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

Absztrakt tesztelrendezések generálása autonóm járművek szisztematikus ellenőrzéséhez

Szakdolgozat

Készítette
Ficsor Attila

Konzulens
dr. Semeráth Oszkár

2019. december 17.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Ficsor Attila*, 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, 2019. december 17.

Ficsor Attila
hallgató

Kivonat

Napjainkban egyre inkább elterjedőben van a mesterséges intelligencián alapuló komponensek használata autonóm járművek fejlesztése során. Autóktól kezdve villamoson át targonca gépekig számos területen használnak vagy terveznek használni önműködő eszközöket. Ezeket a gépeket olyan autonóm komponensek vezérlik, melyek szenzorok által gyűjtött adatok alapján hoznak döntéseket.

Ezen mesterséges intelligencia komponensek helyes működése biztonsági szempontból kritikus, hiszen hiba esetén jelentős anyagi kár keletkezhet, vagy emberéletek kerülhetnek veszélybe. Emiatt ezeknek a rendszereknek szigorú biztonsági előírásoknak kell megfelelniük. Működésük tesztelése, ellenőrzése viszont kihívást jelent, amelyre jelenleg nem ismertek hatékony és automatizált módszerek.

Dolgozatom célja egy olyan automatikus tesztelrendezés generátor megvalósítása, amely hatékonyan képes támogatni autonóm járművek szisztematikus tesztelését. Célom továbbá, hogy a különböző előállított szituációk valóban lényeges különbségeket tartalmazzanak, ezzel garantálva a tesztesetek megfelelő fedettségét.

A szcenáriókat egy kétlépéses generálás segítségével hozzuk létre. Első lépésben absztrakt szcenáriókat generálunk, melyben a környezet elemeinek, valamint a teszteset szereplőinek egymáshoz viszonyított helyzetét adjuk meg. Ezután koordinátákat rendelünk az egyes elemekhez, amivel konkrét, szimulálható szituációkat állítunk elő. A két lépés közül én az elsőt valósítottam meg. Az így előálló tesztelrendezések kipróbálhatóak automatizált szimulátorban, vagy megépíthetőek tesztszobában. Dolgozatomban elért eredményeket egy esettanulmányon szemléltetem.

Megoldásom segítségével automatikusan hozhatók létre új szituációk, amelyekben az autonóm jármű MI komponensét szimulátoros környezetben tesztelhetjük. A modell absztrakciójának köszönhetően az általam generált különböző szituációk lényegi, logikai különbségeket tartalmaznak, amely garantálja a tesztkészlet diverzitását, és mérhetővé teszi a tesztkészlet fedettségét.

Abstract

Nowadays components based on artificial intelligence-based are more and more commonly used for developing autonomous vehicles. From cars to trams and forklifts, many applications already exist and use autonomous components, and even more is under development. These vehicles are controlled by such autonomous components, that make decisions using the data collected by their sensors.

The correct behaviour of these components using artificial intelligence is critical from a safety with respect to safety and reliability, because in case of an error, major property damage can occur, or human lives can be in danger. Therefore, automotive industry specifies strict safety standards, which is challenging to satisfy. However, state-of-the-art testing techniques were unable to support the automated testing and verification of these components are not known.

My goal is to implement such automatic test layout generator, that can efficiently support the systematic verification of autonomous vehicles. My goal is also that, different test layouts should contain major differences from each other, guaranteeing a level of test coverage.

The proposed solution is a two-step generation process. In the first step we generate abstract scenarios, in which we describe the relative state of the elements of the environment and the actors of the test. After that we assign coordinates to the elements, which results in a concrete scenario, that can be executed in a simulator. In this report I implement the first step. The test layouts produced this way can be tested in an automatized simulator or can be built in a test room. In the report the achieved results will be illustrated in a case study.

With the help of my solution, new situations can be automatically created, in which the autonomous vehicle's AI components can be tested in a simulated environment. Thanks to the abstraction of the model the different generated scenarios contain major logical differences, which guarantees the diversity of the test set and makes the coverage measurable.

1. fejezet

Bevezetés

1.1. Kritikus autonóm rendszerek

A mindennapi életünkben már most is találkozhatunk kritikus autonóm rendszerekkel, és ez a közeljövőben várhatóan még gyakrabban elő fog fordulni [3]. Ezek a rendszerek leggyakrabban valamilyen járművek, például autók vagy villamosok, és jellemzőjük, hogy mesterséges intelligencián alapuló komponensek vezérlik őket.

1.2. Autonóm komponensek helyességvizsgálata

Az autonóm komponensek hibás működése esetén jelentős anyagi kár keletkezhet, vagy emberéletek kerülhetnek veszélybe, éppen ezért ezeknek a rendszereknek szigorú biztonsági előírásoknak kell megfelelniük. Azonban működésük tesztelése, ellenőrzése kihívást jelent [10, 14, 13], amelyre jelenleg nem ismertek hatékony és automatizált módszerek. A tesztelés kihívásai közé tartozik, hogy a tesztesetek nem állnak rendelkezésre, kiváltképp különleges diverz esetek, amik a legveszélyesebbek. Ezen kívül nincs fedettség metrika, a tesztelés drága szimulátorban is, tesztszobában pedig még inkább.

A működés helyességének vizsgálata több módon végezhető, például szimulátoros környezetben, tesztpályán vagy közutakon. Utóbbi két megoldás segíthet megtalálni a szenzorok gyenge pontjait, azokat a környezeti tényezőket, melyek negatívan befolyásolják a rendszer képességét a helyzet felismerésére. A szimulátoros tesztelés során a szenzorok megfelelő működését feltételezve lehet vizsgálni a mesterséges intelligencián alapuló komponensek döntéseinek helyességét. Ehhez azonban szükséges egy tesztkészlet, mely tartalmazza az összes lehetséges tesztelrendezést, hiszen csak így tudjuk biztosítani, hogy valóban minden szituációban az elvárt döntést hozza a rendszer.

1.3. Autonóm komponensek szisztematikus tesztelése

A munkám elsődleges célja egy olyan módszer kidolgozása, amely lehetőséget biztosít fizikai tesztelrendezések automatikus és szisztematikus generálására. Az algoritmus kimenete absztrakt tesztelrendezések halmaza, amik koordináták hozzárendelése után felhasználhatóak kritikus rendszerek autonóm komponenseinek szimulátoros teszteléséhez.

1.4. Tesztelrendezések automatikus generálása

A dolgozatban bemutatok egy megközelítést, ami lehetőséget biztosít tesztelrendezések automatikus generálására. A módszer szakterület-specifikus nyelvekre épülő gráfgenerátorokat használ, ami miatt a tesztesetek hatékonyan paraméterezhetőek metamodellekkel és

jólformáltsági kényszerekkel. A modellelemek koordinátáit numerikus megoldókkal oldjuk fel. A bemutatott módszerrel képesek vagyunk akár az összes lehetséges tesztelrendezés generálására, és biztosítani tudjuk, hogy az egyes elrendezések lényegi különbségeket tartalmaznak, így valóban vizsgálható válik a tesztfedettség.

Az elrendezéseket két lépésben generáljuk. Első lépésben a metamodell, valamint a jólformáltsági és strukturális kényszerek alapján állítunk elő absztrakt elrendezéseket. Ezekben az elemek között csupán geometria viszonyok jelennek meg. Ezután ezeket az absztrakt elrendezéseket konkretizáljuk, a geometriai viszonyok alapján rendelünk az elemekhez koordinátákat. Ezek közül én az első lépést valósítom meg.

A javasolt megközelítést egy esettanulmányon mutatom be, melyben autonóm autók tesztelrendezéseit vizsgálom az Open Autonomous Safety szabvány alapján. Hasonló módon elkészíthető egy szakterület-specifikus nyelv például villamosok, vagy akár más kritikus rendszerek leírásához is.

1.5. Dolgozat felépítése

A dolgozat hátralévő része a következőképpen épül fel. A 2. fejezetben összefoglalom a munkámhoz szükséges legfontosabb technikai és elméleti háttértudást. A 3. fejezetben egy rövid áttekintést nyújtok a javasolt megoldás felépítéséről és működéséről. Ezután a 4. fejezet az absztrakt elrendezések generálásának módszerét mutatja be, majd az 5. fejezetben megvizsgálom a megoldás skálázhatóságát. Végül a 6. fejezetben összefoglalom a munkámat.

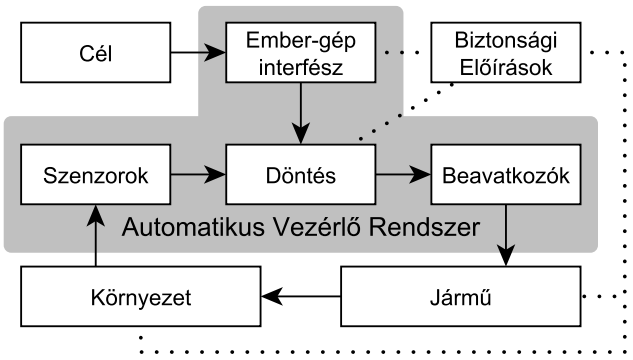
A szakdolgozatom egy TDK dolgozatra [5] épül, melyben a bemutattuk a kétlépéses generálást, az ebben a dolgozatban szereplő absztrakt elrendezések készítését, valamint az ezekhez történő koordináta hozzárendelés folyamatát is.

2. fejezet

Előismeretek

2.1. Autonóm járművek biztonsági előírásai

Az autonóm komponenseket tartalmazó járművek alapvető felépítését, valamint a velük szemben támasztott követelményeket [2], [12] és [16] alapján tárgyalom. Az automatikus vezérlő rendszer (automated driving system, ADS) egy komplex környezetben üzemel, ez láthat a 2.1 ábrán. A rendszer különféle szenzorokkal (kamera, radar, LIDAR, stb.) érzékeli a környezetét. Az autonóm rendszer célját a jármű vezetője vagy utasa határozza meg egy ember-gép interfészen keresztül. Ezután az ADS hoz döntéseket a megfigyelt szituáció alapján és beavatkozókon keresztül vezérli a járművet.



2.1. ábra. Autonóm jármű komponensei

Az autonóm járművek biztonságos működésére vonatkozó követelményeket [2] alapján öt kategóriába sorolhatjuk (ahol az egyes szintek feltételezik az előző szint biztonságát):

1. *Jármű stabilitás*: Elsőként a jármű stabil irányítása szükséges. Az autó feletti kontroll elvesztése következtében a jármű a közlekedés többi résztvevőjével ütközhet, vagy lesodródhat az útról.

2. *Biztonságos távolság tartása* (Assured clear distance ahead, ACDA): Az ACDA a jármű előtti távolság, aminek beláthatónak kell lennie és nem lehet rajta akadály, ami ahhoz szükséges, hogy az autonóm jármű meg tudjon állni.

3. *Minimum távolság*: Az autonóm rendszernek biztosítania kell egy minimum távolságot a jármű és más mozgó és statikus testek (pl. autók, fák) között.

4. *Közlekedési szabályok*: A közlekedési szabályok formális, többnyire biztonság növelését célzó szabályok, melyeket jogilag tartatnak be egy földrajzi területen.

5. *Bevett gyakorlatok* (Driving best practices): A bevált gyakorlatok informális közlekedési szabályok, amik finomítják és kiegészítik a formális szabályokat. Erre példa az, hogy milyen hamar jelezzük a kanyarodásunkat az irányjelzővel, vagy hogyan reagálunk, ha a mögöttünk haladó szűk követési távolságot tart.

Amíg számos létező kutatás a 1-es és 2-es követelményeket vizsgálja [16], én feltételezem, hogy azokat már kielégítették, és [16]-ban felvázolt elképzelésnek megfelően a rendszer-szintű 3-as és 4-es követelményekre koncentrálok. Ezen felül feltételezem, hogy a szenzoros érzékelés és a beavatkozók megbízhatóan működnek. Az általam bemutatott megközelítés elsődleges célja olyan tesztkészlet előállítása, amely olyan hibákat is észre tud venni amik a szenzorok helyes működése mellett következtek be. Végül feltételezem, hogy a követelmények biztosítását elsődlegesen teszteléssel és szimulációval ellenőrzik.

2.2. Open Autonomous Safety alapítvány Scenario Testing eszköztára

A Voyage nevű, autonóm járműveket fejlesztő cég hozta létre az Open Autonomous Safety [23] alapítványt, melyen keresztül szabadon hozzáférhetővé tették az általuk kidolgozott módszereket autonóm járművek tesztelésére. Eredeti terveik szerint szeretnék, ha ez a könyvtár iparági standardként működne. A dokumentum három részből áll, melyek az autonóm működés különböző aspektusaira koncentrálnak:

- **Szcenáriótesztelés** (Scenario Testing): biztonságkritikus szcenáriók leírása, melyeket tesztelni kell a biztonságos működés biztosításához.
- Funkcionális Biztonság (Functional Safety): biztonsági és funkcionális követelmények felírásának és vizsgálatának folyamata.
- Autonómia Értékelés (Autonomy Assessment): a járművek autonómiáját beavatkozások alapján mérő metrikák.

Munkám során a szcenáriótesztelés részt használtam fel, ezért ezt részletezem az alábbiakban, viszont a teljesség kedvéért a többi részt is összefoglalom.

2.2.1. Szcenárió tesztelés

Ez a dokumentum olyan tesztelrendezéseket mutat be, melyekkel egy autonóm jármű találkozhat az utakon közlekedve. Mindegyik szcenárió részletesen leírja a jármű elvárt viselkedését, melyet követve biztosítható a biztonságos közlekedés. A tesztelt autonóm járműre "Ego" néven hivatkozik, ebben a fejezetben én is ezt ez elnevezést használom. Az elrendezések biztonságkritikus szcenáriókat írnak le, melyeket mindenképpen tesztelni kell annak érdekében, hogy az autonóm jármű biztonságosan tudjon navigálni környezetében.

OAS Szcenárió A szcenáriók egy vagy több lépésből állnak. Mindegyik lépéshez tartozik egy kép, egy leírás és az elvárt viselkedés.

- Kép: A világ kezdeti állapota, a modell leírása. Mutatja a tesztelrendezés elemeit, azok pozícióját, az Ego jármű és a többi aktor pozícióját, valamint hogy az aktorok hogyan mozognak.
- Szcenárió Leírás: Az aktorok mozgását írja le az adott lépésben. Az Ego járműre ez csak az első lépésben adja meg a kiindulási állapotot, amit a képen is láthatunk. A többi aktor mozgását minden lépésben leírja. Amennyiben a teszt során ezen feltételek nem teljesülnek, a tesztet rosszul végeztük el.

- Elvárt Eredmény: Ego elvárt viselkedését adja meg. Amennyiben a tesztelt jármű nem ennek megfelelően viselkedik, hibát találtunk a működésében.

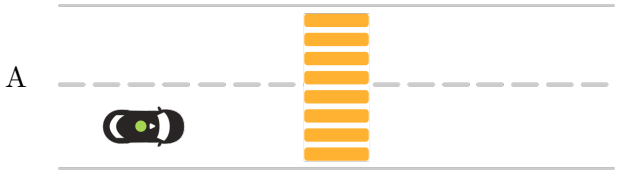
Ha a szcenárió egy lépése az elvárt eredménnyel zárult, a teszt a következő lépéssel folytatódik. Az összes lépés teljesítése esetén a tesztet sikeresnek tekintjük.

Szcenárió ID Minden szcenárióhoz egy azonosítót rendelhetünk, mely a következő elemekből áll:

útszakaszok-sávok-stoptáblák-szcenárió kategória-Ego cselekvése-többi aktor

Nézzük meg az azonosító részeinek jelentését a **2-2-XX-CW-STR-XX** azonosítójú szcenárión, mely az egyik legegyszerűbb az OAS eszköztárában és a 2.2 ábrán látható.

Az azonosító első eleme az útszakaszok számát jelöli, a gyalogátkelő bal és jobb oldalán elhelyezkedő egy-egy útszakasz miatt ennek értéke **2**. A második **2** érték azt jelenti, hogy az út két sávos és a sávok között nem található elválasztó vagy járdasziget. Az elrendezésben nincsenek stoptáblák, ezt az azonosító harmadik elemének **XX** értéke jelzi. A **CW** a szcenárió kategóriáját jelöli, jelentése gyalogátkelőhely, a rövidítés az angol crosswalk szóból származik. A vizsgált jármű egyenesen halad az úton, ezt az **STR** azonosítóval jelölhetjük. A szcenárióban az Ego-n kívül nincs másik aktor, melyet az azonosító végén **XX** értékkel jelölünk.



Szcenárió
Ego közeledik egy gyalogátkelőhöz. A gyalogátkelőn és annak közelében nincs gyalogos
Várt eredmény
Ego áthalad a gyalogátkelőn.

2.2. ábra. 2-2-XX-CW-STR-XX azonosítójú szcenárió az OAS eszköztárában

A szcenáriók azonosítójának különböző részei csak előre meghatározott értékeket vehetnek fel, melyek a következők:

- **Útszakaszok:** Az útszakaszok száma (kereszteződés ágai)

  2 szakaszból álló (egyenes) út 2, 3 irányú kereszteződés 3, 4 irányú kereszteződés 4, n/a (pl. zsákutca esetén használandó) XX, stb.

- **Sávok száma:** A tesztelrendezésben található útszakaszok sávjainak száma és hogy van-e az út közepén elválasztó/sziget

  1, 2, 3, 4, 2 sáv + elválasztó 2M, 4 sáv + elválasztó 4M, 1 sáv + sziget 1I, stb.

- **Stoptáblák:** A szcenáriót felülről nézve melyik irányban lévő útszakaszokon találhatóak stoptáblák (a kereszteződés irányába vezető sávokon).

  Észak N, Kelet E, Dél S, Nyugat W, nincs XX, stb.

- **Szcenárió kategória:** A szcenárió kategóriabesorolása

  Autó követése (Car following) CF, gyalogátkelő (crosswalk) CW, zsákutca (cul-de-sac, tipikusan amerikai stílusú, kör alakú fordulóval rendelkező zsákutca) CDS, kereszteződés (intersection) I, gyalogos az úton (pedestrian in road) PIR, tolató jármű (reversing vehicle) RV, sebességkorlátozás (speed limit) SL, jármű az úton (vehicle in roadway) VR, stb.

- **Ego cselekvés:** Az Ego célját írja le

  Egyenesen halad STR, balra kanyarodik L, jobbra kanyarodik R, megfordul U, stb.

- **Többi aktor:** A szcenárióban résztvevő többi aktor és mozgása **aktor : aktor kezdeti pozíciója > végső pozíciója : aktor cselekvése** formában.

  Egynél több aktor is szerepelhet egy szcenárióban, ebben az esetben a további aktorok az szcenárió ID végén helyezkednek el. Sorrendjüket az alapján határozzuk meg, hogy Ego milyen sorrendben találkozik velük. Az aktorokat és mozgásukat a következő azonosítókkal írhatjuk le:

  - **Aktor:** autó CAR, busz BUS, kerékpár BIKE, motorkerékpár M, golfkocsi GC, gyalogos PED, semmi XX, stb.
  - **Kezdeti/végső pozíciók:** észak N, kelet E, dél S, nyugat W, sávban áll St, kocsifelhajtó Dr, stb.
  - **Cselekvés:** 01, 02, 03, stb. Ezeket a cselekvés opciókat akkor használják, ha az aktor cselekvése nem szokásos vagy nem egyértelmű. Ezek jelentése a szcenárió kategóriák között különbözik. Például a 01 a CF (autó követése) szcenáriókban az elől haladó jármű hirtelen megállását jelöli, míg az I (kereszteződés) esetben azt jelenti, hogy az Ego-nak elsőbbsége van a stoptáblánál. A cselekvések azonosítója mindegyik szcenáriókategória oldalának tetején található.

  További azonosítók új szcenáriókkal együtt lesznek hozzáadva.

Paraméterezés Szimuláció segítségével lehetőségünk van megváltoztatni egy szcenárió főbb változóinak értékét. Ilyen módon gyorsan tudunk átfogó tesztkészletet előállítani. A legtöbb szcenárióban az Ego és a többi jármű sebessége és/vagy távolsága paraméterezhető.

Szcenárió létrehozása Nézzük meg egy összetettebb példán, hogyan készül és épül fel egy szcenárió az OAS eszköztára alapján. Tegyük fel, hogy egy három irányú, T alakú kereszteződésben szeretnénk vizsgálni, hogy az egyenesen haladó autonóm jármű megadja-e az elsőbbséget a jobbról érkező másik járműnek. A kész leírás a 2.4 ábrán látható.

Az OAS weboldaláról letölthető **Testing Toolkit** nevű készletben találhatóak a rajzi elemek, amiket felhasználhatunk. Mi egy olyan 3 irányú kereszteződést szeretnénk, ahol nincsenek gyalogátkelők. A Toolkit-ben csak gyalogátkelőhellyel rendelkező változat van, úgyhogy nekünk kell ez alapján létrehozni a gyalogátkelő nélküli változatot. Az eredeti és az új kereszteződés a 2.3 ábrán látható.

Ezután elhelyezzük az aktorokat a kezdeti állapotukban. Az Ego nyugatról érkezik, mozgását a tetején lévő zöld körrel jelölhetjük, a célját pedig egy zöld téglalappal jelezzük. A másik aktort a déli útszakaszra helyezzük. Mivel a kereszteződésben nincsenek táblák, a jobbkéz-szabály alapján az Ego-nak meg kell állnia, így ez az elvárt eredmény. Ez az elrendezés a 2.4 ábrán az **A** lépésben látható.

A következő, **B** lépésben Ego álló helyzetben van a kereszteződés előtt, ezt a tetején lévő piros körrel jelölhetjük. A másik jármű behajt a kereszteződésbe és balra kanyarodik. Itt a haladás irányát egy nyíllal tudjuk jelölni. A szcenárió ezen lépésében az Ego álló helyzetben kell maradjon, amíg a másik jármű az útjában van.

A **C** lépésben az aktor elhagyja a kereszteződést, így az Ego elindulhat és áthaladhat a kereszteződésen. Az utolsó lépésben pedig az Ego eléri célját, így a szcenárió véget ér.

Az ábrák és leírások elkészülte után hozzá kell rendelnünk egy egyedi azonosítót a korábban látott sablon alapján. Az azonosító első eleme az útszakaszok száma. Egy T kereszteződésbe három útszakasz vezet, így ennek értéke **3**. Az utak két sávosak, így a második

(a) Eredeti T kereszteződés gyalogátkelőkkel

(b) T kereszteződés gyalogátkelők nélkül

2.3. ábra. T kereszteződés

helyen lévő sávok száma a **2** értéket veszi fel. Az azonosító harmadik része a stoptáblák helyzete, a mi elrendezésünkben viszont nincsenek stoptáblák, így helyette **XX**-et írunk. A szcenárió a kereszteződés kategóriába tartozik, így az azonosító negyedik eleme a **I** értéket kapja.

Az azonosító következő része az Ego mozgását kell hogy leírja. A mi szcenáriónkban a jármű célja, hogy egyenesen továbbhaladjon, tehát ide **STR**-t kell írnunk. Mivel a vizsgált járművön kívül szerepet kap egy másik jármű is, ezt jelölnünk kell a **CAR** azonosítóval. Ezután ennek a járműnek a mozgását kell leírnunk. Az aktor dél felől érkezik a kereszteződésbe, ahonnan balra kanyarodva, vagyis nyugat felé távozik, így a mozgását **S>W** jelöli. Az azonosító utolsó eleme a cselekvés azonosító, mely kereszteződések esetében az alábbi négy értéket veheti fel:

- **01:** Ego-nak van elsőbbsége
- **02:** Aktornak van elsőbbsége
- **03:** Ego-nak elsőbbsége van, de aktor nem adja meg
- **04:** Ego-nak elsőbbsége van, de aktor elállja az utat

Ezek közül nekünk a **02**-re van szükségünk, hiszen a jobbkéz-szabály miatt az aktornak van elsőbbsége. Ha az azonosító fentebb látott részeit összerakjuk, a következőt kapjuk: **3-2-XX-I-STR-CAR:S>W:02**

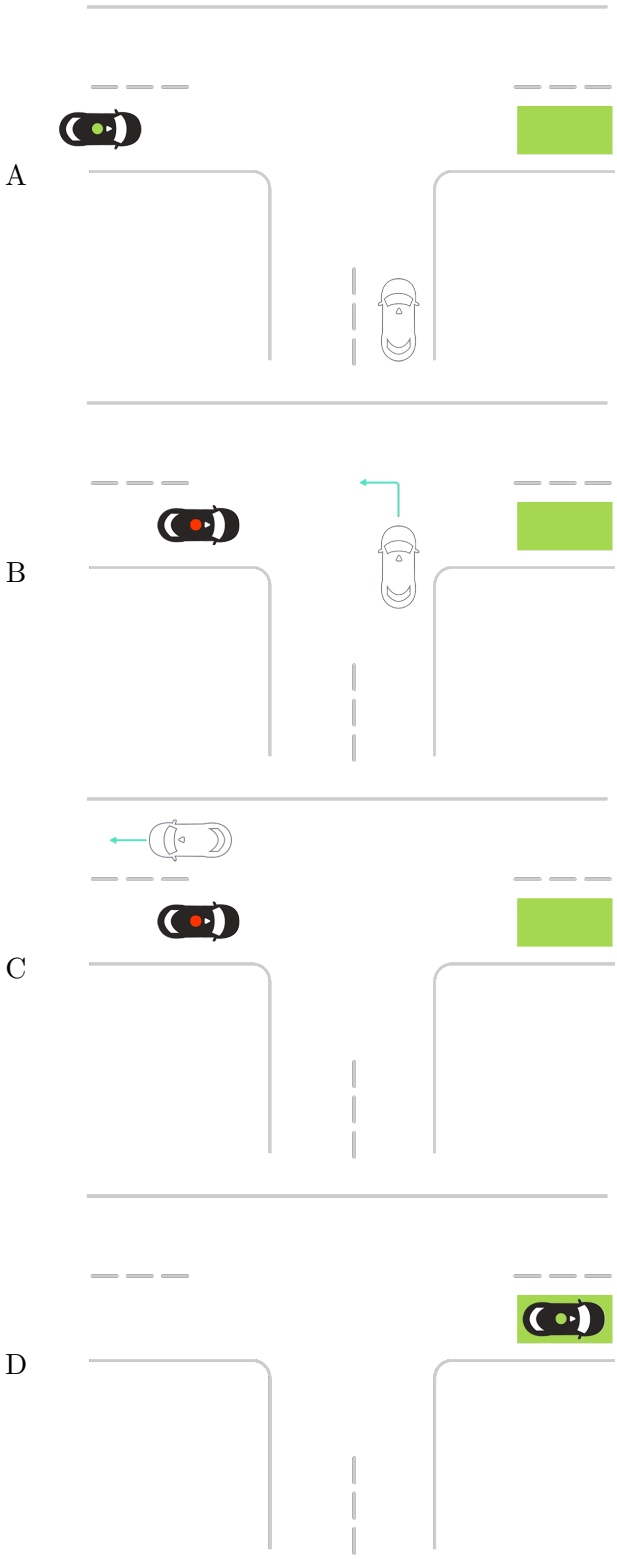
2.2.2. Funkcionális biztonság

Az OAS az ISO 26262 szabvány alapján adja meg a biztonsági megközelítést. Ahhoz, hogy sofőr nélkül is azonosítani tudjuk a meghibásodásokat és mérsékelni tudjuk őket, az autonóm járművekre vonatkozóan robusztus követelményeket kell megfogalmazni és a kockázatelemzéshez valamint a készségek validálásához átfogó, pontosan meghatározott folyamatok szükségesek.

A funkcionális biztonság rész egy-egy folyamatot vázol fel a biztonsági követelmények és a funkcionális követelmények felírására és vizsgálatára.

2.2.3. Autonómia értékelés

Az autonómia fejlődésének mérésére az OAS a Mérföld Per Beavatkozás (Miles Per Intervention, MPI) mértékegységet javasolja. Az MPI nem csak a már használt Mérföld Per



2.4. ábra. 3-2-XX-I-STR-CAR:S>W:02 azonosítójú szcenárió az OAS eszköztára alapján

Kikapcsolás (Miles Per Disengagement, MPD) metrikát rögzíti, hanem azt is magában foglalja, ha egy embernek kell biztonságkritikus döntést hoznia.

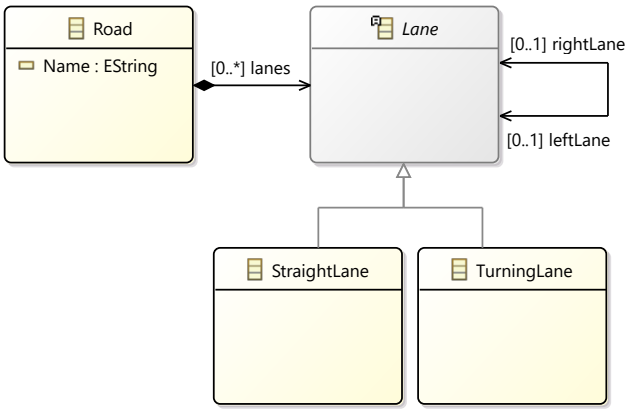
Beavatkozások csoportosítása A beavatkozásokat az alábbi kategóriákba sorolják, a leírás után az adott típusú beavatkozás segítségével mért metrika rövidítése látható (mérföld per beavatkozás):

- **Kritikus:** Kikapcsolás, ahol szükséges a sofőr beavatkozása, hogy megelőzzünk egy veszélyes helyzetet. MPCI
- **Rendszer hiba:** Kikapcsolás, ahol a jármű hardver- vagy szoftverhibát észlel és ennek megfelelően reagál. MPFI
- **Meghibásodás:** Kikapcsolás, ahol a jármű nem a tervezett módon viselkedik. MPMI
- **Nem támogatott:** Kikapcsolás, ahol a jármű olyan szituációba kerül, amit még nem támogat. MPUI
- **Élmény:** Kikapcsolás kellemetlen utazási élmény miatt. MPEI
- **Távoli:** Egy távoli operátor általi beavatkozás. MPRI

Kategóriánként külön mérik az átlagos megtett mérföldeket beavatkozások között.

2.3. Szakterület-specifikus nyelvek

A szakterület-specifikus nyelvek célja, hogy alkalmazási területekhez modellezési nyelvet, valamint hatékony eszköztámogatást nyújtsanak. Ilyen szakterület-specifikus modellezési nyelveket, valamint adatmodelleket például az Eclipse Modeling Framework (EMF) [7] segítségével lehet létrehozni. Ezek alapján tudunk akár Java osztályokat is generálni, vagy az elkészült Ecore metamodell közvetlenül is felhasználható egyéb alkalmazásokban, például gráfok generálásához, vagy gráfokban történő kereséshez. Az így létrehozott modellt metamodellnek nevezzük, ez összefoglalja a modellezési nyelv legfőbb koncepcióit és kapcsolatait, valamint a modellek alapvető felépítését.



2.5. ábra. Egy út metamodellje

A metamodell fogalmát és létrehozását [6], valamint az EMF általam használt részeit egy esettanulmányon keresztül mutatom meg, mely egy út leegyszerűsített metamodelljét írja le, ez látható a 2.5 ábrán.

Alapvetően az alábbi elemekből, és ezek speciális változataiból építhetjük fel a számunkra szükséges adatmodelleket:

- **EClass:** egy osztály nulla vagy több attribútummal és nulla vagy több referenciával.
- **EAttribute:** egy attribútum, aminek van egy neve és egy típusa.
- **EReference:** két osztály közötti asszociáció.
- **EDataType:** egy attribútum típusa, pl.: `int`, `float` vagy `java.util.Date`.

Az osztályokat beállíthatjuk absztrakt osztályként, ahogyan a példában a Lane osztállyal tettem. Ilyenkor belőlük nem, csak a leszármazottaikból tudunk példányt létrehozni. A Lane esetében kettő specializációt hoztam létre, ezek a StraightLane és a TurningLane, az ábrán a Lane irányába mutató nyíl jelzi ezt a kapcsolatot.

Az EReference több típusú relációt jelölhet az osztályok között, de mindegyikre igaz, hogy van egy neve valamint multiplicitása, ami lehet egy konkrét érték, vagy intervallum is. Utóbbi esetben a felső határ egész szám helyett végtelen értéket is felvehet, amit `-1` vagy $*$ jelöl.

Az ERefernce leggyakrabban egyszerű referenciát jelöl. Ennek egy speciális esete a kétirányú vagy inverz referencia, ami egy relációhoz két EReference példányt hoz létre. Ezek egymás inverzei, vagyis ugyanazon relációt jelölik ellentétes irányból. Ez a diagramon egy vonalként jelenik meg, melynek mindkét végén található egy-egy nyíl, erre példa a Lane osztály rightLane és leftLane referenciái.

Ezen kívül az EReference jelölhet tartalmazást is, ami a Road és a Lane osztályok között jelenik meg. A modellekben általában van egy osztály, ami közvetve tartalmazza az összes többit.

2.4. Gráfmintaillesztés

Az előbb látott metamodell alapján létrehozhatjuk utak modelljét. Előfordulhat, hogy ezek között a modellek között szeretnénk keresni olyanokat, amik bizonyos feltételeknek megfelelnek. Ilyen keresésre például akkor van szükségünk, ha szeretnénk kiszűrni olyan modelleket, amikben egyes részek nem felelnek meg az elvárásainknak. Ilyenkor arra van szükségünk, hogy bizonyos mintákat meg tudjunk találni a modellek belsejében. A modelleket értelmezhetjük gráfokként is, ahol az elemek a gráf csúcsai, a köztük lévő relációk pedig a gráf különböző típusú élei. Ezekben a gráfokban szeretnénk megtalálni bizonyos kapcsolatokat, mintákat. Gráfmintaillesztésnek hívjuk azt, amikor a paraméterként megadott mintákhoz hozzárendelünk olyan objektumokat, gráfokat, amikben megtalálható a paraméterként kapott minta.

Ehhez szükségünk van egy nyelvre, amivel le tudjuk írni ezen mintákat, és amit a gráfmintaillesztő értelmezni tud. Több ilyen is létezik, például az OCL (Object Constraint Language) [9] és a VQL (VIATRA Query Language) [8]. Ezekben a minták megfogalmazásához felhasználhatók a metamodellben definiált típusok, referenciák, stb. Az említett nyelvek közül a VQL-t választottam, mivel az általam használt gráfgenerátor ezt képes értelmezni.

VQL esetében a minta az következő módon adható meg. A mintát a `pattern` kulcsszóval kezdjük, ezután jön a minta neve, majd zárójelben a paraméterek. Végül kapcsos zárójelben kell megadni a kényszereket, melyek teljesülése esetén a minta illeszkedni fog a gráfra. A paraméterek esetében ajánlott megadni a típust, de ezt később, kényszerként is megtehetjük:

```
pattern myPattern(a,b : MyClass1, c : MyClass2) {...constraints...}
```

A legegyszerűbb kényszer a típuskényszer. Ezzel egy változó típusát határozhatjuk meg: `StraightLane(entityVariable);` Ez akkor illeszkedik, ha az entityVariable típusa StraightLane. Típuskényszert megadhatunk a Lane-re is, ekkor az összes leszármazottja kielégíti a kényszert.

A relációkra (referencia, kompozíció, attribútum) is adhatunk meg kényszereket. Azt, hogy *a* sáv része *b* útnak, a következő módon írhatjuk fel: `Road.lanes(b, a);`. Vegyük példának az előbb látott út metamodellt. Tegyük fel, hogy a sávok jobb oldali szomszédját szeretnénk megkeresni. Ezt úgy tehetjük meg, hogy Lane-Lane párokat keresünk, amik között rightLane referencia található. Az ehhez tartozó VQL minta az alábbi módon néz ki:

```
pattern rightLane(left : Lane, right : Lane) {
    Lane.rightLane(left, right);
}
```

Ha csak a metamodell alapján készítünk modelleket, egy sáv leftLane vagy rightLane referenciája saját magára is mutathat. Ilyen nyilván nem fordulhat elő a valóságban, hiszen egy sáv nem helyezkedhet el saját maga mellett, ezért szeretnénk megkeresni azokat a modelleket, melyekben ez előfordul. Ezt az alábbi mintával tehetjük meg:

```
pattern laneNextToItself(lane : Lane) {
    Lane.rightLane(lane, lane);
}
```

A gráfban kereshetünk tranzitív lezártat is. A mi esetünkben objektumok felett ennek jelentése, hogy két objektum között található út előre meghatározott típusú éleken keresztül. VQL esetében ezek az élek az általunk definiált minták lehetnek, tehát akár a gráfban lévő különböző referenciák sorozatát is tekinthetjük egy élnek. Amennyiben két elem között ilyen éleken keresztül létezik út, a minta illeszkedni fog rá.

Például megtalálhatjuk az összes sávot, ami egy adott sávtól jobbra található, ha megkeressük a rightLane éleken keresztül elérhető Lane-eket. Ehhez felhasználhatjuk az előbb felírt rightLane mintát. Saját mintákat a `find` kulcsszóval tudunk kényszerként felhasználni, tranzitív lezártat pedig a minta neve után írt + szimbólummal kereshetünk:

```
pattern rightLanes(left : Lane, right : Lane) {
    find rightLane+(left, right));
}
```

Amennyiben olyan Lane-párokat szeretnénk keresni, melyek nincsenek közvetlenül egymás mellett, a `neg find` kulcsszavak kell használnunk. Ezzel negatívan illeszthetünk mintát, vagyis nem ad illeszkedést, ha a hivatkozott mintának van illeszkedése, és ad illeszkedést, ha a gráfban nem található meg a hivatkozott minta.

```
pattern notRightLane(lane1 : Lane, lane2 : Lane) {
    neg find rightLane(lane1, lane2);
}
```

A fenti példa azonban nem biztosítja, hogy lane1 és lane2 különböző sávokat jelöl, ezt külön kényszerként kell felírnunk:

```
pattern notRightLane(lane1 : Lane, lane2 : Lane) {
    lane1 != lane2;
    neg find rightLane(lane1, lane2);
}
```

Amennyiben csak azt szeretnénk tudni, mely sávoktól jobbra található másik sáv, de nincs szükségünk arra az információra, hogy melyik sávok vannak tőle jobbra, a kényszerben a változót _ szimbólummal prefixálhatjuk. Ha az adott változót csak egyszer használjuk, nem szükséges elneveznünk, elég az aláhúzást beírni a változó helyére:

```
pattern hasRightLane(lane1 : Lane) {
    neg find rightLane(lane1, _);}
```

A VQL nyelv valójában ennél több dolgot képes kifejezni, itt viszont csak az általunk használt részeit mutattuk be. Ezeket a nyelvi eszközöket kombinálva felírhatók a számunkra szükséges kényszerek.

2.5. Kapcsolódó munkák

A kapcsolódó munkákat a [16] összefoglalása alapján rendszerezem.

Komponensszintű garancia autonóm rendszerekre Mivel gépi tanuláson alapuló módszerek, például mély neurális hálók gyakran irányítanak autonóm vezérlő komponenseket, a közelmúltban a kutatások elkezdtek a tesztelés és formális verifikáció támogatására koncentrálni, hogy feltárják a mély neurális hálókban rejlő érzékenységi problémákat [15]. Ezek a kutatások többnyire komponensszintű garanciára korlátozódnak, egyetlen autonóm komponens viselkedését vizsgálják (például sávtartó asszisztens), míg a rendszerszintű biztonság kihívásai számos autonóm komponenssel nyitottak maradnak. Ily módon ugyan adaptálunk néhány magas szintű ötletet, ez az adaptáció kihívásokat rejt a garancia szintjeinek különbsége miatt (lásd még [21, 19]).

Rendszerszintű megközelítések Önvezető autók szokásos viselkedésének formális verifikációját javasolták [22]-ben. A közlekedési szituációkat és a járművek lehetséges reakcióit egy absztrakt szinten verifikálják. Széles körben használnak szimulációt, hogy autonóm rendszerek rendszerszintű viselkedését vizsgálják [22]. Azonban a változatos tesztelrendezések elkészítése még nincs megoldva.

Rendszerszintű elrendezés generálás Számos tesztelési megközelítés [1, 11, 17, 18, 27, 26] keresésalapú módszert használ, hogy (1) tesztelrendezéseket állítson elő autonóm ágensek számára, (2) szimulálja az ágenseket egy ellenséges környezetben, (3) ellenőrizze, hogy megfelelnek-e az előírt biztonsági tulajdonságoknak és (4) kihívást jelentő tesztelrendezéseket állítsanak elő úgy, hogy a viselkedés devianciájának mértékét függvényként használják (pl.: egy tesztelrendezés jobb, ha az autonóm ágens gyengén teljesít). A világot leíró gráfoknak komplex időbeli, térbeli és kauzális információt kell reprezentálniuk, ami jelenleg nem támogatott, amikor autonóm rendszerek rendszerszintű garanciájához generálunk szcenáriókat. A dolgozatunkban bemutatott eredmény is ezek közé a megközelítések közé tartozik. A legnagyobb különbség az általunk bemutatott módszer és a korábbi módszerek között, hogy mi garantálni kívánjuk a tesztesetek diverzitását.

3. fejezet

Áttekintés

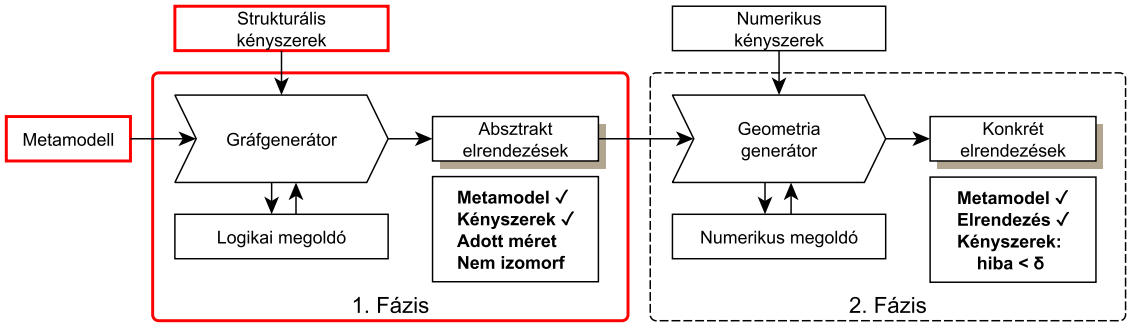
Ahogy korábban említettem, számos tesztpálya-generálási megközelítést javasol a szakirodalom, különböző célokkal és garanciákkal. A dolgozatomban bemutatott módszer leginkább az Open Autonomous Safety [23] szabványban bemutatott elrendezésekhez illeszkedik, melynek célja az önvezető komponensek funkcionális tesztelése. Szisztematikus tesztelésnél a célunk egy módszer kidolgozása, mellyel (egy ekvivalencia-particionáló módszerrel) az összes különböző tesztelrendezést elő tudjuk állítani, valamint a tesztesetek fedését is tudjuk vizsgálni.

3.1. Tesztelrendezés ekvivalencia particionálása

Kritikus rendszerek tesztelésénél az egyik fontos metrika a tesztlefedettség. Autonóm járművek szisztematikus tesztelését jelenleg tipikusan egy előre definiált, manuálisan összeállított tesztkészlet alapján végzik (mint például [23]). Ilyen tesztkészleteknél azonban nehezen határozható meg, hogy az összes lehetséges szituációnak, amelybe a jármű kerülhet, hány százalékát fedik le a tesztesetek.

Az általunk kidolgozott automatikus módszerrel elkészített tesztkészleteknél garanciát vállalunk arra, hogy a tesztesetek jelentősen eltérnek egymástól. A közönséges szoftverek tesztelésénél gyakran alkalmazott módszer az ekvivalencia-particionálás. A teljes teret ekvivalencia partíciókra bontjuk, és egy-egy reprezentáns tesztesetet képzünk ekvivalencia partíciónként. Az ekvivalencia partíciók reprezentáns elemének tesztelése jó módszer véges tesztkészlet tesztlefedettségének maximalizálásához.

3.2. Funkcionális áttekintés



3.1. ábra. Funkcionális áttekintés

A tesztelrendezések generálása két fázisban történik: a 3.1 ábrán látható módon az első fázisban absztrakt elrendezéseket készítünk, a második fázisban pedig ezekhez rendelünk koordinátákat, így elkészítve a konkrét elrendezéseket. Én az első fázist valósítottam meg, ennek megfelelően a dolgozatban ezen részt mutatom be részletesen.

Az **1. fázisban** az absztrakt elrendezések elkészítéséhez egy gráfgenerátort használunk, amely egy metamodell alapján készíti el lehetséges modellek strukturális részét. Egy logikai megoldó segítségével megvizsgálja, hogy a strukturális kényszereknek megfelelnek-e a modellek. Amennyiben hibás modellt talál, eldobja azt, és csak a kényszereknek megfelelőket ad eredményül. A gráfgenerátor paraméterként megkapja a kívánt modellek mennyiségét és méretét, valamint külön megadhatjuk, hogy a különböző típusú modellelemekből hány darab legyen az elkészült modellekben, így meghatározhatjuk azok bonyolultságát.

A logikai megoldónak és a strukturális kényszereknek köszönhetően biztosítani tudjuk azt is, hogy az elkészült elrendezések nem izomorfak, azaz különböző gráfmodellekkel írhatjuk le őket, és lényegi különbség van közöttük.

A **2. fázisban** konkrét elrendezéseket generálunk. A cél, hogy az absztrakt elrendezésekhez konkrét koordinátákat rendeljünk. Az absztrakt modell elemeihez és relációihoz numerikus kényszereket feleltetünk meg, amiket egy szabványos formátumba tudunk exportálni (SMT-LIBv2). A numerikus kényszereket tartalmazó leírás egy cserélhető numerikus megoldó bemenete, ami eredményképpen konkrét számokat rendel az egyes koordinátákhoz, melyek pontosan, vagy δ pontossággal kielégítik a kényszereket (azaz a megoldás egy helyes megoldástól legfeljebb δ különbségre van). Ez bővebben részletezve a dolgozat alapjául szolgáló TDK dolgozatban [5] található.

3.3. Tesztelési módszerek

A dolgozatban bemutatott kétlépéses generálási módszer két különböző stratégiában képes teszteseteket előállítani:

- A teszteseteket szisztematikus generálásánál mindegyik partícióhoz egy reprezentáns tesztesetet generálunk (lásd: 3.1), ezzel a tesztkészlet **teljességét** és **lefedettségét** maximalizáljuk.
- Lehetőséget adunk arra, hogy egy ekvivalencia osztályból nem csak a reprezentáns, hanem alternatív teszteseteket is generáljunk. Ezzel az autonóm járművek tesztelésének a **robusztusságát** terjesztjük ki.

Előfordulhat, hogy olyan teszteset generálódik, ami a valóságban nem megépíthető, nem tesztelhető. Ebben az esetben **finomíthatjuk** úgy a követelményeket, hogy bevezethetünk új strukturális vagy numerikus kényszereket, amikkel javítjuk a tesztesetek generálását.

3.4. Vizsgálandó viselkedésbeli készségek

Az iparágban már számos alkalommal definiálták az autonóm járművek biztonságos működéséhez szükséges standardizált viselkedésbeli készségeket különböző állami intézmények és az iparág vezető szereplői, így tett például az USA Közlekedési Minisztériuma [4] és a Waymo [24]. Ezek egy részhalmaza alkotja az Open Autonomous Safety és az én munkám alapját is. Nézzük meg milyen készségekről van szó és ezek hogyan kapcsolódnak a munkámhoz.

National Highway Traffic Safety Administration (NHTSA)

- **Forgalom észlelése és reagálás (forgalomba történő besorolás):** különböző aktorok elhelyezésével tesztelhető, hogy a jármű képes-e más járművek között ütközés nélkül közlekedni.
- **Az ellenkező menetirányú sávból a mi sávunkba átsodródó jármű észlelése és reagálás:** cél, hogy az aktorok olyan mozgását vagy pozícióját is le tudjuk írni, ami nem szabályos, de a valóságban előfordul. A modell alapján szisztematikusan fel lehet sorolni az aktorok viszonyait.
- **Különböző közlekedési táblák észlelése és reagálás:** A javasolt megoldással nem tudjuk vizsgálni, hogy a táblákat minden körülmények között, például részleges takarásban felismeri-e a gépi látás. Azt tudjuk és szeretnénk ellenőrizni hogy a felismert táblát helyesen értelmezi-e és annak megfelelően cselekszik-e a jármű.
- **A jármű útjában lévő statikus akadályok észlelése és reagálás:** A javasolt megoldással képesek vagyunk először az utak egy elrendezését előállítani, majd ezen elhelyezni aktorokat és akár különböző akadályokat is.
- **Kereszteződések navigálása és kanyarodás:** ez a megoldásom egyik erőssége, ugyanis sokféle, akár nagyon bonyolult kereszteződést elő lehet vele állítani, így vizsgálhatjuk, hogy a jármű eljut-e a kívánt helyre és közben a megfelelő sávokat használja-e.

Az NHTSA által összegyűjtött készségek jelentős része a jármű közlekedéssel kapcsolatos érzékelésére és döntésére vonatkozik. A döntések helyességvizsgálatát tudjuk segíteni a tesztelrendezések létrehozásával.

Waymo

- **Az úton lévő gyalogosok, állatok észlelése és reagálás:** a táblákhoz hasonlóan itt sem a gépi látást kívánjuk vizsgálni, hogy mindig felismeri-e a gyalogosokat és állatokat. A cél annak ellenőrzése, hogy az észlelt aktorokat kikerüli-e a jármű.
- **Jármű feletti kontroll elvesztésének észlelése (csúszós út):** az ilyen jellegű viselkedés megvalósítása alapvetően nem a döntési komponens feladata.
- **Jármű-, rendszer- és komponensszintű hiba észlelése és reagálás:** az ilyen jellegű hibákat nem kívánjuk vizsgálni, ez nem a döntési komponens hatókörébe tartozik.
- **Ütközést nem eredményező veszélyes helyzetek észlelése és reagálás** (pl.: nyitott ajtó, kicsatolt biztonsági öv): ilyen jellegű dolgokkal szintén nem a vizsgálandó döntési komponens foglalkozik.

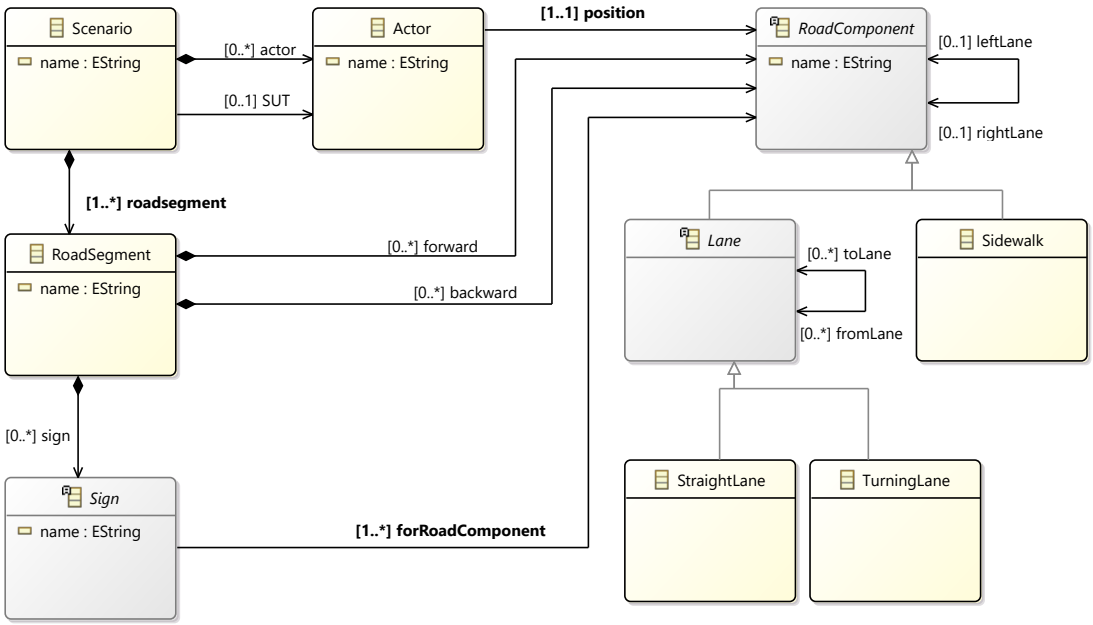
A Waymo természetesen szintén fontosnak tartja, hogy a jármű helyesen döntsön különböző forgalmi szituációkban, viszont több olyan készséget is felsorol, melyek nem tartoznak a döntési komponens hatókörébe.

4. fejezet

Absztrakt elrendezések generálása

4.1. Tesztelrendezés metamodellje

A dolgozat céljának eléréséhez szükséges az utak lehetséges elrendezéseinek egy absztrakt leírására, ehhez hoztam létre egy metamodellt, amely összefoglalja az elkészíthető modellek felépítését, valamint az elemeik között lehetséges kapcsolatokat. Ennek elkészítése során alapvetően az volt a cél, hogy az összes lehetséges elrendezést le tudjuk írni valamilyen módon. Ennek első lépése, hogy leírjuk az összes lehetséges logikai elrendezést, erre használjuk a metamodellezést. Annak érdekében, hogy valóban le lehessen írni az összes lehetséges elrendezést a metamodell alapján, az Open Autonomous Safety szabványt [23] vettem alapul. Ellenőriztem, hogy az abban összegyűjtött tesztesetek az én megoldásommal is leírhatók-e, és általánosítottam őket, hogy méret- és komplexitásbeli korlátok nélkül tudjuk őket modellezni. Az elkészült metamodell a 4.1 ábrán látható.



4.1. ábra. A tesztelrendezések felépítése

4.1.1. Scenario - A modell gyökéreleme

A modell gyökérelemét Scenario-nak nevezzük, ez tartalmazza az összes többi elemet. Modellenként egy darab gyökérelemet hozunk létre, így minden modell egy darab tesztelrendezést tartalmaz.

4.1.2. Actor – Tesztelrendezés szereplői

A szcenáriók szereplőit aktoroknak nevezzük. Ezek lehetnek autók, kerékpárok, egyéb járművek, vagy gyalogosok is. Pozíciójukat az alapján határozzuk meg, hogy melyik RoadComponent elemen helyezkednek el.

4.1.3. RoadSegment – Úthálózat felépítése

A tesztelrendezések legfőbb aspektusa az út, melyen a járművek közlekednek. Az utakat legegyszerűbben úgy kezelhetjük, ha logikus helyeken feldaraboljuk őket. Ezeket a részeket jelöli a modell RoadSegment eleme. Ilyen felosztás lehet például egy kereszteződés esetében az, ha a kereszteződés belsejét egy RoadSegment-be, valamint az az oda bemenő és onnan kivezető utakat is egy-egy RoadSegment-be rendezzük.

4.1.4. RoadComponent – Útszakasz felépítése

Mivel a járművek helyzetét részletesebben szeretnénk leírni annál, mint hogy melyik úton van, a RoadSegment-et tovább kell bontanunk. A RoadComponent elemek leginkább sávszakaszokat jelölnek, de lehetnek járdaszakaszok is. Mivel ez egy absztrakt osztály, közvetlenül nem példányosítható, csak a (nem absztrakt) leszármazottai. A RoadComponent-re vonatkozó leftLane és rightLane referenciákkal az egy RoadComponent-en belüli, azonos menetirányban egymás mellett lévő RoadComponent-ek sorrendjét adhatjuk meg.

Az útszakaszon belüli sávokat meg kell tudnunk különböztetni menetirány szerint, és meg kell tudnunk adni, viszont a balra-jobbra szomszéd viszont szeretnénk menetirány szerint értelmezni. Emiatt a RoadSegment-ek két külön listában tárolják a RoadComponent-eket menetirány szerint, forward és backward néven. Ez a fajta megkülönböztetés globálisan nem értelmezhető, csak egy-egy útszakaszon belül. Kereszteződések esetében még így sem egyértelmű, viszont ott nincs is rá szükség, a kapcsolódó StraightLane-ek irányából kitalálható a sávok iránya és pozíciója, éppen ezért egy kereszteződés belsejében az összes sávszakasz forward irányú. Ez azonban metamodell szinten nem írható le, mivel a kereszteződés ugyanolyan útszakasz, mint az egyenes út. Erre megoldást a 4.2 részben részletezett kényszerek létrehozása jelent.

4.1.5. Lane – Sávok és viszonyuk

Sok esetben eltérő módon viselkednek az egyenes (StraightLane) sávok és a kanyarodó (TurningLane) sávok, a modellek létrehozása során különböző szabályok vonatkoznak rájuk. (Itt a kanyarodó sáv elsősorban a kereszteződések azon sávjait, lehetséges haladási irányait jelenti, melyeket követve a KRESZ alapján irányt változtatunk.

Mivel mindkét típusú sáv a Lane osztályból származik, lehet toLane referenciájuk. Ez azt mondja meg, hogy a KRESZ szabályait betartva adott sávból melyik másik sávokba haladhat tovább egy autó. Ennek inverze a fromLane, mely azt mondja meg, hogy adott sávba melyik sávokból lehet közvetlenül eljutni. Ezekkel a kapcsolatokkal csak a ténylegesen egymást követő sávokat kötjük össze, az egymás mellett lévő sávok közötti sávváltási lehetőséget nem jelöljük.

Abban az esetben, amikor két ellenkező irányú sáv egymás mellett halad, mindkettőtől a saját menetiránya szerint balra van a másik. Ezt azonban nem jelölhetjük leftLane

referenciával, hiszen az inverz referencia miatt a rightLane referencia is beállítódna. Ha szükségünk van arra az információra, mely sávok vannak ilyen viszonyban egymással, megtalálhatjuk őket úgy, hogy mindkét irányból megkeressük a legbalrább lévő sávot.

4.1.6. Sign – Közlekedési szabályok

Az egyes RoadComponent-ekre vonatkozó közlekedési táblákat és szabályokat a Sign elemek segítségével írhatjuk le. Az egyes táblákat a Sign osztályból leszármaztatva hozhatjuk létre. A közlekedési táblákat az útszakaszok tartalmazzák, és a táblák forRoadComponent referenciája mutatja, mely RoadComponent-ekre vonatkoznak. Ha egy tábla forRoadComponent referenciája egy "A" RoadComponent-re mutat, az azt jelenti, a tábla akkor lép érvényre, amikor az autó "A"-ra lép.

4.2. Modellekre vonatkozó kényszerek

Mivel az elsődleges cél a lehető legteljesebb tesztkészlet előállítása mind a modellek komplexitását, mind méretüket tekintve, a metamodellben kevés korlátot vezettem be. Így viszont lehetőség adódik olyan elrendezések létrehozására, melyek az úthálózat logikájából, vagy akár a fizika törvényeiből következően a valóságban nem létezhetnek.

Mint ahogy a 4.1 fejezetben szóba került, a tesztelrendezéseknek meg kell felelniük olyan szabályoknak is, melyek a metamodellben nem kényszeríthetőek ki. Ezeket a szabályokat kényszereknek hívjuk, és több módon is implementálhatóak. Ezen dolgozat elkészítése során a Viatra Query Languege mellett döntöttem, mivel a használt gráfgenerátor ezt használja a modellek létrehozása során.

Meg kell határozni és le kell írni, hogy a különböző sávszakaszok mikor és milyen módon kapcsolódhatnak egymáshoz. Ha csak a metamodellt nézzük, azt láthatjuk, hogy bármely két sáv közé húzhatunk például leftLane-rightLane éleket. A valóságban viszont csak olyan sávok lehetnek egymás mellett, melyek ugyanazon útszakaszon helyezkednek el. A modellgenerátornak meg tudjuk mondani, hogy csak olyan tesztelrendezéseket szeretnénk kapni, melyekben nincsenek külön útszakaszon lévő sávok egymás mellett, sőt, azonos útszakaszon ellentétes menetirányában sem lehetnek egymás mellett. Az ilyen kényszereket a Viatra Query Language segítségével írhatjuk le, ebben az esetben a hibás modelleket kell megtalálnunk.

A fenti példához tartozó VQL minták:

```
1  @Constraint
2  pattern laneInWrongSegmentOrDirection(lane1: RoadComponent, lane2: RoadComponent) {
3      RoadComponent.leftLane(lane1, lane2);
4      neg findlanesInSameSegmentAndDirection(_segm, lane1, lane2);
5  }
6
7  private pattern lanesInSameSegmentAndDirection
8      (seg: RoadSegment, l1: RoadComponent, l2: RoadComponent)
9  {
10     RoadSegment.forward(seg, l1);
11     RoadSegment.forward(seg, l2);
12 } or {
13     RoadSegment.backward(seg, l1);
14     RoadSegment.backward(seg, l2);
15 }
```

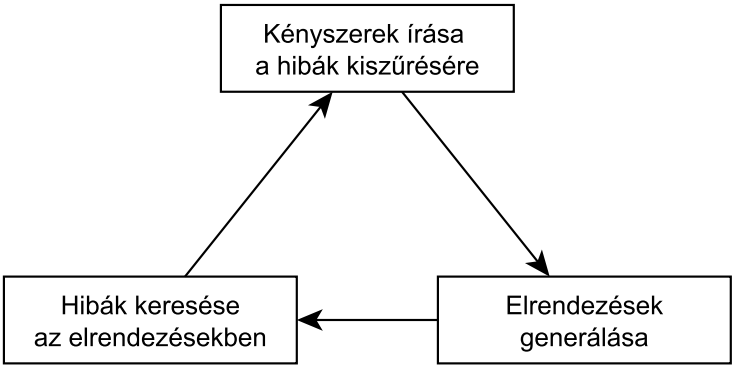
Néhány ilyen szabály triviális, másokat viszont nem egyszerű megtalálni. A kényszereket egy iteratív módszerrel lehet finomítani, ez a 4.2 ábrán látható.

1. A modellekre jólformáltsági kényszereket írunk fel, vagy a meglévőket módosítjuk oly módon, hogy az ismert hibákat kiszűrjék. Ezt a fent bemutatott VQL nyelven

tesszük meg, felírunk olyan mintákat, amiket nem szeretnénk, hogy szerepeljenek a modellben.

2. A finomított kényszereket paraméterként átadva a generátornak elkészítünk néhány modellt. Ahogy a kényszerek egyre több egyszerű hibát kiszűrnek, a modellek méretét és komplexitását növeljük, hogy összetettebb hibák is előfordulhassanak bennük.

3. Megvizsgáljuk a generált modelleket, és hibákat keresünk bennük, amiket az eddigi kényszerek nem szűrtek ki. Ennek egyik, általunk is használt módja, hogy manuálisan megpróbáljuk lerajzolni az elrendezést. Ha ez nem sikerül, megfogalmazzuk a hiba okát, majd amennyire lehet, általánosítjuk azt.



4.2. ábra. A kényszerek keresésének iteratív folyamata

Előfordulhat, hogy egy modellre több minta is illeszkedik, de ez nem probléma, ez gyorsítja a generálás folyamatát. Az elkészült kényszereket több kategóriába csoportosíthatjuk, mindegyik kategóriából a fontosabbakat lentebb részletezzük:

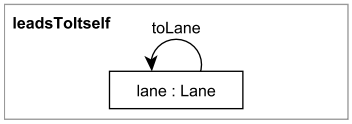
- Összes sávra vonatkozó általános kényszerek
- Egyenes sávokra vonatkozó kényszerek
- Kanyarodó sávokra vonatkozó kényszerek
- Útszakaszokra vonatkozó kényszerek

A kényszerek segítségével többek között azt szeretném elérni, hogy a generált modellben egy útszakaszon belül az egy irányba haladó sávok egymás mellett legyenek, vagyis az összes kapcsolódjon egymáshoz, és csak olyan sávok lehessenek egymás mellett, amik azonos útszakaszhoz tartoznak. Emellett fontos. hogy kiszűrjük azon modelleket melyekben az egymás mellett haladó sávok valamilyen módon keresztezik egymást. Kanyarodó sávok esetében a metamodell felépítéséből adódóan szükséges kényszerek felírása. Mivel ezen sávok iránya csak a hozzájuk csatlakozó sávok irányából tudható meg, fontos, hogy csak egyenes útszakaszhoz csatlakozzanak. Az útszakaszok esetében el kell kerülni, hogy üres, vagyis sávokat nem tartalmazó elemeket hozzon létre a generátor, valamint biztosítani kell, hogy az elrendezés egy összefüggő úthálózatot alkosson.

4.2.1. Sávokra vonatkozó általános kényszerek

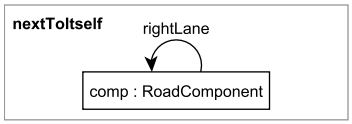
Az alábbi kényszerek minden sávra vonatkoznak, ha pedig Lane helyett RoadComponent típuskényszer szerepel a mintában, az a járdákra is érvényes.

Ellenpélda	Gráfminta



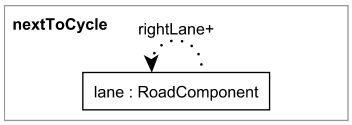

```
pattern leadsToItself(lane: Lane) {
  Lane.toLane(lane, lane);
}
```

Egy Lane toLane referenciája saját magára mutat.



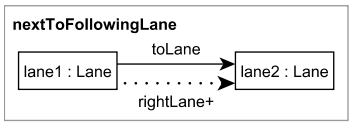

```
pattern nextToItself(comp: RoadComponent) {
  RoadComponent.rightLane(comp, comp);
}
```

Egy RoadComponent rightLane referenciája saját magára mutat.



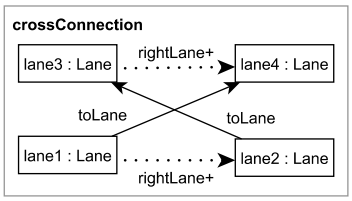

```
pattern nextToCycle(lane: RoadComponent) {
  find rightLane+(lane,lane);
}
```

A modellben rightLane referenciákból álló kör található.



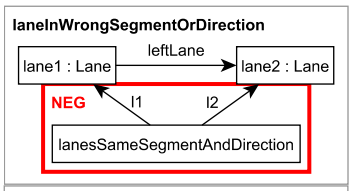

```
pattern nextToFollowingLane(lane1: Lane, lane2: Lane) {
  find rightLane+(lane1,lane2);
  Lane.toLane(lane1, lane2);
}
```

Egy Lane mellett és mögött ugyanaz a Lane található.

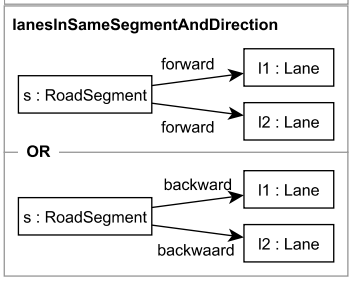



```
pattern crossConnection(lane1: Lane, lane2: Lane, lane3: Lane,
        lane4: Lane) {
  find rightLane+(lane1, lane2);
  find rightLane+(lane3, lane4);
  Lane.toLane(lane1, lane4);
  Lane.toLane(lane2, lane3);
}
```

Két párhuzamos Lane másik két párhuzamos Lane-hez csatlakozik, de közben keresztezik egymást.



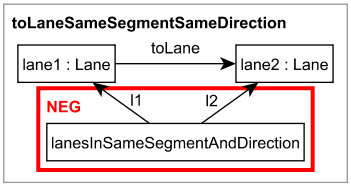

```
pattern laneInWrongSegmentOrDirection(lane1: RoadComponent,
        lane2: RoadComponent) {
  RoadComponent.leftLane(lane1, lane2);
  neg find lanesInSameSegmentAndDirection(_segm, lane1, lane2);
}
```




```
private pattern lanesInSameSegmentAndDirection(
        s: RoadSegment, l1: RoadComponent, l2: RoadComponent) {
  RoadSegment.forward(s, l1);
  RoadSegment.forward(s, l2);
} or {
  RoadSegment.backward(s, l1);
  RoadSegment.backward(s, l2);
}
```

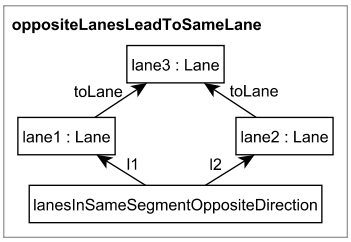
Egy Lane és a mellette lévő Lane vagy különböző RoadSegment-ben, vagy ugyanazon RoadSegment ellentétes menetirányában vannak.

Ellenpélda	Gráfminta



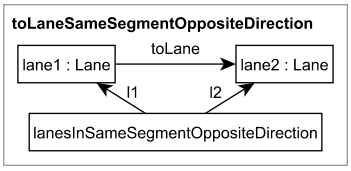

```
pattern toLaneSameSegmentSameDirection(lane1: Lane,
        lane2 : Lane) {
  Lane.toLane(lane1, lane2);
  find lanesInSameSegmentAndDirection(_, lane1, lane2);
}
```

Egy Lane és a hozzá toLane éllel kapcsolódó másik Lane ugyanazon RoadSegment azonos menetirányú sávjai.



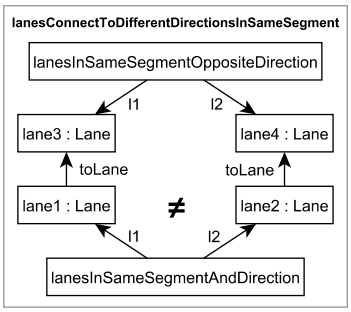

```
pattern oppositeLanesLeadToSameLane(lane1 : Lane, lane2 : Lane,
        lane3 : Lane) {
  find lanesInSameSegmentOppositeDirection(_, lane1, lane2);
  Lane.toLane(lane1, lane3);
  Lane.toLane(lane2, lane3);
}
```

Azonos RoadSegment-ben lévő, ellentétes irányú sávok ugyanabba a sávba vezetnek. Ugyanez felírandó toLane helyett fromLane-nel is.



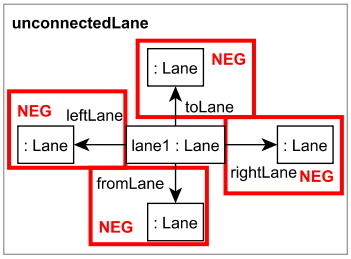

```
pattern toLaneSameSegmentOppositeDirection(lane1: Lane,
        lane2 : Lane) {
  Lane.toLane(lane1, lane2);
  find lanesInSameSegmentOppositeDirection(_, lane1, lane2);
}
```

Egy Lane és a hozzá toLane éllel kapcsolódó másik Lane ugyanazon RaodSegment ellentétes menetirányú sávjai.




```
pattern lanesConnectToDifferentDirectionsInSameSegment(
        l1: Lane, l2: Lane, l3: Lane, l4: Lane) {
  l1 != l2;
  find lanesInSameSegmentAndDirection(_, l1, l2);
  Lane.toLane(l1, l3);
  Lane.toLane(l2, l4);
  find lanesInSameSegmentOppositeDirection(_, l3, l4);
}
```

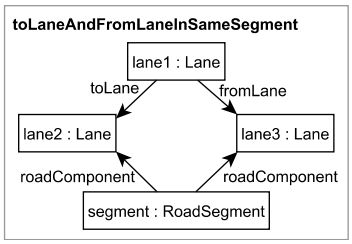
Párhuzamos, azonos menetirányú sávok eltérő RoadSegment-be vezetnek.




```
pattern unconnectedLane(lane1 : Lane) {
  neg find toLane(lane1, _);
  neg find fromLane(lane1, _);
  neg find leftLane(lane1, _);
  neg find rightLane(lane1, _);
}
```

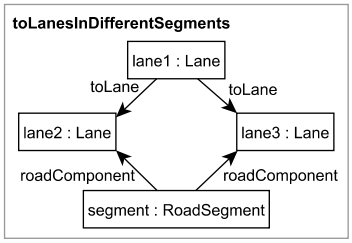
Egy Lane nem kapcsolódik semmihez.

Ellenpélda Gráfminta




```
pattern toLaneAndFromLaneInSameSegment(lane1: Lane) {
  Lane.toLane(lane1, lane2);
  Lane.fromLane(lane1, lane3);
  find roadComponentOfRoadSegment(segment, lane2);
  find roadComponentOfRoadSegment(segment, lane3);
}
```

Két Lane, melyek ugyanahhoz a Lane-hez csatlakoznak, egyik toLane, másik pedig fromLane referenciával, ugyanabban a RoadSegment-ben vannak.

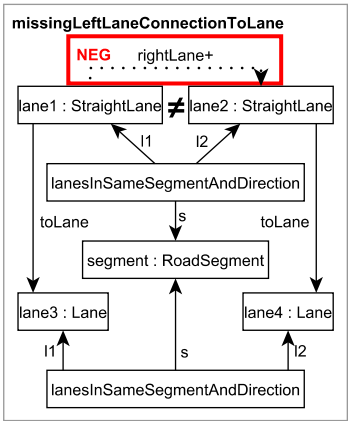



```
pattern toLanesInDifferentSegments(lane : Lane) {
  Lane.toLane(lane, lane2);
  Lane.toLane(lane, lane3);
  find roadComponentOfRoadSegment(segment, lane2);
  neg find roadComponentOfRoadSegment(segment, lane3);
}
```

Egy Lane több különböző RoadSegment-ben lévő Lane-hez is közvetlenül kapcsolódik toLane éllel. Ugyanez fromlane éllel is felírandó.

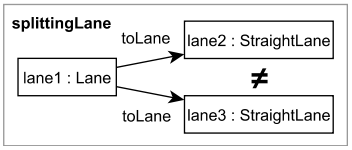
4.2.2. Egyenes sávokra vonatkozó kényszerek

Ellenpélda Gráfminta




```
pattern missingLeftLaneConnectionToLane(lane1 : StraightLane,
        lane2 : StraightLane) {
  lane1 != lane2;
  find lanesInSameSegmentAndDirection(segment1, lane1, lane2);
  Lane.toLane(lane1, lane3);
  Lane.toLane(lane2, lane4);
  find lanesInSameSegmentAndDirection(segment1, lane3, lane4);
  neg find leftLaneOrRightLaneTransitive(lane1, lane2);
}
```

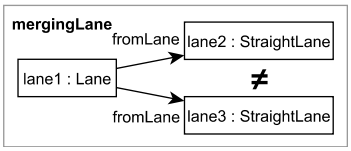
Azonos RoadSegment-ben lévő, megegyező menetirányú, egyenes és párhuzamos sávok között hiányzik a rightLane–leftLane kapcsolat.




```
pattern splittingLane(lane : Lane) {
  Lane.toLane(lane, lane2);
  Lane.toLane(lane, lane3);
  lane2 != lane3;
  StraightLane(lane2);
  StraightLane(lane3);
}
```

Egy Laneből két StraightLane-be is közvetlenül tovább lehet haladni.

Ellenpélda Gráfminta

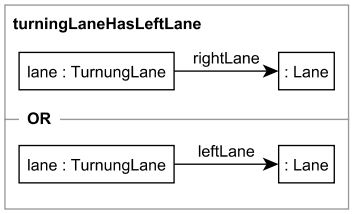



```
pattern mergingLane(lane : Lane) {
  Lane.fromLane(lane, lane2);
  Lane.fromLane(lane, lane3);
  lane2 != lane3;
  StraightLane(lane2);
  StraightLane(lane3);
}
```

Egy Lane-be két egye StraightLane-ből is közvetlenül el lehet jutni.

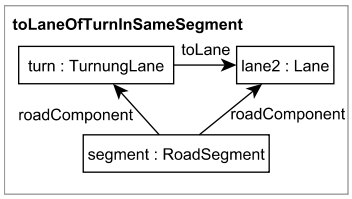
4.2.3. Kanyarodó sávokra vonatkozó kényszerek

Ellenpélda Gráfminta



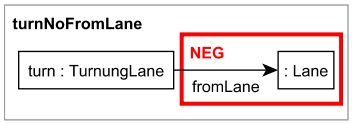

```
pattern turningLaneHasLeftLane(lane: TurningLane) {
  RoadComponent.rightLane(lane, _);
} or {
  RoadComponent.leftLane(lane, _);
}
```

TurningLane-nek van rightLane vagy leftLane referenciája.



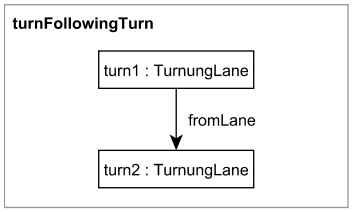

```
pattern toLaneOfTurnInSameSegment(lane1: TurningLane,
        lane2: Lane) {
  Lane.toLane(lane1, lane2);
  find roadComponentOfRoadSegment(segment, lane1);
  find roadComponentOfRoadSegment(segment, lane2);
}
```

Egy TurningLane toLane referenciája olyan Lane-re mutat, amely vele azonos RoadSegment-ben található. Ugyanez felírandó toLane helyett fromLane-nel is.




```
pattern turnNoFromLane(turn: TurningLane) {
  neg find fromLane(turn, _fromlane);
}
```

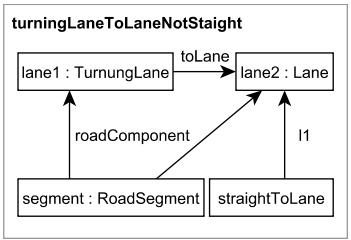
Egy TurningLane nem rendelkezik fromLane referenciával, így a modell alapján nem mondható meg, merre kanyarodik. Ugyanez felírandó fromLane helyett toLane-nel is.




```
pattern turnFollowingTurn(turn1: TurningLane,
        turn2: TurningLane) {
  Lane.toLane(turn1, turn2);
}
```

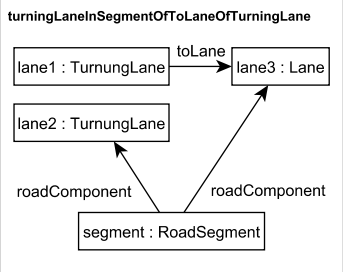
Egy TurningLane közvetlenül kapcsolódik egy másik TurningLane-hez, így a modell alapján nem mondható meg, merre kanyarodik.

Ellenpélda	Gráfminta



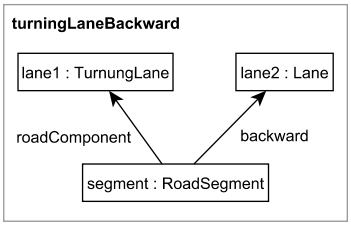

```
pattern turningLaneToLaneNotStaight(lane1 : TurningLane,
        lane2 : Lane) {
  find roadComponentOfRoadSegment(segment, lane1);
  find roadComponentOfRoadSegment(segment, lane2);
  neg find straightToLane(lane2, _);
}
```

Ha egy RoadSegment tartalmaz TurningLane-t, akkor a RoadSegment összes Lane eleme csak StraighLane elemhez kapcsolódhat toLane és fromLane éllel.




```
pattern turningLaneInSegmentOfToLaneOfTurningLane(
        lane1 : TurningLane, lane2 : TurningLane) {
  Lane.toLane(lane1, lane3);
  find roadComponentOfRoadSegment(segment, lane3);
  find roadComponentOfRoadSegment(segment, lane2);
}
```

Egy TurningLane-hez toLane éllel csatlakozó Lane-nel azonos RoadSegment-ben található egy TurningLane. Ugyanez felírandó toLane helyett fromLane éllel is.

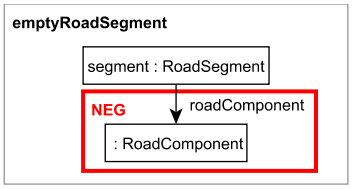



```
pattern turningLaneBackward(lane1: TurningLane, lane2 : Lane) {
  find roadComponentOfRoadSegment(segment, lane1);
  RoadSegment.backward(segment, lane2);
}
```

Egy Lane, amivel azonos RoadSegment-ben található TurningLane is, backward irányú. Kereszteződések esetében még egy RoadSegment-en belül sem különböztethető meg konzisztens módon két menetirány, ezért ilyen esetben a RoadSegment összes RoadComponent-je forward irányú. Pozíciójuk és orientációjuk a kapcsolódó RoadComponent-ek alapján kikövetkeztethető.

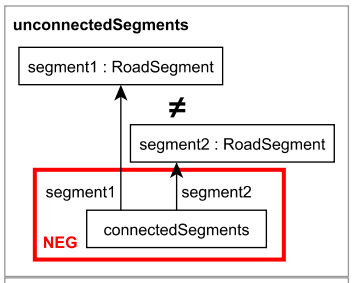
4.2.4. Útszakaszokra vonatkozó kényszerek

Ellenpélda Gráfminta

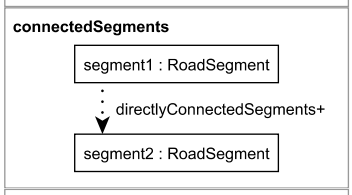



```
pattern emptyRoadSegment(roadSegment : RoadSegment) {
  neg find roadComponentOfRoadSegment(roadSegment, _);
}
```

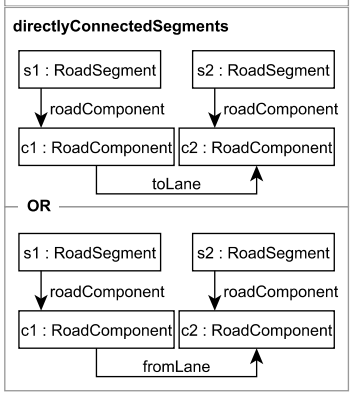
Létezik egy RoadSegment, melyben nem található RoadComponent.




```
pattern unconnectedSegments(segment1 : RoadSegment,
        segment2 : RoadSegment) {
  segment1 != segment2;
  neg find connectedSegments(segment1, segment2);
}
```

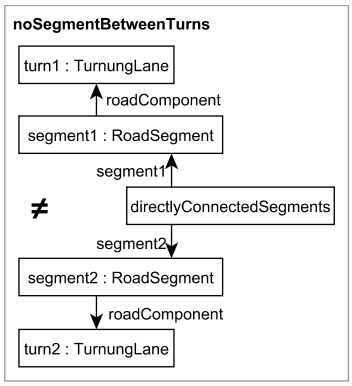



```
private pattern connectedSegments(segment1 : RoadSegment,
        segment2 : RoadSegment) {
  find directlyConnectedSegments+(segment1, segment2);
}
```




```
private pattern directlyConnectedSegments(segment1 : RoadSegment,
        segment2 : RoadSegment) {
  find roadComponentOfRoadSegment(segment1, component1);
  find roadComponentOfRoadSegment(segment2, component2);
  Lane.toLane(component1, component2);
} or {
  find roadComponentOfRoadSegment(segment1, component1);
  find roadComponentOfRoadSegment(segment2, component2);
  Lane.fromLane(component1, component2);
}
```

Egyik RoadSegment-ből nem lehet eljutni egy másik RoadSegment-be.




```
pattern noSegmentBetweenTurns(segment1 : RoadSegment,
        segment2 : RoadSegment) {
  segment1 != segment2;
  find roadComponentOfRoadSegment(segment1, turn1);
  find roadComponentOfRoadSegment(segment2, turn2);
  TurningLane(turn1);
  TurningLane(turn2);
  find directlyConnectedSegments(segment1, segment2);
}
```

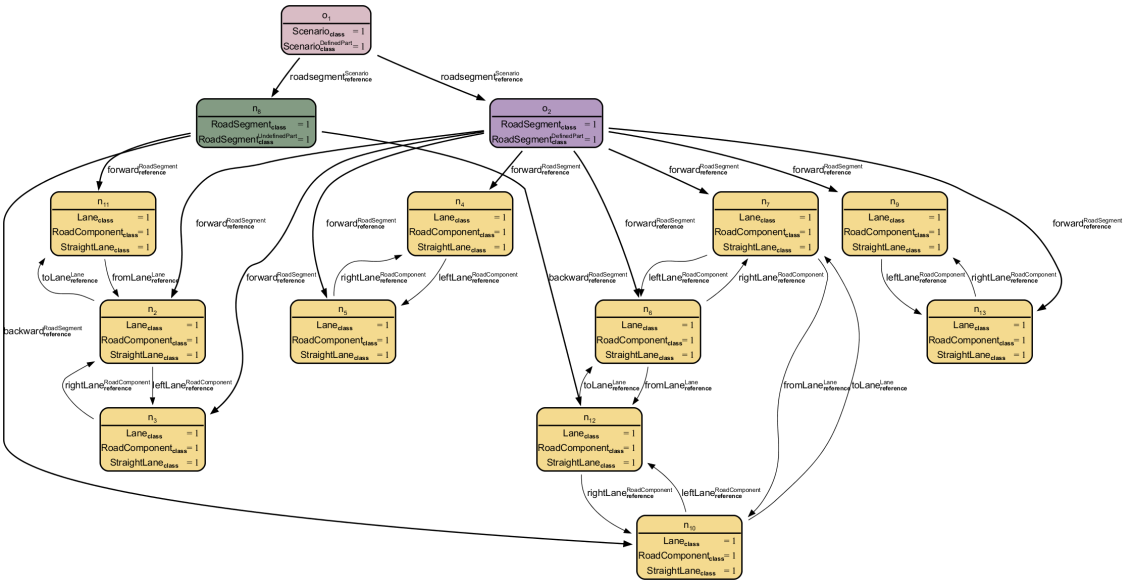
Két TurningLane-t tartalmazó RoadSegment közvetlenül kapcsolódik egymáshoz a bennük lévő Lane-ek toLane és fromLane referenciáin keresztül.

4.3. Tesztelrendezés generálása

Az absztrakt elrendezések generálását a Viatra Generator eszközzel [20] végzem. Ez a 3.1 ábrán látható módon paraméterként megkapja a metamodellt, valamint a strukturális kényszereket. Ezután lehetséges elrendezéseket generál, amikben egy logikai megoldó segítségével keres illeszkedéseket a megadott kényszerekre. Amennyiben talál illeszkedést, az elrendezést eldobja, egyébként pedig kiadja a kimenetén. A generátor biztosítja, hogy a kimeneten megjelenő elrendezések nem izomorfok, a metamodell és a kényszerek pedig arról is gondoskodnak, hogy a modellekben megjelenő különbségek lényeges eltérést jelentenek.

Az elkészült modellek `.xmi` formátumú fájlok, melyek szöveges formában tárolják a modellek felépítését. Az EMF lehetőséget nyújt ennek szerkesztésére, valamint különböző programozási nyelveken történő feldolgozására. Emellett a gráfokat `.gml` formátumú fájlokban is el tudja menteni, ami például a yEd [25] programmal megnyitva grafikus, szerkeszthető formában ábrázolja a gráfokat, a gráfgenerátor ezt a grafikus megjelenítést `.png` formátumban is képes elmenteni. Erre látható egy példa a 4.3 ábrán.

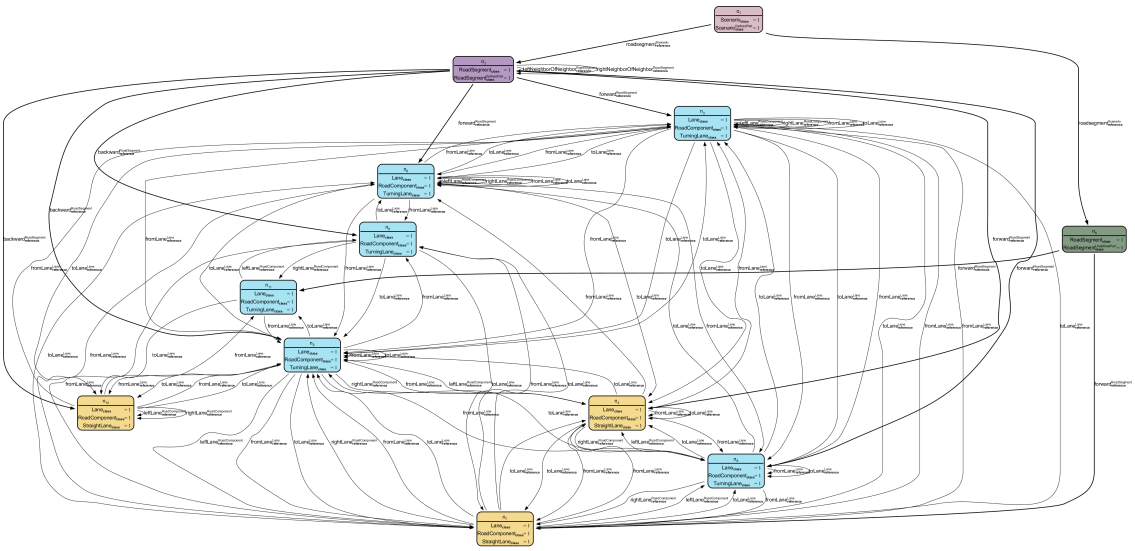
Annak érdekében, hogy a kényszerek hatékonyságát be tudjam mutatni, a 4.4 ábrán látható, kényszerek nélkül generált modellel hasonlítom össze a kimenetet. A 4.3 ábrán a sárga, a 4.4 ábrán pedig a sárga és a kék téglalapok jelölik a sávokat. A kényszerek nélkül generált modellben jól láthatóan véletlenszerűen lettek behúzva a különböző élek, semmilyen struktúra nincs a sávok között. Az ilyen úthálózatot nem lehet megjeleníteni, mert megszegi az alapvető geometriai kényszereket. Ezzel szemben a kényszerekkel generált modellben sokkal kevesebb él található az elemek között, és megfigyelhetünk különböző méretű egyenes útszakaszokat, a sávok között néhány kisebb-nagyobb kört, és ezek a szakaszok legfeljebb egy-egy helyen csatlakoznak egymáshoz.



4.3. ábra. Egy generált absztrakt elrendezés grafikus megjelenítése

4.4. Absztrakt elrendezés és OAS Szcenárió összehasonlítása

A generáláshoz használt metamodell elkészítése során az OAS megvalósítása is alapul szolgált. Fontos cél volt, hogy az OAS által definiált elrendezéseket az én módszeremmel is le lehessen írni. Ezen felül szerettem volna feloldani a modellek korlátozásait, ily módon lehe-



4.4. ábra. Egy kényszerek nélkül generált absztrakt elrendezés grafikus megjelenítése

tővé téve még változatosabb tesztelrendezések létrehozását. Egy három ágú kereszteződés példáján keresztül mutatom be, miben hasonlít és miben különbözik a két megközelítés.

Nézzük meg az OAS 3-2-S-I-STR-CAR:S>W azonosítójú szcenárióját. A 4.5a ábrán látható az OAS tesztkészletében található rajz, a 4.5b ábrán pedig ugyanennek az elrendezésnek az én metamodellem alapján készített modellje található.

4.4.1. Hasonlóságok

Az OAS szcenárió megadja, hogy a kereszteződésnek három ága van, valamint az utak kettő sávosak. Az általam létrehozott absztrakt modell ehhez hasonlóan tartalmaz három útszakaszt (RoadSegment), melyekben kettő-kettő sáv (Lane) található.

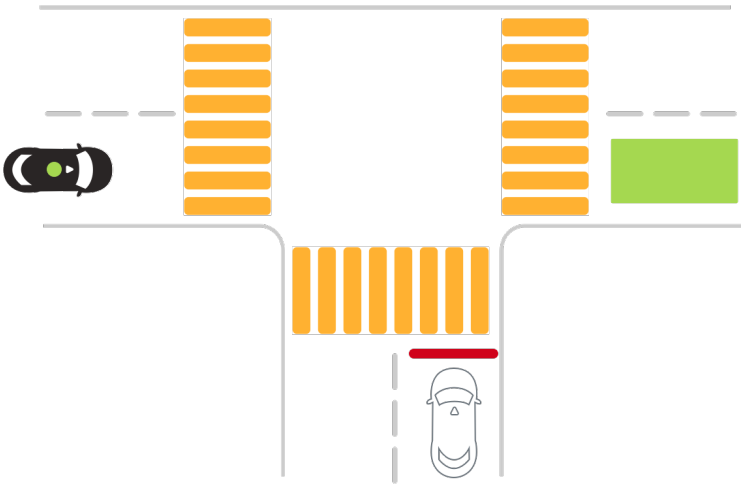
A gyalogátkelőhelyek az OAS szcenáriójában a kereszteződés minden ágánál megjelennek. Az én megoldásomban mindhárom útszakaszhoz létrehoztam egy-egy gyalogátkelőhelyet, melyek a Sign elemből származnak le. Egy referenciával adható meg, mely sávokra vonatkoznak.

A stoptáblák helyzete a szcenárió azonosítójában található. Ebben a példában a dél felől bejövő sávon van egy stoptábla. A metamodellben a gyalogátkelőhelyhez hasonlóan a Sign elemből származik le a stoptábla, az adott RoadSegment tartalmazza, és egy referencia adja meg, mely sávokra vonatkozik. Ebben az esetben a kereszteződésben lévő sávokra mutat a referencia, mivel az adott sávra lépés előtt lép életbe a tábla.

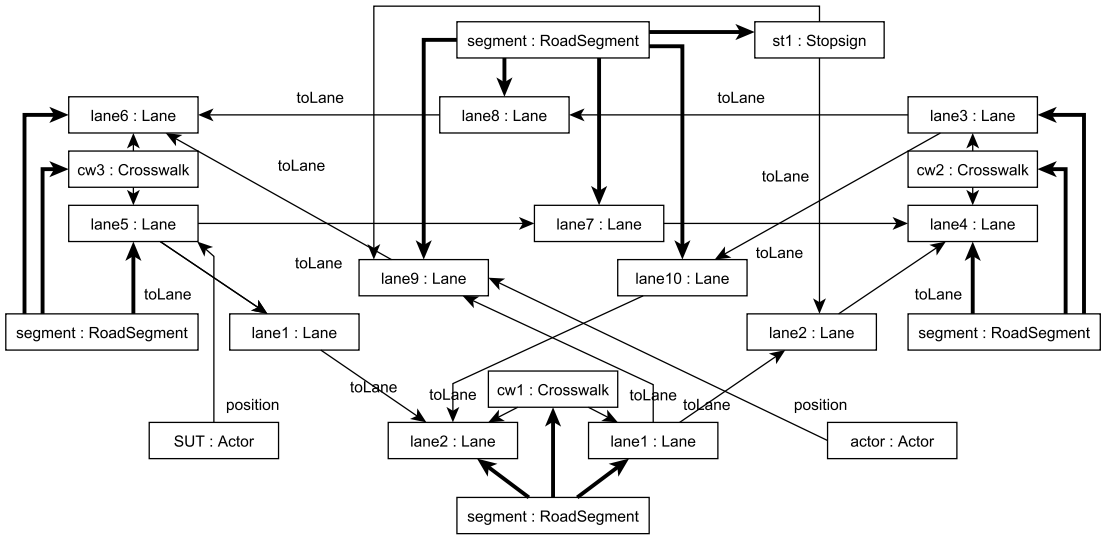
4.4.2. Különbségek

Az OAS szcenáriók mindig legfeljebb egy kereszteződést tartalmaznak és az útszakaszok száma az azonosítóban szerepel. Az azonosítóban csak négy irányt lehet megkülönböztetni (észak. dél, kelet, nyugat), ezért a szcenáriók is legfeljebb 4-ágú kereszteződéseket tartalmazhatnak. Ezzel szemben az általam megvalósított megoldásban egy modell több kereszteződést is tartalmazhat, melyekbe akármennyi útszakasz vezethet.

A kereszteződések esetében a haladási irányokat nem tudjuk megadni az OAS szcenárióiban, ezért a kanyarodás mindig minden irányban lehetséges, így nem tudjuk vizsgálni ezen szabályok betartását. Hasonló módon kanyarodósávokat sem tudunk létrehozni. A metamodellem alapján készült absztrakt elrendezésekben a kereszteződés belsejében is



(a) 3-ágú kereszteződés OAS alapján



(b) 3-ágú kereszteződés absztrakt modellje

4.5. ábra. T kereszteződés modellje

modellezzük a sávokat, vagyis a lehetséges menetirányokat, így lehetőségünk van kanyarodósávok modellezésére is.

Az OAS megvalósításában nincs lehetőség különböző számú sávból álló utakat egy szcenárióban elhelyezni, így nem tudunk például egysávos útról többsávos útra való kanyarodást leírni. Ezzel szemben az általam kidolgozott megoldással a különböző utak különböző számú sávot tartalmazhatnak, ezáltal sokkal változatosabb elrendezéseket tudunk létrehozni, valamint kanyarodósávokat is tudunk modellezni.

A tesztelt autonóm jármű az OAS szcenáriókban mindig nyugati irányból érkezik, az utakat ennek megfelelően kell elforgatni. A mozgás ezután lépésenként van leírva, és mindegyik lépéshez tartozik egy ábra, melyen látszódik az aktorok pozíciója. Az én absztrakt modelljeimben az orientációnak nincs szerepe, a vizsgált jármű és a többi aktor pozíciója is az út egy-egy elemével pontosan megadható. A megoldás jelenlegi formájában csak pillanatnyi állapotot képes leírni, de egy kis kiegészítéssel akár az OAS szcenárióhoz hasonlóan lehetőségünk van több időpillanatban megadni az aktorok elhelyezkedését.

5. fejezet

Kiértékelés

A dolgozatban bemutatott eredmények értékelése során az alábbi kérdésekre kerestem a választ:

K1 Hogyan skálázódik az absztrakt elrendezések generálása, ha egyre nagyobb modelleket generálunk? (méret skálázhatóság)

K2 Hogyan skálázódik a megoldásunk, ha növeljük a generált absztrakt elrendezések számát? (darabszám skálázhatóság)

5.1. Mérési elrendezés

A mérést egy átlagos laptopon végeztem[1], 16 GB heap memóriával. A méréseket eclipse fejlesztői környezetben végeztem. Annak érdekében, hogy a modellgenerátort bemelegítsem memóriakezelés és optimalizálás szempontjából, 5 extra futást végeztem minden kiértékelt mérés előtt, melyek futásidejét a kiértékelés során figyelmen kívül hagytam. A mérések során a generátor számára biztosított 16 GB memória minden esetben elegendő volt. A gráfgenerátor a Viatra Solver [20] logikai megoldót használta.

K1-hez 10 és 40 közötti elemszámú modelleket generáltam 5-ös lépésközzel. Mindegyik méretből 10 darabot generáltam egy-egy mérés során és a mérést ötször ismételtem meg. Az értékeléshez a mérési eredmények mediánját vettem figyelembe.

K2-höz 20, 30 és 40 elemű modelleket generáltam, mindegyikből 100-100 darabot (a 40 elemű modellek esetében csak 93 modell készült el az időkorláton belül), és mértem az egyes modellek elkészülése között eltelt időt. Ezen méréseket a hosszú futásidő miatt csak egyszer végeztem el.

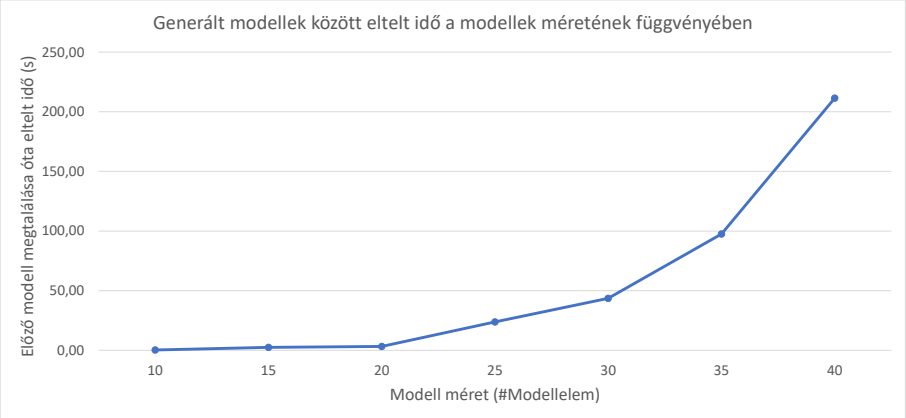
5.2. Mérési eredmények

K1 vizsgálata során az egyes mérések között csak kis különbségek voltak, és a K2-höz végzet mérés is azonos eredményeket mutat. Az 5.1 ábrán a vízszintes tengely a generált elrendezések méretét mutatja, a függőleges tengelyen pedig az adott méretű elrendezés generálása során két modell elkészülte között eltelt idő mediánja látható. Ezen az ábrán jól látható, ahogy a különböző méretű modellek generálásának idejei a modellek méretével nőnek. Míg 20 elemű modellekből 8,35 másodperc volt az elemek megjelenése közötti idő mediánja, addig 30 eleműeknél ez az érték 43,61 másodperc, 40 eleműeknél pedig közel 4 perc.

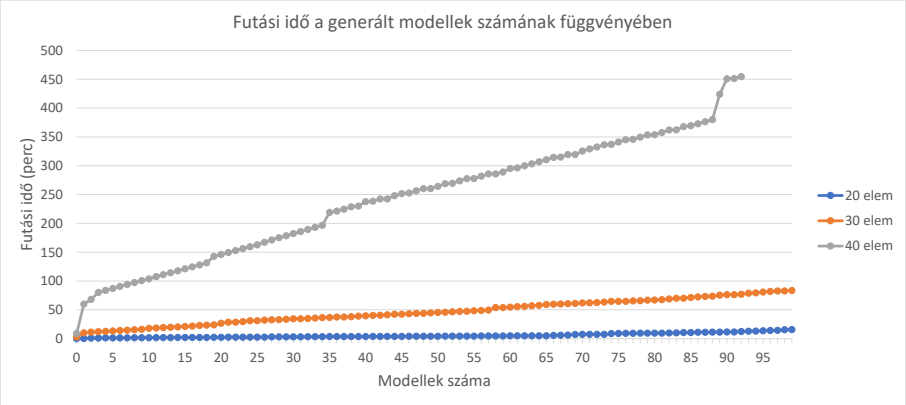
K2 esetében csupán egyszer végeztem el a mérést, azonban a hosszú futásidő alatt végig konzisztens volt a modellek elkészülte között eltelt idő. Az 5.2 ábrán a vízszintes

[1]CPU: Intel Core-i7-8550U, SSD, MEM: 20GB, OS: Windows 10 Pro

tengelyen a generált modellek száma látható, a függőleges tengely skálájáról pedig adott számú modell generálásához szükséges idő olvasható le. Az 5.2 ábrán jól látható, hogy a generált modellek száma egyenes arányosságot mutat a teljes futási idővel viszont az első néhány modell előállítása általában tovább tart, míg később az egyes modellek elkészülte között eltelt idő alacsony szinten marad néhány kiugrással felfelé. Amikor két modell elkészülése között kevés idő telt el, a modellek csak kis mértékben tértek el egymástól. Ha pedig ez az időkülönbség nagyobb az átlagosnál, a kapott modellek között jelentős az eltérés.



5.1. ábra. Két modell megtalálása óta eltelt idő a modellek méretének függvényében



5.2. ábra. Teljes futási idő absztrakt elrendezések generálása során

5.3. Eredmények kiértékelése

A kapott eredményeket elfogadhatónak tekintjük, mivel a vártaknak megfelelően a modellek számával egyenesen arányos a futásidő, aminek nagyságrendje szintén nem volt meglepetés, hiszen meglehetősen összetett kényszereknek kell megfelelniük a modelleknek. A több órás futásidő valójában nem probléma, mivel megfelelő metamodellt és kényszereket felírva a generálást csak egyszer kell elvégeznünk.

A modell méretével exponenciálisan növekszik a generálás ideje, azonban a vizsgált 30-40 elemű modellek már meglehetősen nagyok, és még ebben az esetben is elfogadható a futásidő. Az OAS eszközkészletében lévő szcenáriók többsége ennél kisebb, csak a nagyobb kereszteződések közelítik meg ezt a méretet.

K1 A tesztelrendezések generálása során a futási időt elsősorban a modellek mérete határozza meg, ami mérettel nő.
K2 A generált elrendezések száma egyenesen arányos a generálás teljes futási idejével, két modell elkészülte között a sokadik modell esetében is körülbelül annyi idő telik el, mint a generálás korai szakaszában.

5.4. Lehetséges mérési hibák

- Az első mérés során csupán ötször futtattam a méréseket, de a futásidők csak kis eltéréseket mutattak egymáshoz képest. A mérések során felmerülő mérési zajt medián számítással küszöböltem ki.
- A második mérést a hosszú futási idő miatt csak egyszer végeztem el, ezért ebben az esetben a mérési zajt nem tudtam kiküszöbölni, azonban a mért eredmények a várt lineáris kapcsolatot mutatják a generált modellek száma és a futási idő között mindhárom méret esetében.
- A mérést csak egy esettanulmányon végeztem el, de az kellő összetettségű ahhoz, hogy a skálázhatóságot vizsgálni tudjuk.

6. fejezet

Összefoglalás és jövőbeli tervek

A dolgozatban egy olyan gyakorlati módszert írtam le, mellyel autonóm járművek szisztematikus teszteléséhez generálhatók tesztelrendezések. Az absztrakt modell garantálja, hogy generált tesztelrendezések lényegi különbségeket tartalmaznak, a tesztkészlet teljessége és lefedettsége maximális. Egy második lépésként az absztrakt modellhez konkrét koordináták rendelhetők, így az elrendezések egy tesztszobában megépíthetők. Új strukturális kényszerek bevezetésével finomíthatjuk a követelményeket a tesztesetekkel szemben.

A tesztesetek generálásához különböző eszközöket használok fel. Absztrakt elrendezéseket a strukturális követelmények és a metamodell alapján gráfgenerátorral állítom elő. A szakdolgozat alapjául szolgáló TDK dolgozatban [5] szereplő folyamaton felül az Open Autonomous Safety tesztkészletének vizsgálatát és a két megközelítés összehasonlítását is tartalmazza ez a dolgozat.

A dolgozatban kidolgozott módszer egy eddig ritkán alkalmazott megközelítésre ad megoldást, mely egyszerűen adaptálható egyéb autonóm rendszerek vizsgálata során is. Egy összetett esettanulmányon bemutattam, hogy a módszer alkalmazható komplex tesztelrendezések előállításához is.

A jövőben bővíteni szeretnénk a metamodellt és a strukturális kényszereket úgy, hogy a tesztesetek több típusú modellelemet több attribútummal tartalmazzanak (például: közúti jelzések, több típusú aktor, relatív sebességek). Ugyan a megoldásunk összes lépése közepes méretű tesztelrendezések generálását másodpercek alatt elvégzi, szeretnénk minél nagyobb tesztkörnyezeteket is rövid idő alatt generálni. Szeretnénk integrálni az általunk generált tesztelrendezéseket a *BME Felsőoktatási Intézményi Kiválósági Program Járműintelligencia és Kommunikáció* alprojektje alatt fejlesztett szimulátorral.

Köszönetnyilvánítás

A dolgozatot részben az MTA-BME Lendület Kiberfizikai Rendszerek Kutatócsoport, valamint BME Felsőoktatási Intézményi Kiválósági Program / Mesterséges Intelligencia / Járműintelligencia és Kommunikáció alprojektje támogatta.

Irodalomjegyzék

[1] R. Ben Abdessalem – S. Nejati – L. C. Briand – T. Stifter: Testing vision-based control systems using learnable evolutionary algorithms. In *Int. Conf. on Software Engineering (ICSE)* (konferenciaanyag). 2018, 1016–1026. p.

[2] Krzysztof Czarnecki: On-road safety of automated driving system (ads) - taxonomy and safety analysis methods. 2018. 07.

[3] George Dimitrakopoulos: *The Future: Towards Autonomous Driving*. Cham, 2017, Springer International Publishing, 113–121. p. ISBN 978-3-319-47244-7.
URL `https://doi.org/10.1007/978-3-319-47244-7_6`.

[4] Federal Automated Vehicles Policy. Jelentés, 2016, National Highway Traffic Safety Administration. Available at `https://www.transportation.gov/AV/`.

[5] Attila Ficsor – Balázs Somorjai: Tesztelrendezések automatikus generálása autonóm járművek szisztematikus ellenőrzéséhez. Jelentés, 2019, Budapest University of Technology and Economics.

[6] The Eclipse Foundation: Domain Model Tutorial. `https://wiki.eclipse.org/Sirius/Tutorials/DomainModelTutorial`, 2019. [2019.10.21].

[7] The Eclipse Foundation: Eclipse Modeling Framework (EMF). `https://www.eclipse.org/modeling/emf/`, 2019. [2019.10.21].

[8] The Eclipse Foundation: The VIATRA Query Language Documentation. `https://www.eclipse.org/viatra/documentation/query-language.html`, 2019. [2019.10.21].

[9] Object Management Group: Object Constraint Language. `https://www.omg.org/spec/OCL/`, 2019. [2019.10.21].

[10] Philipp Helle – Wladimir Schamai – Carsten Strobel: Testing of autonomous systems – challenges and current state-of-the-art. *INCOSE International Symposium*, 26. évf. (2016) 1. sz., 571–584. p.

[11] Muhammad Zohaib Iqbal – Andrea Arcuri – Lionel Briand: Environment modeling and simulation for automated testing of soft real-time embedded software. *Software & Systems Modeling*, 14. évf. (2015) 1. sz., 483–524. p.

[12] Road vehicles — Functional safety — Part 1: Vocabulary. Standard, Geneva, CH, 2018. december, International Organization for Standardization.

[13] Nidhi Kalra – Susan M. Paddock: How many miles of driving would it take to demonstrate autonomous vehicle reliability? RR-1478-RC. Jelentés, 2016, RAND Corporation.

[14] Philip Koopman – Michael Wagner: Challenges in autonomous vehicle testing and validation. *SAE Int. J. Trans. Safety*, 4. évf. (2016) 1. sz.

[15] Alexey Kurakin – Ian J Goodfellow – Samy Bengio: Adversarial examples in the physical world. In *Artificial Intelligence Safety and Security*. 2018, Chapman and Hall/CRC, 99–112. p.

[16] István Majzik – Oszkár Semeráth – Csaba Hajdu – Kristóf Marussy – Zoltán Szatmári – Zoltán Micskei – András Vörös – Aren A. Babikian – Dániel Varró: Towards system-level testing with coverage guarantees for autonomous vehicles. In *IEEE / ACM 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)* (konferenciaanyag). 2019, IEEE, IEEE. URL `https://modelsconf19.org/`. New Ideas and Vision Track.

[17] Zoltán Micskei – Zoltán Szatmári – János Oláh – István Majzik: A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In *Agent and Multi-Agent Systems*. 2012, 504–513. p.

[18] Cu D. Nguyen – Simon Miles – Anna Perini – Paolo Tonella – Mark Harman – Michael Luck: Evolutionary testing of autonomous software agents. *Autonomous Agents and Multi-Agent Systems*, 25. évf. (2012) 2. sz., 260–283. p.

[19] Cu D. Nguyen – Anna Perini – Carole Bernon – Juan Pavón – John Thangarajah: Testing in multi-agent systems. In *Agent-Oriented Software Engineering X* (konferenciaanyag). 2011, 180–190. p.

[20] Oszkár Semeráth – András Szabolcs Nagy – Dániel Varró: A graph solver for the automated generation of consistent domain-specific models. In *40th International Conference on Software Engineering (ICSE 2018)* (konferenciaanyag). 2018. 5, ACM, 969–980. p.

[21] Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles, 2018.

[22] Cumhur Erkan Tuncali – Georgios Fainekos – Hisahiro Ito – James Kapinski: Sim-ATAV: Simulation-based adversarial testing framework for autonomous vehicles. In *Int. Conf. on Hybrid Systems* (konferenciaanyag). 2018, 283–284. p. ISBN 978-1-4503-5642-8. 2 p.

[23] Voyage Inc.: *Open Autonomous Safety*. 2019. `https://oas.voyage.auto/`.

[24] Waymo Safety Report. Jelentés, 2018, Waymo LLC. Available at `https://waymo.com/safety/`.

[25] yWorks GmbH: *yEd Graph Editor*. 2019. `https://www.yworks.com/products/yed`.

[26] Ágnes Barta: Abstract test data generation for autonomous and distributed systems. Jelentés, 2014, Budapest University of Technology and Economics.

[27] Ágnes Barta – Oszkár Semeráth: Consistency analysis of domain-specific languages. Jelentés, 2013, Budapest University of Technology and Economics.