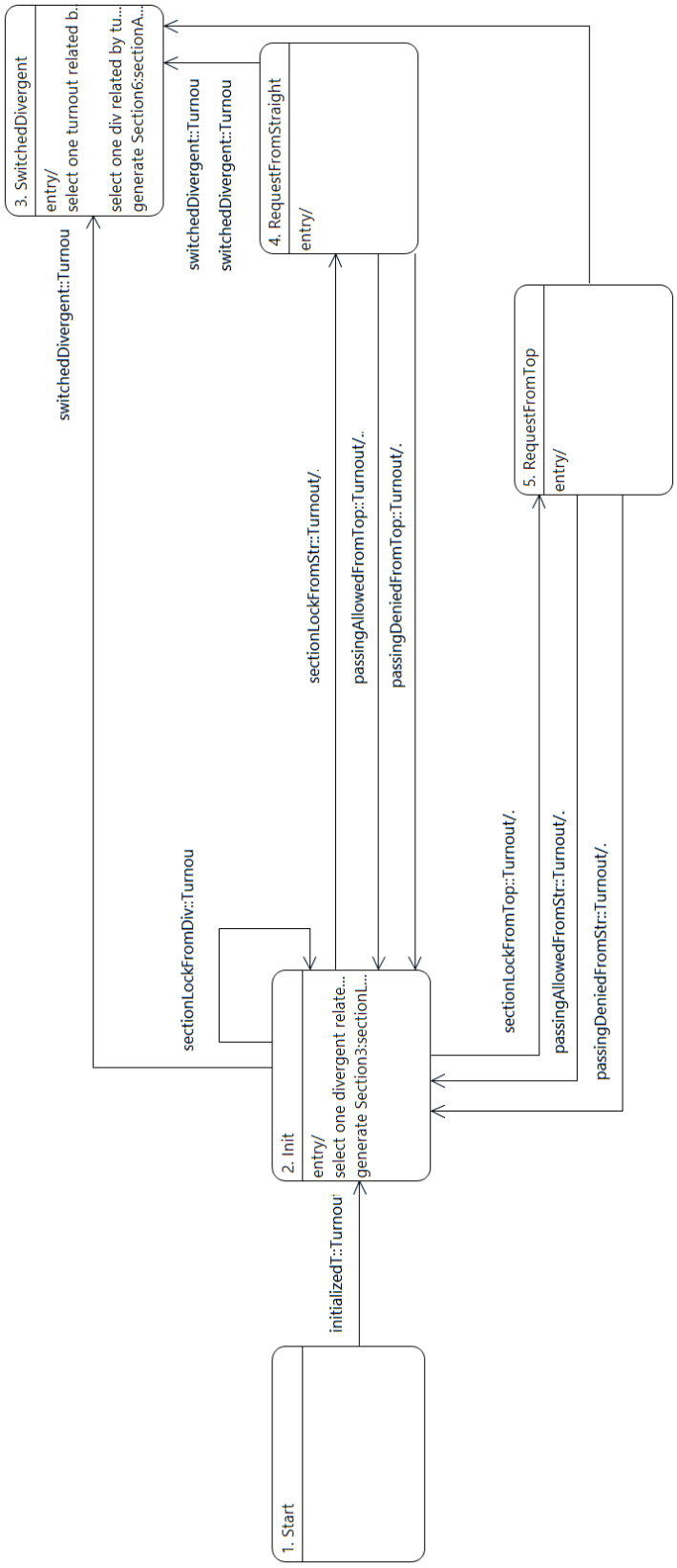
F.2.3. ábra. ProtocolStraight állapotgép

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



F.2.4. ábra. Foglalt szakasz állapotgép

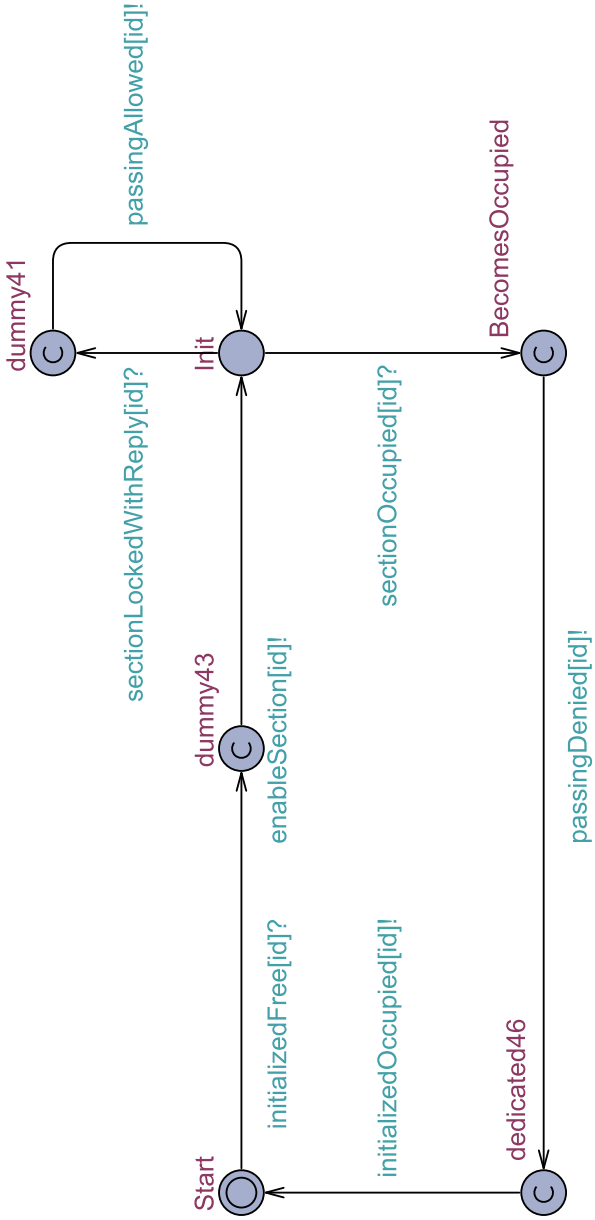
A *StraightTurnout* osztályhoz tartozó példányszintű állapotgép, melynek feladata a hagyományos váltóhoz tartozó, lokális döntést tartalmazó viselkedés megvalósítása.



F.2.5. ábra. Egyenes állású váltó állapotgép

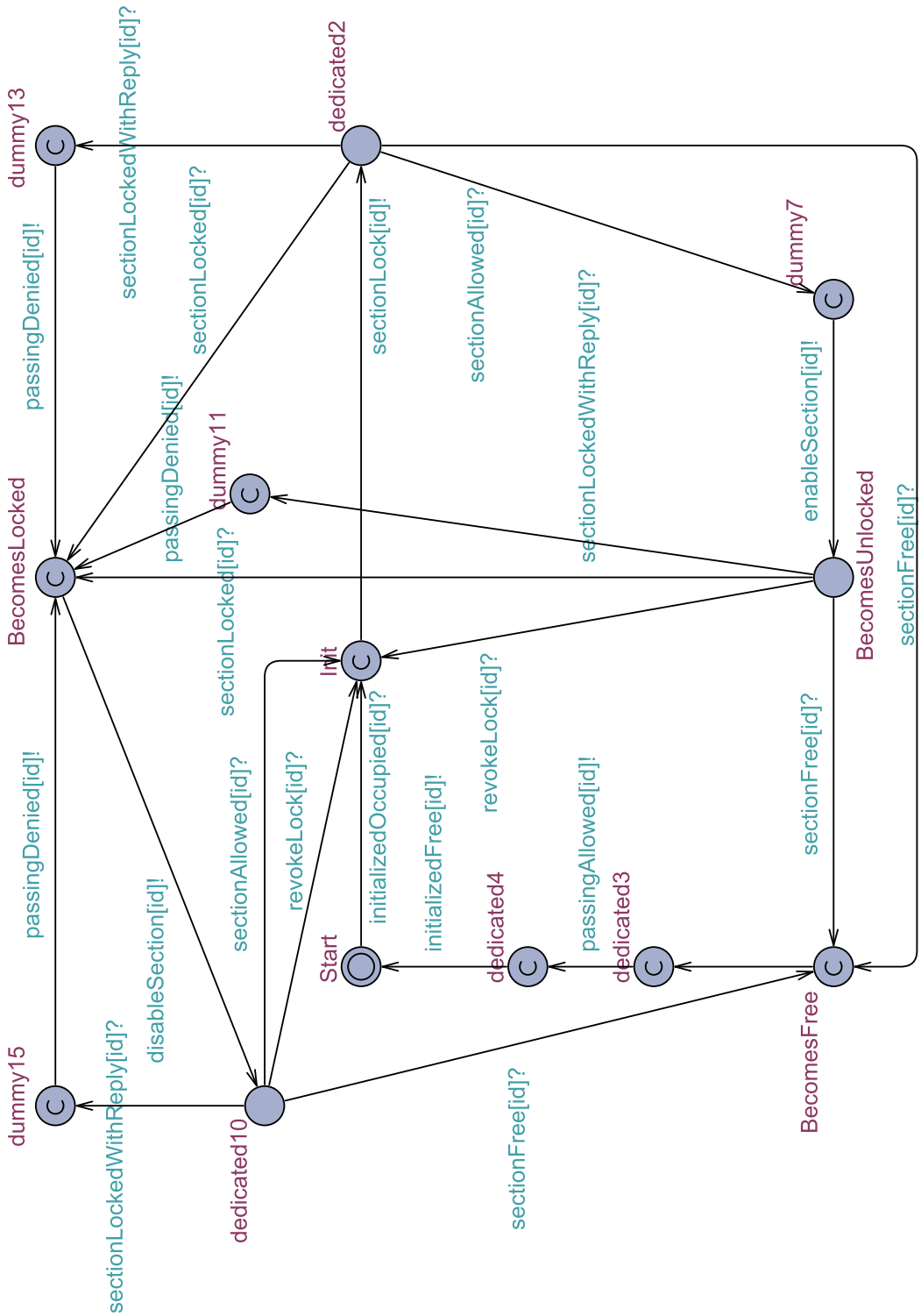
F.3. UPPAAL automata részletek

A *FreeSection* állapotgéphez tartozó formális automata, melynek feladata a szabad szakasz viselkedésének formalizálása.



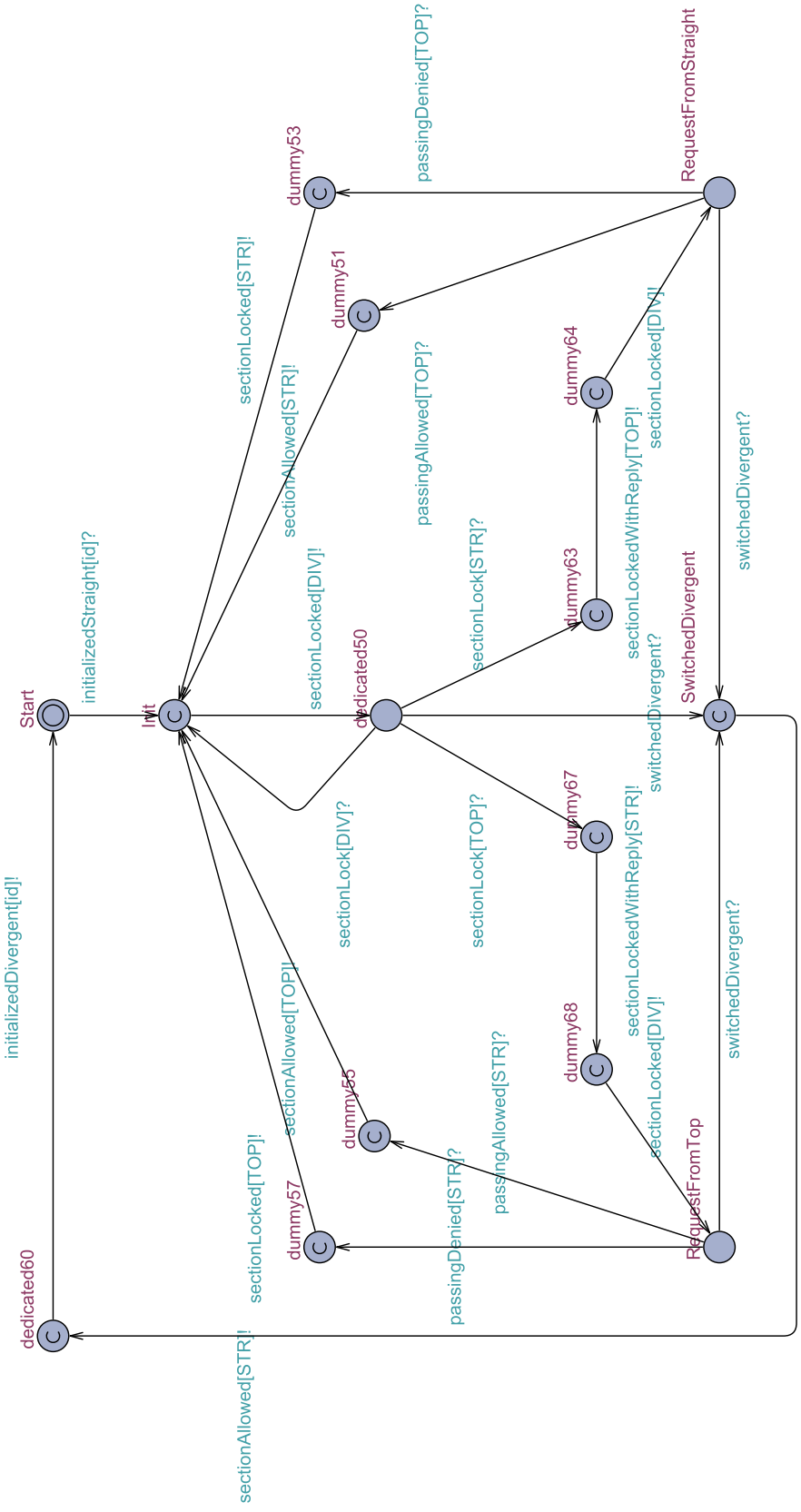
F.3.1. ábra. Szabad szakasz automata

Az *OccupiedSection* állapotgéphez tartozó formális automata, melynek feladata a foglalt szakasz viselkedésének formalizálása.



F.3.2. ábra. Foglalt szakasz automata

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



F.3.3. ábra. Egyenes állású váltó automata