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

Tesztelrendezések automatikus generálása autonóm járművek szisztematikus ellenőrzéséhez

TDK dolgozat

Készítette:

Ficsor Attila
Somorjai Balázs

Konzulens:

dr. Semeráth Oszkár

2019

Tartalomjegyzék

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.

Dolgozatunk célja egy olyan automatikus tesztelrendezés generátor megvalósítása, amely hatékonyan képes támogatni autonóm járművek szisztematikus tesztelését. Célunk továbbá, hogy a különböző előállított szituációk valóban lényeges különbségeket tartalmazzanak, ezzel garantálva a tesztesetek megfelelelő 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. Az így előálló tesztelrendezések kipróbálhatóak automatizált szimulátorban, vagy megépíthetőek tesztszobában. Dolgozatunkban elért eredményeket egy esettanulmányon szemléltetjük.

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

Abstract

Nowadays components 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 is why they have to meet high safety standards. However, state-of-the-art testing techniques were unable to support the automated testing and verification of these components are not known.

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

In this report, we propose 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. The test layouts produced this way can be tested in an automatized simulator or can be built in a test room. In our report the achieved results will be illustrated in a case study.

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

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 [6]. 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 [15, 20, 19], 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 vizsálata 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 bizosítani, hogy valóban minden szituációban az elvárt döntést hozza a rendszer.

1.3. Autonóm komponensek szisztematikus tesztelése

A munkánk elsődleges célja egy olyan módszer kidolgozása, amely lehetőséget biztosít fizikai tesztelrendezések automatikus és szisztematikus generálására. Az algoritmus kimenete tesztelrendezések halmaza konkrét koordinátákkal, ami felhasználható kritikus rendszerek autonóm komponenseinek szimulátoros teszteléséhez.

1.4. Tesztelrendezések automatikus generálása

A dolgozatban bemutatunk 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.

A javasolt megközelítést egy esettanulmányon mutatjuk be, melyben autonóm autók tesztelrendezéseit vizsgáljuk a 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

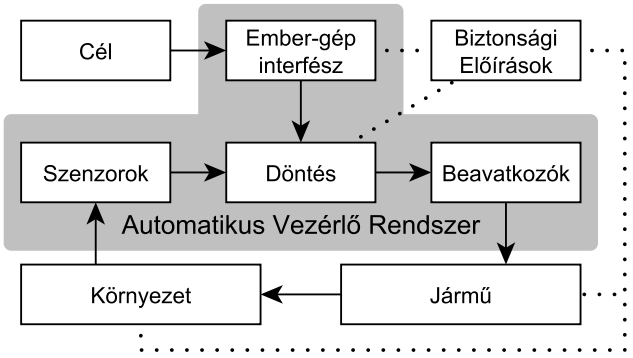
A dolgozat hátralévő része a következőképpen épül fel. A 2. fejezetben összefoglaljuk a munkánkhoz szükséges legfontosabb technikai és elméleti háttértudást. A 3. fejezetben egy rövit áttekintést nyújtunk a javasolt megoldás felépítéséről és működéséről. Ezután a 4. fejezet az absztrakt elrendezések generálásának módszerét mutatja be. A tesztesetek konkretizálásáról az 5. fejezetben lesz szó, majd a 6. fejezetben megvizsgáljuk a megoldás skálázhatóságát. Végül a 7. fejezetben összefoglaljuk a munkánkat.

2. fejezet

Előismeretek

2.1. Autonóm járművek biztonsági előírásai

Az autonóm komponenseket tartalmazó járművek alapvető felépítését, valamint a velük szemben támasztott követelményeket [4], [17] és [22] alapján tárgyaljuk. 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 [4] 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.

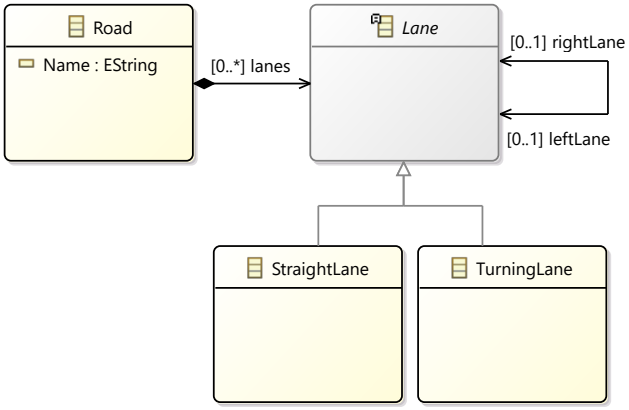
4. *Közlekedési szabályok*: A közlekedési szabályok formális, többnyire biztonság növelését célzó szabályok, melyeket jogilag tartatnak be egy földrajzi területen.

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 [22], mi feltételezzük, hogy azokat már kielégítették, és [22]-ben felvázolt elképzelésnek megfelően a rendszer-szintű 3-as és 4-es követelményekre koncentrálunk. Ezen felül feltételezzük, hogy a szenzoros érzékelés és a beavatkozók megbízhatóan működnek. Az általunk bemutatott megközelítés elsődleges célja olyan tesztkészlet előállítása, amely olyan hibákat is észre tud venni amik a szenzorok helyes működése mellett következtek be. Végül feltételezzük, hogy a követelmények biztosítását elsődlegesen teszteléssel és szimulációval ellenőrzik.

2.2. Szakterület-specifikus nyelvek

A szakterület-specifikus nyelvek célja, hogy alkalmazási területekhez modellezési nyelvet, valamint hatékony eszköztámogatást nyújtsanak. Ilyen szakterület-specifikus modellezési nyelveket, valamint adatatmodelleket például az Eclipse Modeling Framework (EMF) [9] 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.



2.2. ábra. Egy út metamodellje

A metamodell fogalmát és letrehozását [8], valamint az EMF általunk használt részeit egy esettanulmányon keresztül mutatom meg, mely egy út leegyszerűsített metamodelljét írja le, ez látható a 2.2 ábrán.

Alapvetően az alábbi elemekből és speciális változataikból építhetjük fel az adatmodellünket:

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 tettük. Ilyenkor belőlük nem, csak a leszármazottaikból tudunk példányt létrehozni. A Lane esetében kettő specializációt hoztunk 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.

2.3. 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) [12] és a VQL (VIATRA Query Language) [10]. Ezekben a minták megfogalmazásához felhasználhatók a metamodellben definiált típusok, referenciák, stb. Az említett nyelvek közül a VQL-t választottuk, mivel az általunk 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 kényszerereket. 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 kere-

sünk, 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, 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. Egy $G = (V, E)$ irányított gráf tranzitív lezártja az a $G' = (V, E')$ gráf, amelyben u-ból v-be pontosan akkor vezet él, ha u-ból vezet irányított út v-be az eredeti G gráfban. A mi esetünkben, mivel több különböző él található a gráfban, meg kell adnunk, milyen típusú élből álló irányított út kell, hogy vezessen u-ból v-be. VQL esetében ezek az élek az általunk definiált minták lehetnek.

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 tudi, 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.

2.4. Satisfiability modulo theories nyelv és megoldók

SMT (Satisfiability modulo theories) problémák az informatikában és matematikában elsődrendű logikai és numerikus formulákon értelmezett, döntési vagy kényszerkielégítési problémák. Ezeket a problémákat leírhatjuk az SMT szabványos nyelvén, melyet az SMT-LIBv2 definiál. A probléma definiálása után ezt a leírást egy megoldónak (SMT-Solver) bemenetéül adhatjuk, ami eredményül megmondja, hogy a felírt kényszerek kielégíthetőeke, és ha igen, akkor egy modellt is le lehet kérdezni. A modellben az összes változóra és függvényre kaphatunk egy értéket, ami mellett a kényszerek kielégülnek.

SMT-LIBv2 [3] az SMT problémák szabványos leíró nyelve. Formátuma szöveges, egyszerűen szerkeszthető külön eszköz vagy IDE nélkül. A szintaxisa úgy van kialakítva, hogy egyszerűen lehessen programból generálni. Az egyenleteket és egyenlőtlenségeket lengyel formában [13, 1] kell felírni, ezért kézzel szerkesztve elsőre nehezen átlátható. A szöveges fájl elején különböző paramétereket adhatunk a megoldónak. Ilyen lehet például az, hogy milyen logikát használjon a megoldáshoz (például: "Quantifier-free real arithmetic").

```
1 (set-logic QF_UF)
```

Azonban nekünk kell utánajárni, hogy melyik megoldó milyen logikát támogat. Az inicializáció után deklarálhatjuk, hogy a probléma-térben milyen konstansok és függvények vannak.

Példákon keresztül megismerhetjük az SMT-LIBv2 nyelvet. Deklaráljunk egy "a" nevű konstanst. Deklaráljunk egy "f" egész visszatérési értékű függvényt, ami egy egész számot, és egy logikai (Bool) értéket vesz paraméterül.

```
1 (declare-const a Int)
2 (declare-fun f (Int Bool) Int)
```

Függvényeket nem csak deklarálhatunk, hanem definiálhatunk is, azaz explicit megadhatjuk, hogy adott bemenetre milyen kimenetet ad a függvény. Például definiálhatunk egy függvényt, ami a paraméterül kapott "time" valós számra 1-gyel tér vissza, ha az "availabilitybeg" és "availabilityend" között van, egyébként 0.

```
1 (define-fun active((time Real)) Int
2     (ite(and(< availabilitybeg time)(< time availabilityend))
3             1 0 ))
```

Ebben a függvény definiálásában használtuk a "ite" (if than else) elágazást, "and" logikai operátort, illetve a "<" relációs operátort. Mindezt természetesen lengyel formában adtuk meg. A konstansok és függvények deklarálása és definiálása után adhatjuk meg a különböző kényszereket. Ezek a kényszerek minden esetben elsőrendű logikai kifejezések. A megoldó a kényszerek kielégíthetőségét vizsgálja. Ezeket a kényszereket az "assert" kulcsszóval írhatjuk le.

```
1 (assert (> a 10))
2 (assert (< (f a true) 100))
```

Utána futtassuk a megoldót és ha kielégíthetőek a kényszerek, kérjünk egy modellt egy lehetséges megoldásról. Példaképpen az alábbi bemenetre futtassuk a megoldót:

```
1 (declare-const a Int)
2 (declare-fun f (Int Bool) Int)
3 (assert (> a 10))
4 (assert (< (f a true) 100))
```

```
(check-sat)
(get-model)
```

A Z3 SMT-Solver [5] erre a bemenetre az alábbi kimenetet adja:

```
sat
(model
    (define-fun a () Int
        11)
    (define-fun f ((x!1 Int) (x!2 Bool)) Int
        (ite (and (= x!1 11) (= x!2 true)) 0
            0))
)
```

A kimenetet értelmezve megtudhatjuk, hogy a kényszerek kielégíthetőek, azaz "sat". Egy példa megoldást tartalmazó modellt is lekértünk, amiben az 'a' konstans megegyezik egy konstans 11 értékű függvénnyel. Az "f" függvénynél kicsit érdekesebb, de korrekt megoldást kapunk. Az első paraméter ha 11 és a második igaz, akkor a függvény értéke 0, különben is 0. Visszaellenőrizve láthatjuk, hogy ez a megoldás tényleg teljesíti a kényszereket. A nyelv ennél sokkal mélyebb részleteket is tartalmaz, ahol valamit ezeken túl felhasználtunk, arra az adott résznél külön kitérünk.

2.5. Módszerek koordináták modellhez rendeléséhez

2.5.1. Hozzárendelések összehasonlítása

Hozzárendelések vizsgálatánál két típusú kényszert különböztetünk meg, hard- és soft constraint-et [7]. Elsőként meg kell állapítani, hogy a hozzárendelés helyes-e, azaz az összes hard constraint-nek teljesülnie kell. Az összes megoldási módszer használatához szükségünk van eldönteni két helyes megoldás közül, hogy melyik jobb. Ezt kiértékelésnek (evaluation-nek) nevezzük, ekkor a soft constraint-ek alapján döntjük el két helyes megoldásról, hogy melyik a jobb.

Az lenne a legegyszerűbb, ha egy megoldáshoz egy számot tudnánk rendelni, és ezeket a számokat hasonlítjuk össze. Ezt két ellentétes irányból tehetjük meg. Vagy azt határozzuk meg, hogy az adott megoldás mennyire költséges (cost), vagy azt hogy mennyire jó (fitness). A problémától függ, hogy melyiket érdemes választani. Kiértékelés szempontjából lényegi különbség nincs a kettő között amennyiben a relációs jeleket megfordítjuk.

Az a szám ami a költséget vagy jóságot jellemzi egyszerűsített esetben akár a nem teljesült kényszerek száma is lehet. A különböző soft-constraint-ek között súlyozást is fel lehet állítani.

Előállhat olyan eset, hogy nem egy kész megoldásra kell meghatározni a költségeket, ezt parciális kiértékelésnek [29] nevezzük. Ilyenre akkor van szükség, ha például egy modellt iteratív lépésekben építünk fel, és ha már az építés közben látszik, hogy nem lehet optimálisabb, mint egy másik megoldás, akkor be sem kell fejezni az építését hasonlóan egy visszalépéses keresésnél.

2.5.2. Lokális optimumra törekvő hozzárendelés

Ezek olyan megoldások, amelyek valamilyen szempontokból optimális megoldást nyújtanak. Nem céljuk a tökéletes megoldás felkutatása. Csak bizonyos szempontokat figyelembe véve heurisztikus módszerrel képeznek egy megoldást. Általában kevés általánosítással, és problémaspecifikusan implementáltak.

Egy közismert ilyen elven alapuló megoldással rendelkező probléma például az utazóügynök problémája [18].

2.5.3. Globális optimumra törekvő hozzárendelés

2.5.3.1. Lehetséges permutációk vizsgálata

A legtriviálisabb globális optimumra törekvő megoldás a összes lehetőség vizsgálata. Ez a probléma méretéből adódóan csak elvi megoldást jelenthet, kiszámítása túl sok ideig tartana. Ezt a módszert nem is részletezzük tovább.

2.5.3.2. Numerikus megoldók használata

Az SMT megoldók egy kényszer-halmaz kielégíthetőségét vizsgálják. Változókat deklarálhatunk és definiálhatunk, melyek konstansok és függvények is lehetnek. A kényszereket elsőrendű logikával adhatjuk meg. A problémát leíró változók és kényszerek halmazát egy szöveges fájlban, vagy bizonyos megoldók esetén futásidejű könyvtárakkal adhatjuk meg.

Amennyiben a modell kényszereiből sikerült transzformálni egy ilyen leírást, a megoldó nem csak a kielégíthetőségét mondja meg a feltételeknek. Ha kielégíthető, lehet kérni egy olyan modellt (olyan értékeket), amelyekre teljesülnek a kényszerek.

2.5.3.3. Metaheurisztikus módszerek, genetikus algoritmus

Sokszor nincs lehetőségünk arra, hogy a legjobb megoldást megkeressük. Optimalizálásra használhatunk metaheurisztikus módszereket, amelyek azt ígérik, hogy viszonylag rövid idő alatt egy elfogadhatóan jó megoldást nyújtanak. Mivel a probléma teljes terét nem ismerhetjük, így csak a már ismert megoldásokhoz tudjuk hasonlítani a kapott eredményt.

A metaheurisztikus módszerek egy gyűjtőfogalom, amely több megközelítésben is eltérő csoportot tartalmaz. Ebben a dokumentumban leginkább „természet inspirált” populáció alapú genetikus algoritmussal foglalkoztunk. A populáció tagjai az egyedek, amelyek egy-egy lehetséges(vagy nem lehetséges) megoldást reprezentálnak. A populáció egyedein végzett különböző genetikai módosítások, mint például mutáció, keresztezés változtatják az egyedeket. A génállományból (genotípus) meghatározzuk az egyedek konkrét megtestesülését (fenotípus). Egy fenotípust össze tudunk hasonlítani másik fenotípusokkal, és így a gyengébb egyedeket például kiszórhatjuk a populációnkból, hogy helyükre a jobb egyedek valamilyen változtatás utáni „leszármazotta” kerülhessen. Ettől a módszertől azt várjuk, hogy a természetes szelekció hatására az idő múlásával egyre jobb megoldást kapjunk.

2.5.3.4. Összehasonlítás

Fontos kérdés lehet, hogy amikor adunk egy megoldást, amiről azt gondoljuk, hogy egészen jó, akkor az mégis mit jelent. Milyen garanciát tudunk adni?

Az előzőekben bemutatott módszerek közül a megoldás explicit kiszámítása adja a legnagyobb garanciát, hiszen pontosan meghatározzuk a legoptimálisabb megoldást. Ezért mondhatjuk azt, hogy a megoldás garantáltan optimális.

Az SMT megoldók esetén tudunk mondani minden független költségre vagy összköltségre egy felső becslést az optimumtól való eltérésre. Egy adott költségre (minél kisebb, annál optimálisabb) meg tudunk határozni egy c1 költséget, ami a legjobb megoldásunk költsége, és egy c2 költséget, ami már nem tud teljesülni. Minden esetben $c1 \leq c2$. Az adott költségre vonatkozó garanciánk c1-c2. Ez azt jelenti, hogy a globális optimális megoldás legfeljebb c1-c2-vel jobb.

A metaheurisztikus módszerek közvetlenül nem alkalmasak arra, hogy bármilyen garanciát adjanak. Ezek az algoritmusok minél tovább futnak, annál jobb megoldást adnak, vagy legalábbis nem rontanak a korábbin. Jó lenne tudni, hogy mikor érdemes leállítani. Mivel önmagukban erre nem képesek, bizonyos esetekben hibrid megoldások lehetnek cél-

ravezetőek. Az optimalizálást metaheurisztikus módszerrel végezzük, de a garanciát (egy felső becslést) egy másik módszerrel határozzuk meg.

A megoldási módszer kiválasztásánál figyelembe kell vennünk azt is, hogy az eredeti modellt át kell transzformálni egy olyan leírássá, vagy újabb modellé, ami az adott algoritmus bemenete. Át tudjuk-e transzformálni az összes kényszert, illetve ennek a transzformációnak mekkora a költsége?

Fontos összehasonlítási szempont lehet az adott megoldási módszer teljesítménye, illetve skálázódása. Egyáltalán nem mindegy, hogy mennyi idő alatt kapunk megoldást. Elképzelhető olyan extrém eset, hogy mire a megoldás kiszámítódik, az adott probléma már nem is áll fenn, vagy nem úgy áll fenn.

Alapvetően arra vagyunk kíváncsiak, hogy a probléma mérete és a futásidő között milyen összefüggés van. De az is kérdés lehet, hogy új kényszerek felvételével hogyan változik a megoldás kiszámításának ideje.

A futásidőn túl még egyéb erőforrás-igényeket is vizsgálhatunk. A teljesítményt vagy egyáltalán a futtathatóságot például a számításokhoz és eredmények tárolásához felhasznált memória is korlátozhatja.

2.6. Kapcsolódó munkák

A kapcsolódó munkákat a [22] összefoglalása alapján rendszerezzük.

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 [21]. 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 [27, 25]).

Rendszerszintű megközelítések Önvezető autók szokásos viselekdésének formális verifikációját javasolták [28]-ban. A közlekedési szituációkat és a járművek lehetséges reakcióit egy absztrakt szinten verifikálják. Széles körben használnak szimulációt, hogy autonóm rendszerek rendszerszintű viselkedését vizsgálják [28]. Azonban a változatos tesztelrendezések elkészítése még nincs megoldva.

Rendszerszintű elrendezés generálás Számos tesztelési megközelítés [2, 16, 23, 24, 33, 32] 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 függvénykent 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

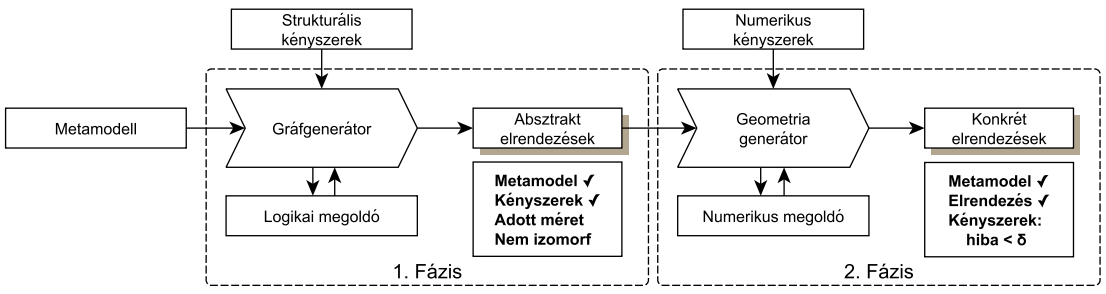
Ahogy korábban említettük, számos tesztpálya-generálási megközelítést javasol a szakirodalom, különböző célokkal és garanciákkal. Dolgozatunkban bemutatott módszer leginkább az Open Autonomous Safety [30] szabványban bemutatott elrendezésekhez illeszkedik, melynek célja az önvezető komponensek funckcionális tesztelése. Szisztematikus tesztelésnél a mi célunk egy módszer kidolgozása, mellyel (egy ekvivalencia-partícionáló módszerrel) az összes különböző tesztelrendezést elő tudjuk állítani, valamint a tesztesetek fedését is tudjuk vizsgálni.

3.1. Tesztelrendezés ekvivalencia particionálása

Kritikus rendszerek tesztelésénél fontos metrika a tesztlefedettség. Autonóm járművek szisztematikus tesztelését tipikusan egy előre definiált, manuálisan összerakott tesztkészlet alapján végzik (mint például [30]). Ilyen tesztkészleteknél nehezen meghatározható, hogy az összes lehetséges szituációnak, amelybe a jármű kerülhet, hány százalékát fedik le.

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-partícionálás. 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: az első fázisban absztrakt elrendezéseket készítünk, a második fázisban pedig ezekhez rendelünk koordinátákat, így elkészítve a konkrét elrendéseket.

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 struktúrális kényszereknek megfelelnek-e a modellek. Amennyiben hibás modellt talál, eldobja azt, és csak a kényszereknek megfelőeket 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, bővebben: 2.4). 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).

3.3. Tesztelési módszerek

A dolgozatunkban bemutatott 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 **robosztusságát** terjesztjük ki.

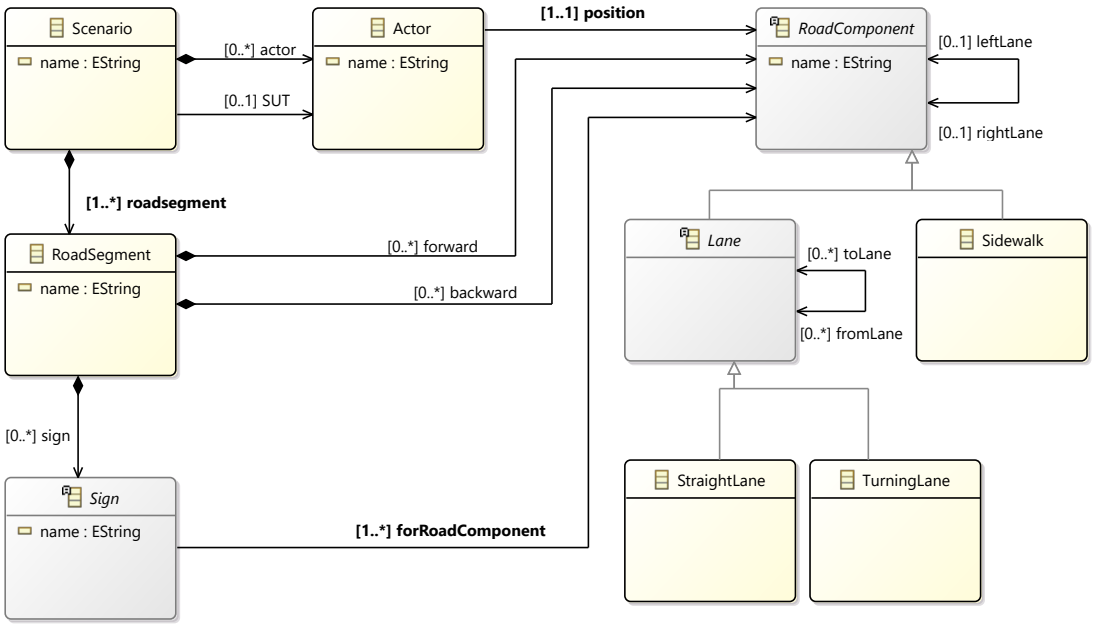
Előfordulhat, hogy olyan teszteset generálódik, ami a valóságban nem megépíthető, nem tesztelhető. Ebben az esetben **finomíthatjuk** úgy a követelményeket, hogy bevezethetünk új struktúrális vagy numerikus kényszereket, amikkel javítjuk a tesztesetek generálását.

4. fejezet

Absztrakt elrendezések generálása

4.1. Tesztelrendezés metamodellje

A dolgozat céljának eléréséhez szükséges az utak lehetséges elrendezéseinek egy absztrakt leírására, ehhez hoztunk 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 [30] vettük alapul. Ellenőriztük, hogy az abban összegyűjtött tesztesetek a mi megoldásunkkal is leírhatók-e, és általánosítottuk őket, hogy méret- és komplexitásbeli korlátok nélkül tudjuk őket modellezni. Az elkészült metamodell a 4.1 ábrán látható.



4.1. ábra. A tesztelrenezések 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 tesztelerendezést tartalmaz.

4.1.2. Actor – Tesztelrendezés szereplői

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

4.1.3. RoadSegment – Úthálózat felépítése

A tesztelrendezések legfőpbb 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 kereszterődés 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, hogy melyik úton van, a RoadSegment-et tovább kell bontanunk. A RoadComponent elemek leginkább sávszakasszokat 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.

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 sávszsakasz 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) és a kanyarodó (TurningLane) sávok, a modellek létrehozása során különböző szabályok vonatakoznak rájuk. (Itt a kanyarodó sáv elsősorban a kereszeződések 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 segítíségével 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 vezettünk be. Így viszont lehetőség adódik olyan elrendezések létrehozására, melyek az úthálozat 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 patternlanesInSameSegmentAndDirection(
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 }
```

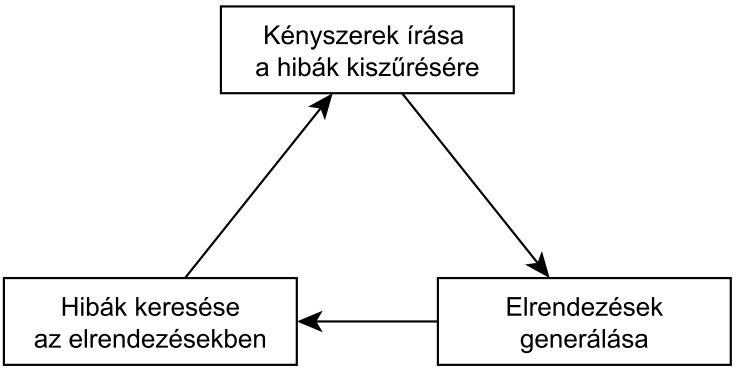
Néhány ilyen szabály triviális, másokat viszont nem egyszerű megtalálni. A kényszereket egy iteratív módszerrel lehet finomítani, ez a 4.2 ábrán látható.

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:

- Sávokra vonatkozó általános kényszerek
- Egyenes sávokra vonatkozó kényszerek
- Kanyarodó sávokra vonatkozó kényszerek
- Útszakaszokra vonatkozó kényszerek

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.

Egy Lane toLane referenciája saját magára mutat.

```
1 pattern leadsToItself(lane: Lane) {
2   Lane.toLane(lane, lane);
3 }
```

Egy RoadComponent rightLane referenciája saját magára mutat.

```
1 pattern nextToItself(comp: RoadComponent) {
2   RoadComponent.rightLane(comp, comp);
3 }
```

A modellben rightLane referenciákból álló kör található.

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

Egy Lane mellett és mögött ugyanaz a Lane található.

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

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

```
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);
}
```

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

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

private pattern lanesSameSegmentAndDirection(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 hozzá toLane éllel kapcsolódó másik Lane ugyazon RoadSegment azonos menetirányú sávjai.

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

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

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

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

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

Párhuzamos, azonos menetirányú sávok eltérő RoadSegment-be vezetnek.

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

Egy Lane nem kapcsolódik semmihez.

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

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

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

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

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

4.2.2. Egyenes sávokra vonatkozó kényszerek

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

```
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);
}
```

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

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

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

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

4.2.3. Kanyarodó sávokra vonatkozó kényszerek

TurningLane-nek van rightLane vagy leftLane referenciája.

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

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

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

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

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

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

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

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

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

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

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

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

4.2.4. Útszakaszokra vonatkozó kényszerek

Létezik egy RoadSegment, melyben nem található RoadComponent.

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

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

```
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);
}
```

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

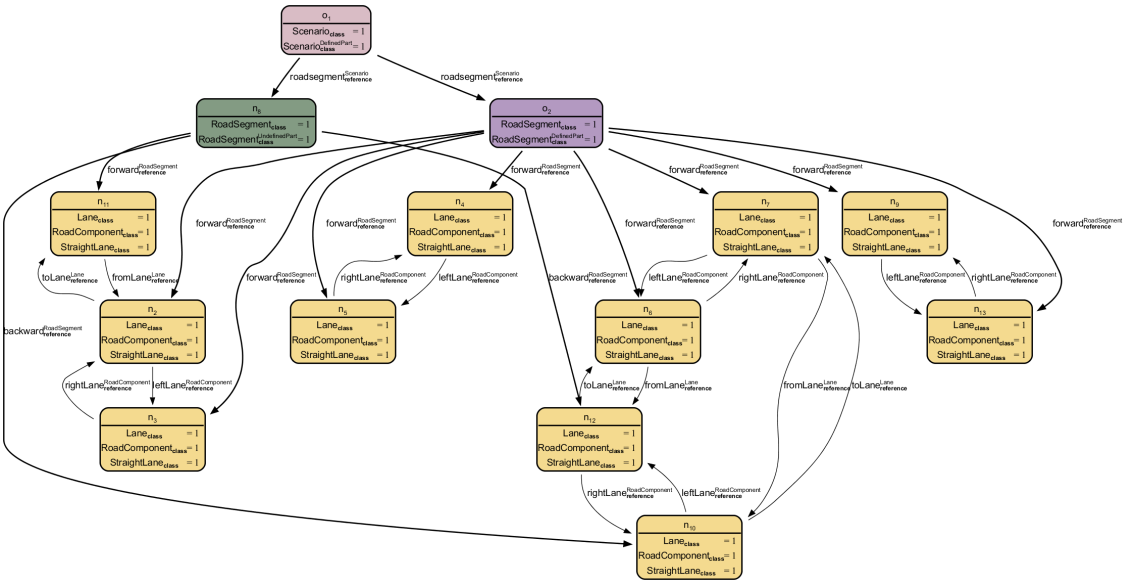
```
pattern noSegmentBetweenTurns(segment1 : RoadSegment, segment2 : RoadSegment) {
  segment1 != segment2;
  find roadComponentOfRoadSegment(segment1, turn1);
  find roadComponentOfRoadSegment(segment2, turn2);
  TurningLane(turn1);
  TurningLane(turn2);
  find directlyConnectedSegments(segment1, segment2);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

4.3. Tesztelrendezés generálása

Az absztrakt elrenedzések generálását a Viatra Generator eszközzel [26] végezzük. 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 gondoskdnak, hogy a modellekben megjelő 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` fromátumú fájlokban is el tudja menteni, ami például a yEd [31] programmal megnyitva grafikus, szerkeszthető formában ábrázolja a gráfokat, a gráfgenerátor ezt a grafikus megjenítést `.png` formátumban is képes elmenteni. Erre látható egy példa a 4.3 ábrán.



4.3. ábra. Egy absztrakt elrendezés grafikus megjelenítése

5. fejezet

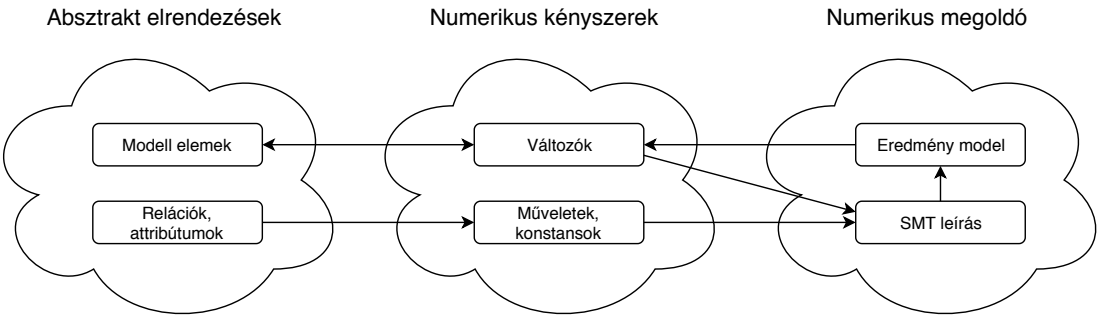
Tesztesetek konkretizálása numerikus megoldókkal

A tesztelrendezések generálásának második fázisában (ahogy a 3.1 ábra bemutatja) az absztrakt elrendezéseket konkretizáljuk. Az absztrakt elrendezések korábban megismert modelljének elemeihez konkrét koordinátákat rendelünk. A megoldáshoz a megismert SMT numerikus megoldókat használjuk fel. A második fázis elkezdése előtt az absztrakt elrendszések rendelkezésre állnak. A numerikus megoldó numerikus kényszereken tud dolgozni, ezért az absztrakt elrendezéseket valamilyen numerikus kényszerek alapján tovább kell transzformálni. Minden modellelemet el kell helyezni, ebben a megoldásban az elrendezést síkban végezzük. Eredményként konkrét koordinátákat kapunk, egy modell elem térben is kiterjedhet, ezért nem csak egy, hanem több koordinátát is el kell helyezni.

5.1. Geometriai modell és absztrakt modell kapcsolata

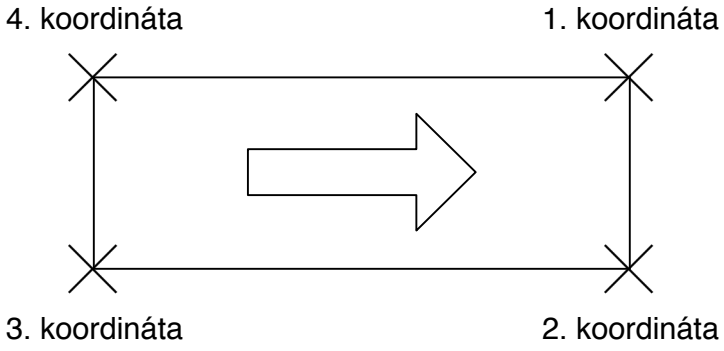
A generálás második fázisában absztrakt elrendezésekből kell konkrét koordinátákat kapni. Ebből adódik, hogy a kényszerek a konkrét koordinátákra vonatkoznak, azaz a modell elemekhez koordináták lesznek a numerikus kényszerekben deklarált változók. Ezek alapján a megfeleltetés egyértelmű és bi-direkcionális az absztrakt elrendezések modell elemei és a numerikus kényszerekhez használt változók között.

Az absztrakt elrendezésekben a modell elemek között értelmezett relációk határozzák meg azt, hogy a koordináták között milyen numerikus és logikai műveleteket kell elvégezni a változókon, azaz mi a konkrét kényszer az SMT leírásban. A modellben a modell elemekhez négy attribútumot rendelünk, ami a négy sarokpontjának a koordinátáját jelöli. Az elrendezés szempontjából releváns attribútumok is hasonlóan a relációhoz, a koordinátákra kényszereket határoznak meg.



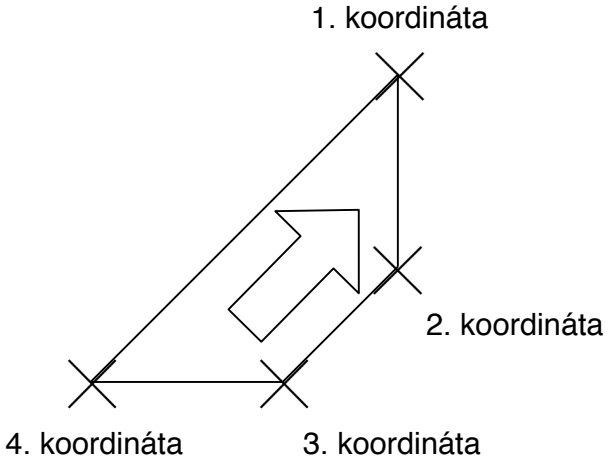
5.1. ábra. Leképzési irányok

A teljes generálási folyamat a következőképpen néz ki (lásd 5.1. ábra). Bemenetként absztrakt elrendezéseket várunk, melyekben vannak modell elemek, rajtuk értelmezett relációk, attribútumok. A modell elemekhez a típusuktól függően deklarálunk megfelelő számú változót. A transzformációt útszakaszokra vizsgáljuk részletesebben. Egy útszakaszt a megoldásunkban négy koordináta párral jellemzünk, és minden útszakasz a valóságban egy menetiránnyal rendelkező forgalmi sáv vagy annak egy részének leképzése (lásd 5.2. ábra). Az útszakasz négy pontját óramutató járásával megegyező irányban számozzuk. Az 1-es és 2-es pontok közötti szakasz a sáv vége, a 3-as és 4-es közötti szakasz a sáv eleje. Ebből következik, hogy az útszakasz menetirány szerinti bal széle az 1-es és 4-es közötti szakasz, a jobb széle pedig a 2-es és 3-as pontok közötti szakasz. Egy egyenes sáv ami nem kanyarodik viszonylag jól reprezentálható négy koordinátával.



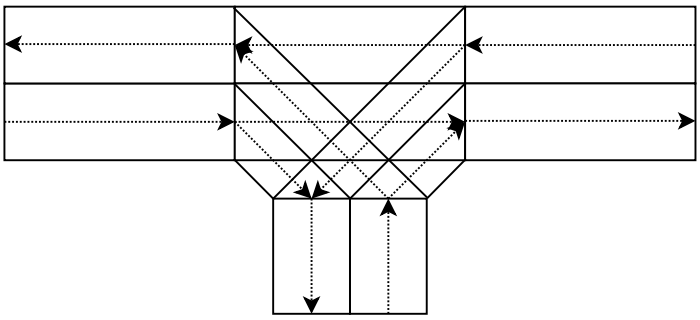
5.2. ábra. Egyenes sáv reprezentáció

A forgalmi sávok azonban kanyarodhatnak is. A megoldásunkban a kanyarodás ívét nem modellezzük, így a kanyarodó útszakaszokat is négy koordinátával jellemezzük. Az 1-es és 2-es koordináta közötti szakasz a sáv vége, a 3-as és 4-es közötti szakasz a sáv eleje. Ezek között pedig úgy tekintjük, hogy a kanyarnak nincs íve, azaz a 1-es és 4-es koordináták és 2-es és 3-as koordináták között is egyenes van.



5.3. ábra. Jobb kanyar reprezentáció

Ezzel a módszerrel viszonylag jól reprezentálhatóak bonyolultabb útszakaszok is úgy, hogy a pontok száma nem lesz túl nagy, ami fontos, hiszen egy új pont két új változót (x és y koordináta) jelent a kényszerek teljesülésének vizsgálatánál. Egy ilyen bolyolultabb példa lehet a "T" (vagy három irányú) elágazás (5.4 ábra), ahol három irányból érkezik és három irányba távozik forgalom. Kanyarodni bármelyik irányból bármelyikbe lehet.



5.4. ábra. T elágazás

Ahogy látszik is, a mi modellünkben ez 12 útszakasszal valósítható meg. Irányonként egy bejövő, és egy kimenő forgalmi sáv összesen 6. Ezeket páronként összekötve minden irányból még 6 sávot kapunk. Ez alapján úgy ítéljük meg, hogy kellőképpen bonyolult forgalmi szituációk is reprezentálhatóak megfelelő részletességgel.

5.2. Geometriai kényszerek leképzése

A kényszerek olyan elsődrendű logikával felírt kifejezések, melyeket teljesítenie kell a konkrét elrendezéseknek. Ezek a kényszerek azok, amiket a numerikus megoldónak "`assert`"-ként adunk be. A kényszerek sokfélék lehetnek. Vannak amik az absztrakt modellből transzformált kényszerek, de lehetnek olyan kényszerek is, ami a modell elem típusából adódik. Ezekre példát később adunk.

5.2.1. Változók deklarálása

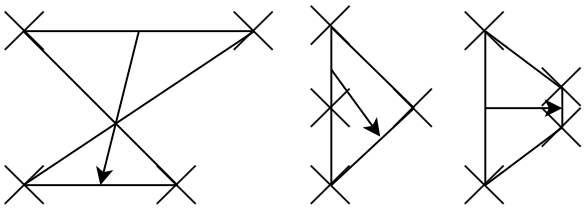
Változók deklarálása szükséges a a kényszerek felírásához. Ezeket a modell elemeiből tudjuk származtatni. A forgalmi sávoknál ezek a változók a 5.1 részben ismertetettek alapján négy pontra, azaz összesen nyolc valós számként jelennek meg (`lane1x1-lane1y4`). Egy forgalmi sávra kigenerálva:

```
1 (declare-fun lane1x1 () Real)
2 (declare-fun lane1x2 () Real)
3 (declare-fun lane1x3 () Real)
4 (declare-fun lane1x4 () Real)
5 (declare-fun lane1y1 () Real)
6 (declare-fun lane1y2 () Real)
7 (declare-fun lane1y3 () Real)
8 (declare-fun lane1y4 () Real)
```

5.2.2. Modell elem típusából adódó kényszerek

Modell elem típusából adódó kényszerek nem triviálisan adódnak, és tapasztalatunk szerint leginkább az eredmény vizsgálata után állíthatóak elő. A matematikai megoldó nem tudja, hogy forgalmi sávokra írjuk fel az egyenleteket, és azt sem, hogy hogyan néz ki egy forgalmi sáv. Ezért ha hiányoznak kényszerek, akkor tapasztalatunk szerint nem realisztikusan kinéző, megvalósíthatatlan forgalmi sávokat kapunk (5.5 ábra).

Szükséges olyan kényszerek bevezetése is, ami a forgalmi sáv kinézetére vonatkozik, de sajnos több okból sem szabályozhatunk túl. Ha túl sok szabályt (kényszert) veszünk fel, akkor nagyon lekicsinyítjük a leképezhető absztrakt elrendezések számát. Továbbá minden



5.5. ábra. Nem realisztikus sávok

szabállyal létrejövő kényszer lassíthatja a megoldás kiszámítási idejét [14]. Egy jó egyensúlyt kell megtalálni, ami nem szabályoz túl, és nem is túl szabad. A legtöbb solver nem támogat alapból szögfüggvényeket, ezért a forgalmi sávok szélességét és átlóinak hosszát szabályozva kaptunk számunkra megfelelő eredményt. Egy forgalmi sávra kigenerálva:

```
1  (assert (= 100000.0
2     (+ (* (- lane1x1 lane1x2) (- lane1x1 lane1x2))
3        (* (- lane1y1 lane1y2) (- lane1y1 lane1y2)))))
4  (assert (= 100000.0
5     (+ (* (- lane1x3 lane1x4) (- lane1x3 lane1x4))
6        (* (- lane1y3 lane1y4) (- lane1y3 lane1y4)))))
```

Az első két kényszer a forgalmi sáv négyzetes szélességét rögzíti. A realisztikusság, esztétikusság és egyszerűsítés érdekében rögzítettük a forgalmi sávok szélességét.

```
1  (assert (<= 100000.0
2      (+ (* (- lane1x1 lane1x4) (- lane1x1 lane1x4))
3         (* (- lane1y1 lane1y4) (- lane1y1 lane1y4)))))
4  (assert (<= 100000.0
5      (+ (* (- lane1x2 lane1x3) (- lane1x2 lane1x3))
6         (* (- lane1y2 lane1y3) (- lane1y2 lane1y3)))))
7  (assert (>= 400000.0
8      (+ (* (- lane1x1 lane1x4) (- lane1x1 lane1x4))
9         (* (- lane1y1 lane1y4) (- lane1y1 lane1y4)))))
10 (assert (>= 400000.0
11     (+ (* (- lane1x2 lane1x3) (- lane1x2 lane1x3))
12        (* (- lane1y2 lane1y3) (- lane1y2 lane1y3)))))
```

A következő négy kényszer a forgalmi sáv job és bal oldalát határoló szakaszok négyzetes hosszát korlátozza egy intervallumra.

```
1  (assert (<= 200000.0
2      (+ (* (- lane1x1 lane1x3) (- lane1x1 lane1x3))
3         (* (- lane1y1 lane1y3) (- lane1y1 lane1y3)))))
4  (assert (<= 200000.0
5      (+ (* (- lane1x2 lane1x4) (- lane1x2 lane1x4))
6         (* (- lane1y2 lane1y4) (- lane1y2 lane1y4)))))
7  (assert (>= 500000.0
8      (+ (* (- lane1x1 lane1x3) (- lane1x1 lane1x3))
9         (* (- lane1y1 lane1y3) (- lane1y1 lane1y3)))))
10 (assert (>= 500000.0
11     (+ (* (- lane1x2 lane1x4) (- lane1x2 lane1x4))
12        (* (- lane1y2 lane1y4) (- lane1y2 lane1y4)))))
```

Az utolsó négy kényszer a forgalmi sáv átlóinak négyzetes hosszát korlátozza egy intervallumra. A megoldók alapértelmezetten nem támogatják a szögfüggvényeket, ezért

választottuk ezt a megoldást. Egy forgalmi sávra vonatkozólag sok kinézeti kényszernek tűnhetnek, de ezek csak a legszükségesebbek.

5.2.3. Modell elemek közötti kapcsolatokból adódó kényszerek

Modell elemek közötti kapcsolatokból adódó kényszerek közül csak kettőt vizsgálunk részletesebben. Forgalmi sávoknál fontos, hogy melyik sáv melyikbe csatlakozik (lásd: 4.1.5). Ez a reláció a sávok között viszonylag kevés kényszerrel megoldható, az egyik forgalmi sáv végét jelentő 1-es és 2-es pontok koordinátája megegyezik a rákövetkező sáv elejét jelentő 4-es és 3-as pontokkal páronként (5.2 és 5.3 ábra). Ha a "lane1" sáv rákövetkezője a "lane2" sáv, akkor a kényszerek kigenerálva az alábbiak:

```
1 (assert (= lane1x1 lane2x4))
2 (assert (= lane1y1 lane2y4))
3 (assert (= lane1x2 lane2x3))
4 (assert (= lane1y2 lane2y3))
```

Ennél a kapcsolatnál be kellett vezetni egy nem triviális kényszereket is, hogy a forgalmi sávok elrendezése realisztikusabb legyen. Ezek a kényszerek arra irányulnak, hogy a rákövetkező forgalmi sáv is hasonló irányban folytatódjon, mint a megelőző.

```
1 (assert (ite (< lane1x4 lane1x1)
2     (< lane2x4 lane2x1)
3     (< lane2x1 lane2x4)))
4 (assert (ite (< lane1y4 lane1y1)
5     (< lane2y4 lane2y1)
6     (< lane2y1 lane2y4)))
```

A következő kényszer amit vizsgáltunk, az az amikor két forgalmi ellenkező irányba halad, és menetirány szerinti bal oldalon illeszkednek egymással (lásd: 4.1.5). Itt is hasonlóan a rákövetkezéssel, pontokat páronként illesztünk. Az egyik sáv 1-es, és a másik sáv 4-es koordinátájával megegyezik, és a másik irányból is szimmetrikusan.

```
1 (assert (= lane1x1 lane2x4))
2 (assert (= lane1y1 lane2y4))
3 (assert (= lane1x4 lane2x1))
4 (assert (= lane1y4 lane2y1))
```

5.3. Megoldás keresése fix építőelemekre

A problémát meg lehet oldani úgy is, hogy előre definiált elemeket használunk. Ezzel a leképezhető megoldások számát csökkentjük, viszont a megoldások fix építőelemei realisztikusnak választhatók. Eddig egy viszonylag nagy szabadsággal rendelkező útszakasz volt, amire különböző realisztikusságot szabályzó kényszereket írtunk. Ebben az esetben viszont előre definiáljuk, hogy hogyan néznek ki azok az építőelemek, melyeket lehet használni egy absztrakt modell realizációjánál. Ilyen építőelemek lehetnek a különböző irányba néző egyenesek, jobb és bal kanyarok eltérő ívekkel. Minél több építőelem van, annál sokfélébb megoldást lehet képezni számítási idő növekedésének árán.

Az előre definiált építőelemek sokkal kevesebb kényszert tartalmaznak darabonként, mint a nagyobb szabadsági fokkal rendelkező általános útszakasz. Ha az építőelemeket úgy definiáljuk, hogy nincs szabaságuk forogni, akkor egy referencia pontot kell tudni csak meghatározni, és a többi pontja ehhez képest csak egy eltolás. Az is lehetséges, hogy egy

elemként definiáljuk a forgatás után egybevágó elemeket, akkor egy elforgatási változónak is szabadnak kell lennie.

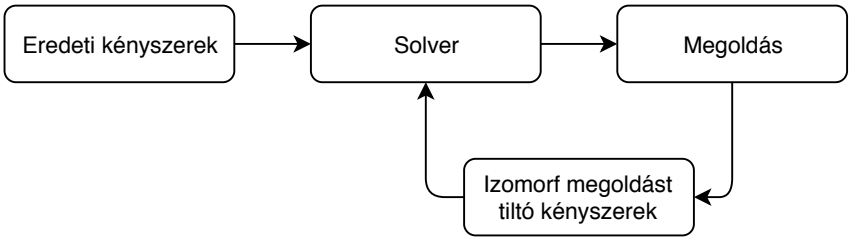
A modell építést visszalépeses keresés [11] alapon implementáljuk. Minden lépésben egy elemre választjuk ki, hogy milyen típusú legyen. A kiválasztott típusból és az elemből származó kényszereket is adjuk hozzá a korábbi kényszerekhez. Ha a bővített kényszerek kielégíthetők maradnak, akkor lépünk a következő elemre. Itt is végigpróbáljuk az összes típust, ha azt tapasztaljuk, hogy egyik típussal sem lesz kielégíthető, akkor visszalépünk az előző elemre, és próbáljuk tovább a típusokat, különben lépünk a következő elemre. Ha az első elemnél sem jó egyik típus, és vissza kellene lépni, akkor nem kielégíthető a megadott építőelemekkel.

5.4. Alternatív hozzárendelések leképzése

Tudhatjuk-e, hogy egy absztrakt modellhez létezik-e egynél több leképzés? Könnyen belátható, hogy egy végtelen térben absztrakt modellhez végtelen koordinátákra való leképzés tartozik, ha legalább egy tartozik. Például úgy lehet egy új megoldást képezni, hogy az összes előző megoldásból származó pontot ugyan úgy toljuk vagy forgatjuk el. Az ilyen szempontból izomorf megoldásokat lényegében nem tekintjük új, vagy az előzőtől különböző megoldásnak.

A numerikus megoldó alap logikáinak eredményét determinisztikusnak tekintjük. A tapasztalat azt mutatja, hogy ugyanarra a bemenetre ugyanazt a kielégítő modellt adja minden lefutás alkalmával. Ezért nem számíthatunk arra, hogy egy újabb lefutás egy új eredményt fog adni.

Alternatív hozzárendeléseket úgy tudunk képezni, hogy egy meglévő megoldáshoz hasonló megoldást nem engedélyezünk új kényszerek bevezetésével. Több lépésben futtatva a numerikus megoldót, ha megtartjuk az eredeti kényszereket és a korábbi izomorf megoldásokat tiltó kényszereket, minden eredmény új, nem izomorf megoldás lesz (5.6 ábra). Ezeknek a kényszereknek a megfogalmazása nem triviális, és azt is jól kell definiálnunk, hogy mit tekintünk izomorfnak.



5.6. ábra. Alternatív megoldások generálása

Egy jó kiindulási pont, hogy egyik változót sem engedjük felvenni egy már korábbi megoldásban felvett érték $r > 0$ sugarában. Útszakaszok generálásánál ezen túl izomorfnak értjük egy korábbi megoldás eltolását, forgatását és minden tükrözését és ezek összes kombinációját. Ezek a kényszerek mind felírhatók matematikai formulákkal, ezért SMT kényszerekké alakíthatók.

5.5. Implementáció

Az absztrakt modell szöveges fájlból felolvasása után a kényszereket forráskódból generálva hozzuk létre a **Z3** java API-n keresztül. Ez lehetőséget ad java kódban felépített modell alapján SMT-LIBv2 fájl (lásd: 2.4) generálására, és kódból is meg lehet hívni a **Z3** numeri-

kus megoldót. Az eredményt két numerikus megoldóval teszteltük, **Z3** és **dReal**. A **dReal** megoldóhoz nem tartozik API, ezért a kigenerált SMT fájlt használjuk bemenetnek.

A két megoldó közötti fő különbség a bizonyítás miként je, a **Z3** egy egzakt megoldást ad, egy konkrét számra bizonyít, ezzel szemben a **dReal** megoldásként egy olyan számot ad, aminek egy δ környezetében van olyan megoldás, ami kielégíti az egyenleteket. A **Z3** hatékonyan oldja meg a logikai egyenleteket a matematikai egyenleteken túl, illetve a nem kielégíthetőséget is tudja bizonyítani. A **dReal** hatékonysága a valós számokon értelmezett bonyolult matematikai kifejezéseken alapuló kielégíthetőség vizsgálatában kiemelkedő.

6. fejezet

Kiértékelés

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

K1.1 Hogyan skálázódik az absztrak elrendezések generálása, ha egyre nagyobb modelleket generálunk? (méret skálázhatóság, I. fázis)

K1.2 Hogyan skálázódik a megoldásunk, ha növeljük a generált absztrakt elrendezések számát? (darabszám skálázhatóság, I. fázis)

K2 Hogyan skálázódik a konkrét elrendezések generálása, ha egyre nagyobb modelleket generálunk? (méret skálázhatóság, II. fázis)

6.1. Absztrakt teszteset generálás

6.1.1. Mérési elrendezés

A mérést egy átlagos laptopon végeztük[1], 16 GB heap memóriával. A mérés során három különböző méretű modellból generáltunk 100-100 darabot, valamint 7 különböző mértűből 10-10 darabot. A generátor a Viatra Solver logikai megoldót használta [26].

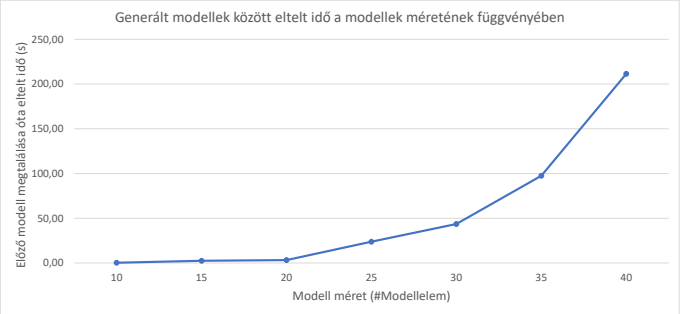
6.1.2. Mérési eredmények

K1.1-hez 10 és 40 közötti elemszámú modlleket generáltunk 5-ös lépésközzel. Mindegyik méretből 10 darabot generáltunk és mértük az egyes modellek elkészülése között eltelt időt, melyeknek a mediánját vettük figyelembe. A mérést 5-ször ismételtük meg és vettük a mérések mediánját. A 6.1 ábrán a vizszntes 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.

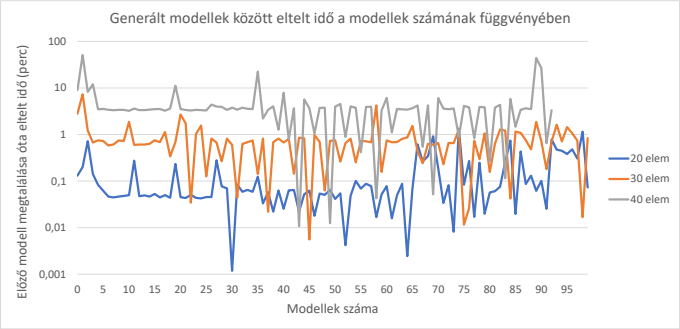
K1.2-höz 20, 30 és 40 elemű modellekből generáltunk 100-100 darabot (a 40 elemű modellek esetében csak 93 modell készült el az időkorláton belül), és mértük az egyes modellek elkészülése között eltelt időt. A 6.2 ábrán a vízszintes tengelyen a generált modellek száma látható, a függőleges tengely logaritmikus skálájáról pedig adott számú modell generálása után két modell elkészülte közötti idő olvasható le. A 6.2 ábrán jól látható, hogy a generált modellek száma egyenes arányosságot mutat a teljes futási idővel viszont az

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

első néhány modell előállítása általában tovább tart, míg később az egyes modellek elkészülte között eltelt idő adott szint környékén marad néhány kiugrással felfelé és számos kiugrással lefelé. 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.



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



6.2. ábra. Egy-egy modell elkészítésének ideje absztrakt elrendezések generálása során

6.1.3. Eredmények kiértékelése

A kapott eredményeket elfogadhatnek tekintjük, mivel a vártaknak megfelelően a modellek számával egyenesen arányos a futásidő, nagyságrendje pedig szintén nem volt meglepetés, hiszen meglehetősen összetett kényszereknek kell megfelelniük a modelleknek.

K1.1 A tesztelrendezések generálása során a futási időt elsősorban a modellek mérete határozza meg, ami mérettel nő.

K1.2 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.

6.2. Konkrét teszteset generálása

6.2.1. Mérési elrendezés

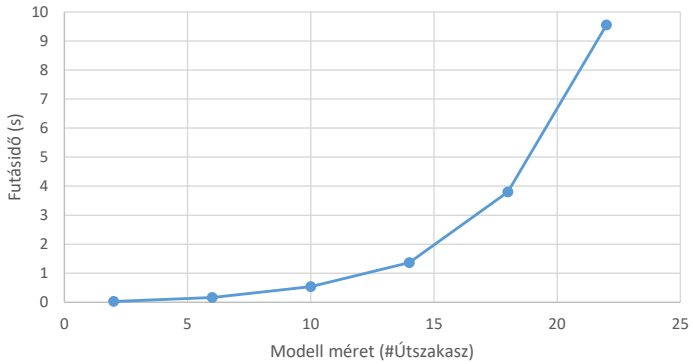
A mérést egy átlagos laptopon végeztük[2]. A mérés során absztrakt elrendezés bemenetként 2-22 hosszúságú egymás utáni útszakaszokat konkretizáltunk 4-es léptékkel, azaz rendeltünk útszakaszokhoz konkrét koordinátákat. Szakaszonként 8 változót kell kitöltenie a

[2]Intel i7-5600U, 16GB RAM, SSD

megoldónak, melyekre szakaszonként 10 távolságkényszert és 2 egyenlőségkényszert kell betartania. A mérést dReal megoldóval futtattuk.

6.2.2. Mérési eredmények

K2-hez a 6.3 diagramon a futásidőt vizsgáltuk a méret (útszakaszok száma) függvényében. A vízszintes tengelyen az útszakaszok száma található, melyek a 5.2 fejezetben leírt kényszereket tartalmazzák. A függőleges tengely generálás idejét mutatja másodpercben.



6.3. ábra. Forgalmi sávok konkretizációjának futásideje

6.2.3. Eredmények kiértékelése

K2 A futásidő a mérettel exponenciálisan nő. Egy valóságban előállított tesztelrendezés, ahol az útszakaszok száma $n < 22$ kevesebb mint 10s alatt konkretizálhatók. Ezzel a teljesítménnyel alkalmas a feladatra.

7. fejezet

Összefoglalás és jövőbeli tervek

A dolgozatban egy olyan gyakorlati módszert írtunk le, mellyel autonóm járművek szisztematikus teszteléséhez generálhatók tesztelrendezések. Az absztrakt modell garantálja, hogy generált tesztelrendezések lényegi különbségeket tartalmaznak, a tesztkészlet teljessége és lefedettsége maximális. A generált tesztelrendezések konkrétak, koordinátákkal ellátottak, egy tesztszobában megépíthetők. Kidolgoztunk módszert egy kigenerált teszthez hasonló alternatív tesztesetek generálására, ezzel a tesztek robosztusságát növelve. Új struktúrális és numerikus kényszerek bevezetésével finomíthatjuk a követelményeket a tesztesetekkel szemben.

A tesztesetek generálásához különböző eszközöket használunk fel. Absztrakt elrendezéseket a struktúrális követelmények és a metamodell alapján gráfgenerátorral állítünk elő. Az absztrakt elrendezés modelljének elemeihez és relációihoz numerikus kényszereket rendelünk, melyet egy szabványos formátummá transzformálunk, amit több, cserélhető numerikus megoldóval is meg lehet oldatni.

A dolgozatban kidolgozott módszer egy eddig ritkán alkalmazott megközelítésre ad megoldást, mely egyszerűen adaptálható egyéb autonóm rendszerek vizsgálata során is.

Megvizsgáltuk milyen módszerek lehetségesek a tesztesetek konkretizálásánál, ezeket összehasonlítottuk, és a numerikus megoldók használatánál döntöttünk.

A jövőben bővíteni szeretnénk a metamodellt, struktúrális kényszereket és numerikus 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

Szeretnénk köszetet mondani konzulensünknek, Dr. Semeráth Oszkárnak, aki rendelkezésünkre bocsátotta a dolgozat elkészítéséhez szükséges szakirodalmat, valamint észrevételeivel és tanácsaival segítette munkánkat.

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] Clark Barrett – Pascal Fontaine – Cesare Tinelli: The SMT-LIB Standard: Version 2.6. Jelentés, 2017, Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.

[2] 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.

[3] David R Cok és mások: The smt-libv2 language and tools: A tutorial. *Language c*, 2011., 2010–2011. p.

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

[5] Leonardo De Moura – Nikolaj Bjørner: Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (konferenciaanyag). 2008, Springer, 337–340. p.

[6] 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.

[7] Carmel Domshlak – Steve Prestwich – Francesca Rossi – Kristen Brent Venable – Toby Walsh: Hard and soft constraints for reasoning about qualitative conditional preferences. *Journal of Heuristics*, 12. évf. (2006) 4-5. sz., 263–285. p.

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

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

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

[11] John Gaschnig: A general backtrack algorithm that eliminates most redundant tests. In *IJCAI* (konferenciaanyag). 1977, 457. p.

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

[13] Charles L Hamblin: Translation to and from polish notation. *The Computer Journal*, 5. évf. (1962) 3. sz., 210–213. p.

[14] Andrew Healy – Rosemary Monahan – James F Power: Predicting smt solver performance for software verification. *arXiv preprint arXiv:1701.08466*, 2017.

[15] 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.

[16] 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.

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

[18] David S Johnson – Lyle A McGeoch: The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, 1. évf. (1997) 1. sz., 215–310. p.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] Frank Van Harmelen – Alan Bundy: Explanation-based generalisation= partial evaluation. *Artificial Intelligence*, 36. évf. (1988) 3. sz., 401–412. p.

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

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

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

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