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

Tesztkörnyezetek előállítása automatikus szabályalapú modellgenerálás segítségével

Tudományos Diákköri Konferencia Dolgozat

Készítette
Sólyom Alexandra Anna

Konzulens
Nagy András Szabolcs
Semeráth Oszkár
Szárnyas Gábor

2016. október. 27.

Tartalomjegyzék

Kivonat

Napjainkban a kritikus rendszerek (pl. autók, repülőgépek szoftverei) elterjedésével egyre szélesebb körben alkalmaznak modellvezérelt tervezést. A modellvezérelt tervezés során számos fejlesztési lépést automatizált kódgenerálással váltanak fel, ami megfelelő eszközök segítségével megkönnyíti, hatékonyabbá és megbízhatóbbá teszi a komplex rendszerek tervezését és megvalósítását.

Azonban egy komplex modellező eszköz helyes és hatékony megvalósítása már önmagában kihívást jelent, ezért ezek az eszközök ma még nem igazán aknázzák ki a bennük rejlő lehetőségeket. Továbbá a teljesítmény és skálázhatóság következetes mérése, illetve az objektív benchmarkok hiánya is problémát jelent. A ma használt tervező eszközökön tipikusan nem végeznek teljesítmény méréseket.

Dolgozatom célja egy módszer kidolgozása, amely különböző paraméterezéssel képes diverz, helyes vagy skálázható modellek generálására, így lehetőséget nyújt a modellező eszközök átfogó tesztelésére.

A dolgozatomban bemutatom, hogyan lehet a szabályalapú tervezésitér-bejárásra épülő keretrendszer segítségével helyes, sokszínű vagy nagy méretű modelleket generálni. Ennek során részletezem (i) a tervezésitér-bejárás alapú modellnövesztés módszerét, ezen belül (ii) az absztrakció alapú modellnövesztés megközelítését, (iii) a modellnövesztés probléma lehetséges felbontásait kisebb problémákra a gyorsabb generálás érdekében, illetve (iv) a probléma különböző felbontásainak összehasonlítását és értékelését. A módszerhez (v) készítettem egy prototípus alkalmazást, amely a Viatra Design Space Exploration, (VIATRA-DSE) keresésitér-bejáró keretrendszerre épül. Ennek működését a (vi) train benchmarkon prezentálom.

Az általam készített módszer alkalmas modellező eszközök tesztelésének támogatására, benchmarkok készítésére és összehasonlító mérések segítségével a megfelelő modellező eszköz kiválasztásának megkönnyítésére.

Abstract

Nowadays, as a consequence of wide-spreading safety critical systems (e.g., automobiles and airplanes) model-driven development gains an increasing role in software development. Through model-driven development (MDD) many development steps can be substituted with automatic code generation, which increases the efficiency and reliability of complex system design processes.

However, the correct and efficient implementation of modeling tools is a challenging task, therefore, these tools still not really exploit their potential. In addition, performance, scalability and the lack of comprehensive benchmarks are also problems. Today, modeling tools typically lack the correct measurements. The aim of this paper is to present a method for throughout testing of these tools by generating diversive, correct and/or scalable models according to the parameterizing.

In this paper, I present how to generate diversive and scalable models with a framework for rule-based design space exploration. I will describe (i) model growing by rule-based design space exploration and (ii) abstraction-based model creation, (iii) the possible partitioning of the problem into subparts and (iv) the comparison of these problems. I developed (v) a prototype application for this problem, which uses the Viatra Design Space Exploration (DSE) framework. I use the train benchmark to demonstrate the usability of my approach.

My method supports the testing of modeling tools, benchmark creation and it also eases the selection of modeling tools by means of comparative measurements.

1. fejezet

Bevezetés

Napjainkban a biztonságkritikus rendszerek (pl. autók, repülőgépek szoftverei) elterjedésével egyre szélesebb körben alkalmaznak modellvezérelt tervezést. A modellvezérelt-tervezés során számos fejlesztési lépést automatizált kódgenerálással váltanak fel [11], hogy a növeljék a fejlesztők produktivitását és egyben javítsák a szoftverek minőségét. Ezáltal a szoftverek fejlesztése mellett a modellek fejlesztésére és modellezőeszközök használatára is komoly hangsúly kerül.

Amíg közönséges szoftverek ellenőrzésére és modellek vizsgálatára számos módszer van, maguknak a modellezőeszközöknek ellenőrzésére és teljesítménymérésére csak kezdetleges technikák léteznek. Modellezőeszközök modellekkel dolgoznak, így teszteléshez [16] és teljesítménybenchmarkokhoz [21] is helyes modellek kellenek. Azonban a meglévő módszerek vagy véletlen modelleket generálnak [6], vagy logikai következtetőket használva nem skálázódnak [17].

Dolgozat célja egy modellgenerátor elkészítése, ami képes helyes és nagy modellek előállítására, valamint jobban skálázódik mint a korábbi megoldások. Dolgozatomban bemutatok egy tervezésitér-bejárás alapú rendszert, amely növesztési transzformációk sorozatán keresztül készíti el a helyes modelleket. A transzformációs szabályok előállítására több módszert is bemutatok, amelyeket értékelek helyesség, teljesség és teljesítmény szempontjából. A modellgenerálási technikám egy esettanulmányon szemléltetem [21], a jobb skálázhatóságot pedig mérési eredményekkel támasztom alá. A módszer az előzetes mérési eredmények alapján képes 4000 objektumot tartalmazó helyes modellek előállítására is, csaknem két nagyságrenddel jobb skálázhatóságot mutat, mint ami logikai következtetőkkel elérhető volt a múltban.

Az elkészített eszközzel helyes példák generálhatóak tetszőleges modellezési nyelvhez, amelyek használhatóak tesztesetként, vagy részeit alkothatják teljesítmény benchmarkoknak.

Dolgozat felépítése A dolgozat a következőképpen folytatódik: a 2. áttekintést ad dolgozatban található legfontosabb fogalmakról. A 3. fejezetben bemutatja a tervezésitér-bejárás alapú modellnövesztést és a VIATRA-DSE keretrendszer működését. A 4. fejezetben kifejti a modellnövesztés gyorsítására alkalmazott módszereinket. Az 5. fejezetben leírom az általam végzett méréseket és ezek eredményeit. A 6. fejezetben bemutatom, hogy kik és milyen módon foglalkoztak korábban a témával és áttekintést adok arról, hogy milyen kihívásokkal szeretnék foglalkozni a jövőben.

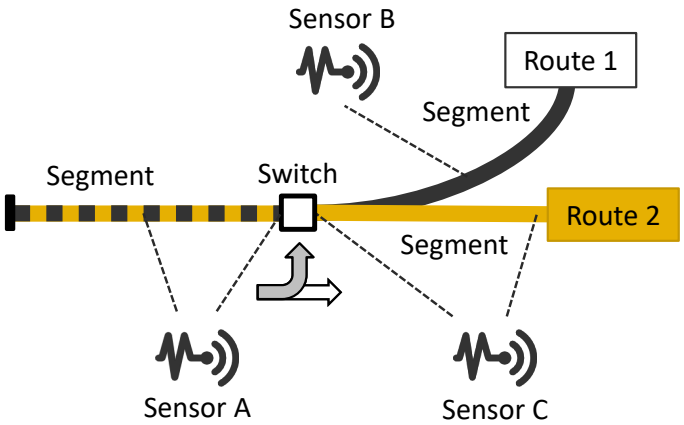
2. fejezet

Háttérismeret

2.1. Esettanulmány

A következőkben ismertetem a tanszéki kötődésű, a dolgozat során későbbiekben használt a vasúti hálózatok modellezésére alkalmas Trainbenchmark [21, 8, 22] esettanulmányt. A benncmark célja különböző modellezőeszközök teljesítményének és skálázhatóságánal összemérése. A benchmark során egy vasúti modell ellenőrzését hajtjuk végre, mely során számos helyességi kritériumot ellenőriznek az eszközök. A dolgozatot a későbbiekben helyes vasúti modellek generálásával szemléltetjük, amelyek felhasználhatóak tervezőeszközök teljesítménymérésére és tesztelésére egyaránt.

A 2.1. ábrán egy vasúti modell részlete látható. A modell szerint a vasúti hálózatot alkotó sínszakaszokat két csoportba sorolhatjuk. A *szegmensek* (Segment), két végponttal rendelkező egyenes vagy görbített szakaszok, amelyeken a vonat egyik végponttól a másikig tud eljutni. A *váltók* (Switch), a valóságos váltók modellezett megfelelői. Minden esetben három végpontjuk van, amelyek mindannyian másik switch vagy szegmens végpontjához kapcsolódnak. A switcheknek a valódi kétállapotú váltókhoz hasonlóan egy darab végpontjuk van, amelyet (amennyiben nem fordul vissza) a vonat mindenképpen érint (from), míg a másik két végpont közül (left, right) a switch-hez tartozó váltó állapota alapján csak az egyiken fog áthaladni.



2.1. ábra. Vasúti modell részlet három útvonallal.

A két négyzet egy-egy váltót jelöl. A négyzetek feletti nyilak mutatják, hogy két irányba lehet tovább haladni, a váltó aktuális állapota alapján. A szürke nyíl jelzi, hogy melyik váltó aktív egy adott pillanatban. A váltókból futnak ki a szegmensek, amelyek egyszerű sínszakaszoknak feleltethetőek meg.

A szegmensek és váltók egyaránt vasúti elemek (TrackElement), amelyek a fizikai elhelyezkedésük alapján régiókba oszthatók. Egyes vasúti elemeken találhatók szenzorok (sensor), amelyek a vonatok helyzetének meghatározását végzik, vagyis folyamatosan monitorozzák a rajtuk keresztülhaladó forgalmat és kiszámítják, hogy egy adott sínszakaszon található-e jármű. A nagyobb kockázatot jelentő szakaszokon több szenzort helyeznek el, mint a kockázatmenteseken.

A vonatok útvonalakon (Route) haladnak. Az útvonalak végén jelzőlámpák (semaphore) helyezkednek el, ami tudatja az arra haladó vonattal, hogy ráhajthat-e az útvonalra, vagy várakoznia kell (például egy másik vonat elhaladására). Az útvonalakhoz legalább két szenzor tartozik. Az útvonal két végén lévő vasúti elemeknél biztosan megtalálható egy-egy, ami segít megállapítani, hogy hány vonat tartózkodik az adott útvonalon. Minden az útvonalhoz tartozó váltóról tároljuk, hogy a vonat áthaladásához milyen pozícióban kéne állnia (switchposition), amelyet könnyen össze lehet hasonlítani a váltó aktuális állapotával.

2.2. Modellezési háttérismeretek

2.2.1. Meta- és példánymodellek

Szakterület-specifikus nyelveket tipikusan *metamodellel* és kényszerekkel határozhatunk meg. A metamodell összefoglalja a szakterület legfontosabb fogalmait, relációit és a modellek alapvető struktúráját. További kényszerekkel az érvényes modellek körét határozhatjuk meg.

A metamodell elemeiből építhetjük fel a *példánymodelleket*. A példánymodellek tekinthetőek címkézett gráfnak, ahol a típussal címkézett csomópontok a modell objektumainak, a köztük futó élek pedig az elemek közötti kapcsolatoknak feleltethetőek meg.

A továbbiakban használt metamodellek az *Eclipse Modelling Framework* (EMF) [5, 19] keretrendszerben készültek, amely a modellezőnyelvek de facto szabványa. Az eszközzel osztálydiagramszerű metamodelleket lehet készíteni, amelyből Java kód generálható. Az eszköz többek között támogatást ad modellek létrehozására egy fa struktúrájú szerkesztővel, modellek szerializására, illetve változásokra való feliratkozásnak.

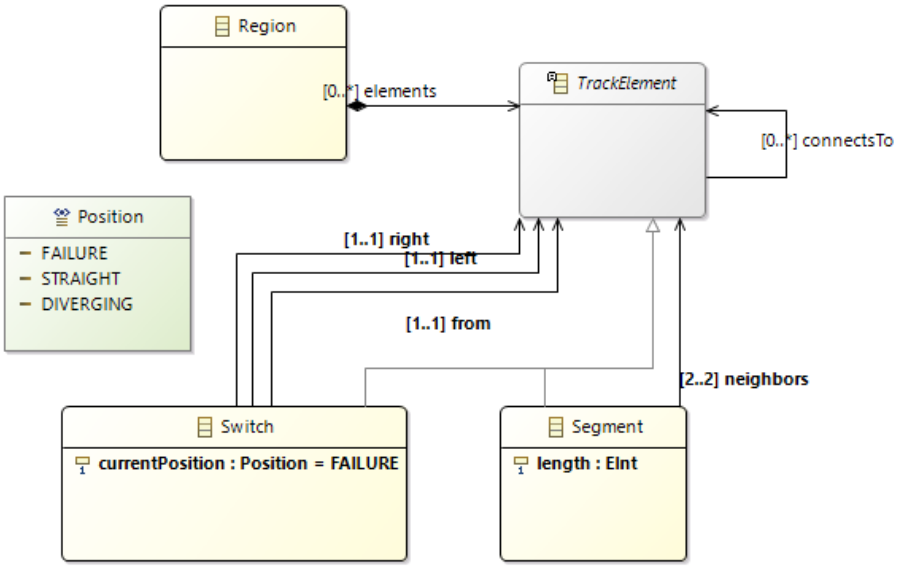
Ábra 2.2 ábra a tanszéken használt trainbenchmark metamodelljének egy részletét ábrázolja. Az EMF négy jelentősebb osztály segítségével teszi lehetővé a metamodellek definiálását.

- Az *EClassok* (osztályok), határozzák meg a a modellezési nyelv típusait. Ilyen EClassok a 2.2 ábrán a Segment, a Sensor, a TrackElement és a Region osztályok. Az EClassok példányait a modellben EObject-eknek, vagy egyszerűen objektumoknak nevezzük. EClassok között öröklési viszony definiálható (mint például Segment és TrackElement között), amelyel megadhatjuk hogy két típus elemeinek halmaza részhalmaz viszonyban van, illetve elemeket *absztrakt*nak jelölhetünk (TrackElement-et), ami megtiltja hogy egy objektum közvetlenül legyen az absztrakt osztálynak a példánya.

- Az *EAttribútumoknak* (attribútumok) van egy típusa, amit az Ecore modellben az attribútumhoz tartozó EDataType határoz meg. Például a Segmenthez tartozó length attribútum EInt típusú.

- Az *EDataType* a Javaban található változótípusok modellje (EInt, EDouble stb.).

- Az *EReferencia* (referencia) az osztályok közötti kapcsolatokat jeleníti meg. Az EReferenciáknak szintén van multiplicitásuk, amely egy alsó és egy felső korláttal adható



2.2. ábra. A Trainbenchmark metamodelljének egy részlete

meg. Például a Segment és Trackelement osztályokat összekötő neighbours kapcsolat definálja, hogy egy szegmensnek, minden esetben pontosan két szomszédja van, mivel mind az alsó, mind a felső multiplicitás kettő. Ezen kívül némely referencia tartalmazásként van definiálva (például elements). EMF-ben elvárt, hogy minden model tartalmazási élek mentén fát alkosson.

- Az *EEnum* osztályok előre definiálják a változó által felvehető névkonstansok halmazát. A Switch osztályban látható switchPosition attribútum Position típusú, ami egy általunk definiált enum, három felvehető értékkel.

2.2.2. Gráfminta

Gráfmintákkal modellelemek közötti strukturális feltételeket határozhatunk meg. Funkciójuk egyszerűbben érthető, ha a kapott példánymodellekre, mint a gráfokra gondolunk, ahol az egyes EObjectek feleltethetőek meg a csúcsoknak és a köztük lévő kapcsolatok a gráf éleinek.

Az alábbiakban a VIATRA Query szintaktikáját követve mutatom be a gráfmintákat.

Egy gráfminta meghatároz szimbolikus paramétereket, és a paraméterekre szabályokat definiál. Ha matematikailag szeretnénk megfogalmazni, akkor egy $(p1,\ldots,pn)$ paraméterlistával rendelkező q minta $body_1,\ldots,body_m$ törzsek konjukciójával van meghatározva:

```
pattern q(p1,...,pn) = body1 or ... or bodym
```

Egy $body_i$ mintatörzs bevezethet újabb $v_1,\ldots,v_n$ változókat, illetve strukturális kényszerek konjunkcióját határozhat meg a változókra és a paraméterekre:

```
bodyi = {Constraint1(v1,...vj, p1,...,pn); ... Constraintk(v1,...vj, p1,...,pn);}
```

Szemantikailag egy gráfmintát akkor teljesítenek objektumok, ha létezik olyan törzse a mintának, amelyben az újonnan bevezetett $v_1,\ldots,v_n$ változók megfelelő értéke mellett minden kényszer teljesül. A legfontosabb kényszer típusok:

- Típuskényszer
- Útvonal (összekötöttségi) kényszer
- Csúcsok egyenlősége / különbözősége
- Minta hivatkozás
- Negatív minta hivatkozás
- Tranzitív lezárt

A gráfmintáknak felhasználásának több módja ismert. A következőkben három általam alkalmazott felhasználási módot fogok bemutatni, úgy mint 1) a jólformáltsági feltételek definiálása, 2) a származtatott értékek megadása és 3) a gráftranszformációs szabályok végrehajtási pontjainak meghatározása.

A jólformáltsági feltételek segítségével, a metamodellben meghatározott kényszereknél összetettebb szabályokat fogalmazhatunk meg. Ha egy modell mind a metamodellből származó, mind pedig a jólformáltsági kényszerekkel definiált feltételeket betartja, akkor jólformált modellről beszélünk.

A jólformáltsági feltételek párja a nem jólformáltsági kényszerek, amik olyan mintákat fogalmaznak meg, amelyek ha előfordulnak a modellben, akkor az nem lehet jól formált.

Az általunk használt Train Benchmark részmodell esetén a modell jól formált, ha:

- Az összes szegmens és váltó hozzá van rendelve egy régióhoz.
- Az összes szegmensnek kettő, és az összes váltónak három szomszédja van.
- A váltók három szomszédja közül egy from, egy right és egy left kapcsolaban van.
- Se a váltóknak, se a szegmenseknek nem lehet kétszer ugyanaz az elem a szomszédjuk.
- Se váltó, se szegmens nem lehet önmaga szomszédja.

Látható, hogy az első három feltétel, aránylag könnyen megadható a metamodell segítségével, azonban az utolsó kettő feltétel megadása nem triviális a modell tervezés keretein belül. Amennyiben gráfmintákkal szeretnénk megadni őket, akkor néhány sorban képesek lehetünk megfelelő feltételek megfogalmazására. Például a váltónak két szomszédja azonos:

```
pattern SwitchEqualNeighbours(switchElement: Switch){
  Switch.left(switchElement,target);
  Switch.right(switchElement,target);
}
```

A gráfminták alkalmasak származtatott értékek definiálására. Ezek az értékek lehetnek referenciák vagy attribútumok, melyeknek az értéke származtatható a modellből. Például emberek közötti barátok kapcsolatból származtható egy barátnők kapcsolat, amely a barátok kapcsolatot szűri nem szerint. Hasonlóan, egy ember életkorát származtatni lehet a születési éve alapján.

A gráfminták harmadik felhasználási módja, hogy a segítségükkel illeszkedéseket keresünk a példánymodellen. Illeszkedésnek (match) nevezzük azokat a modellelemeket, amelyek a minta törzsének egyikében található összes kényszernek megfelelnek. Ha gráfként tekintünk a példánymodellre, akkor úgy fogalmazhatjuk meg, hogy azon részgráfok, amelyek felépítése megegyezik a gráfmintában definiált gráf felépítésével. Egy modellben egy minta több illeszkedő részgráfot is megadhat. A definiált illeszkedésekből gráftranszformációs szabályokat alkothatunk.

2.2.3. Gráftranszformációs szabályok

A gráftranszformációs szabályok lehetséges módosításokat definiálnak egy modellen (gráfon). A gráftranszformációs szabályok két részből állnak: egy előfeltételből (left-hand-side, LHS), amelyet egy gráfminta segítségével adhatunk meg és egy utófeltételből (right-hand-side, RHS), amely módosításokat definiálja [15].

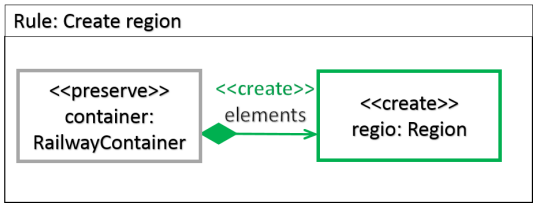
Az előfeltétel meghatározza, hogy a gráf mely részgráfjain (match) lehetséges a végrehajtás. Egy részgráfot akkor találunk erre alkalmasnak, ha a felépítése megfeleltethető az előfeltételként megadott gráfminta felépítésének, vagyis azonos számú és típusú elemük van, amelyek kapcsolatai és attribútumai is megegyeznek. Gráftranszformációs szabály esetében az illeszkedést aktivációnak is hívjuk.

Az utófeltétel megmutatja, hogy mi lesz a transzformáció eredménye, milyen módosításokat végzünk el a részgráfon a transzformáció során. Ahogy egy adott gráfon belül több illeszkedés is előfordulhat, úgy az egyes illeszkedéseken is végrehajthatunk több fajta transzformációs műveletet, amelyekkel különböző utófeltételekhez juthatunk.

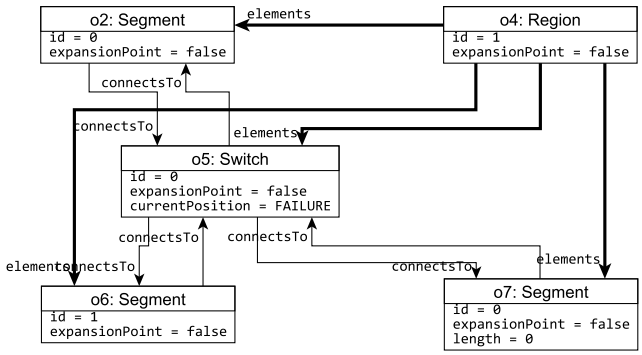
A fenti vasútmodelles példa esetében egy jól formált modell létrehozásához például tudnunk kell régiókat létrehozni. A 2.3 ábra bemutat egy ennek megfelelő gráftranszformációs szabályt (create Region) a Henshin [7] nevű keretrendszer szintaxisával. Ez a transzformáció létrehoz egy régiót, amely tartalmazhatja a hozzá tartozó vasúti szakaszokat. A create Region szabálynak mindig pontosan egy illeszkedése létezik, mivel a gyökér elem csak egy példányban van jelen és más paraméter nem befolyásolja az illeszkedést.

2.3. Szabályalapú tervezésitér-bejárás

A szabályalapú tervezésitér-bejárás (Rule-based Design Space Exploration, DSE) célja különböző rendszertervek automatikus generálása, akár tervezési, akár futási időben. A modell-vezérelt, szabályalapú DSE egy gráfjellegű kezdeti modellből indul ki, amelyen gráftranszformációs szabályokat alkalmazva jut el a kényszereket kielégítő célállapotokba. A strukturális kényszereket gráfmintákkal lehet definiálni, amely megkötéseket adhat rendszerelemek adott tulajdonságaira, vagy azok összekötöttségére. Ellentétben a numerikus számítási módszerekkel, a modell-vezérelt, szabályalapú megközelítés nem csak a kényszereket kielégítő célmodellt adja megoldásul, hanem az elérési útvonalat (szabály



2.3. ábra. A régiók létrehozására szolgáló transzformáció. A régió létrehozásához csak a container elem meglétét kell vizsgálni



2.4. ábra. Az ábrán egy a 2.2 metamodellhez tartozó példánymodell látható. Öt látható elemmel, és egy az ábrán nem látszódó tároló osztállyal, amely a régiókat tartalmazza.

végrehajtások egy szekvenciáját, vagy trajektóriát) is rendelkezésünkre bocsájt. A DSE probléma három fontos bementi paramétere 1) a kezdeti modell 2) a megadott gráftranszformációs szabályok és 3) a gráfminták segítségével meghatározott célkényszerek.

Ha az általunk használt modellel szeretnénk párhuzamba állítani a bemeneti paramétereket, akkor a következő megfeleltetéseket használhatjuk:

- Kezdeti modell: általában a legegyszerűbb kezdeti modellben csak egy darab gyökér elem van, amely mint belépési pont szolgál a DSE számára, ezért a modellábrázolásokon nem jelenik meg. Azonban az ideális kezdeti modellnek figyelembe kell vennie a megadott gráftranszformációkat és a végrehajtás során folyamatosan érvényben lévő kényszereket. Például, ha szeretnénk váltó nélküli példánymodelleket generálni, akkor szükséges a kezdeti modellben egy szegmens elhelyezése, amelyből végrehajtható a create Segment transzformáció.

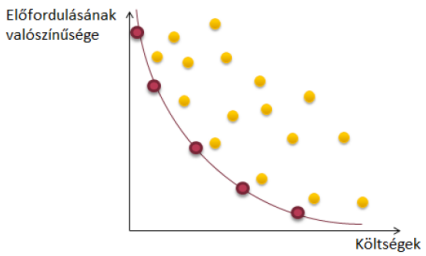
- Transzformációk: Többek között megadjuk a 2.3-t transzformációs szabálynak, ami régiókat fog létrehozni a modellben.

- Célkényszerek: megadjuk, hogy mekkora modellt szeretnénk elérni, hogy milyen legyen a modellben az elemek kapcsolata. Itt adhatjuk meg, hogy minden váltónak három és minden szegmensnek két szomszédja legyen pontosan. Ezeket a korábban definiált gráfmintákkal tehetjük meg.

2.3.1. Többcélú tervezésitér-bejárás

Mindamellett, hogy a rendszerterveknek több strukturális és numerikus kényszernek kell megfelelnie, különböző rendszerjellemzőkre (pl. válaszidő és költség) optimálisnak vagy közel optimálisnak kell lenniük. A többcélú tervezésitér-bejáró algoritmus (Multi-Objective DSE, MODSE) [2] többcélú optimalizálási problémákat definiál. A többcélú optimalizálás során különböző szempontok párhuzamos optimalizációjára törekszünk. A fenti vasúti példa alkalmával, például optimalizálhatunk arra, hogy a ma létező vasúti hálózatok által teljesített feltételeknek minél jobban megfeleljenek, amennyiben ezeket le tudjuk írni matematikai módon. Például meghatározhatjuk a váltók és szegmensek arányát, vagy a jelenlévő körök méretének gyakoriságát.

Mivel több cél esetén két megoldás közül nem tudjuk egyértelműen eldönteni, hogy melyik megoldást célszerű választani, ezért az úgynevezett Pareto frontokkal határozzuk meg a legjobb megoldások halmazát. A Pareto front a megoldások egy olyan halmaza,



2.5. ábra

amely minden más nem a Pareto frontba tartozó megoldást dominál (minden célfüggvény szerint jobb), de a Pareto frontba tartozó megoldások nem dominálják egymást. Egy megoldás dominál egy másikat, ha egyik célfüggvény értéke sem rosszabb, mint a dominált megoldásé, és van legalább egy szempont, amely szerint jobb. Ha a célfüggvény értékeket minimalizálni akarjuk, akkor a Pareto frontba tartozó elemeket a következőképpen írhatjuk le:

Definíció Ha X_1 és X_2 megoldásai egy többcélú optimalizációs problémának egyenletnek, ahol az optimális megoldás során minimalizálásra törekszünk, akkor X_1 dominálja X_2-t akkor és csak akkor, ha:

1. $\exists O_i(X1) < O_i(X_2)$, ahol O_i, az i. célfüggvény és
2. $\nexists O_i(X_1) > O_i(X_2)$

Mindezt a Ábra 2.5 szemlélteti, ahol két változó minimalizációjára törekszünk. A megoldásokat koordinátarendszerben ábrázolhatjuk. A tengelyeken helyezkednek el a célfüggvények, amik függvényében ábrázolhatjuk a megoldásokat. A legjobb megoldások által kifeszített pareto frontot az ábrán piros görbével jelöltük. Amennyiben a célfüggvények minimalizálására törekszünk, akkor a többi megoldástól lefelé és balra helyezkedik el az általunk kifeszített optimális megoldásokat tartalmazó pareto front.

2.3.2. VIATRA-DSE

A VIATRA-DSE keretrendszer [23] egy többcélú, szabályalapú tervezésitér-bejáró keretrendszer, amely az EMF és VIATRA Query technológiákra épül. A transzfromációk jobb oldalát Java kóddal lehet megadni. Több beépített bejárási stratégiával rendelkezik, úgy mint mélységi és szélességi bejárás, hegymászó keresés és genetikus algoritmusok (NSGA-II, PESA), de új algoritmusokkal is ki lehet egészíteni. További előnye, hogy képes a többszálú futásra is, ezzel is növelve a hatékonyságot.

3. fejezet

Modellgenerálás áttekintése

A fejezetben bemutatjuk a modellgenerálási módszer alapvető felépítését,

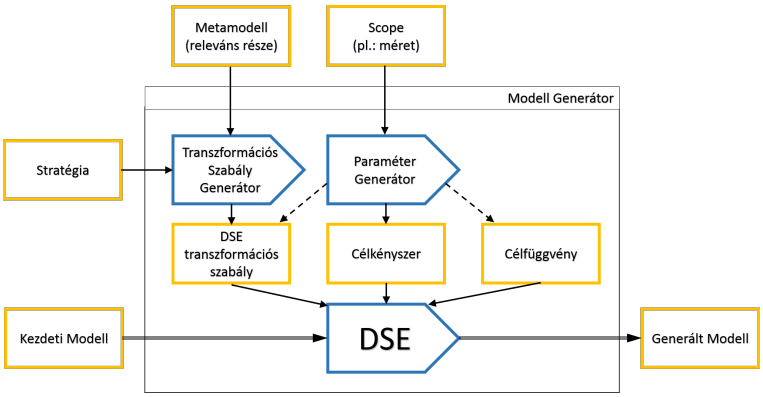
3.1. Modellgenerálás elvárt tulajdonságai

Modellgenerálás során célunk, hogy egy bemenetként megadott metamodellhez előre meghatározott méretű példánymodelleket generáljunk. A példánymodelekre alapvető elvárás, hogy a metamodellben definiált típusokat használja, illetve az alapvető strukturális kényszereket (multiplicitás, tartalmazási hierarchia) betartsa. Ha a generált modelleket tesztelésre vagy teljesítménymérésekre szeretnénk felhasználni, akkor a generált modelleknél négy szempontot kell vizsgálni:

- **Szabályosság:** a generálás során alapvető cél, hogy a metamodellből származó strukturális kényszereket betartsák a generált modellek. Egy komplex szakterület-specifikus nyelv esetén a metamodellt számtalan jólformáltsági kényszer egészít ki. Amennyiben helyes modelleket szeretnénk generálni, a kimenetnek ezeket is teljesíteni kell. Amennyiben a kimeneti modellek a kapcsolódó jólformáltsági kényszereket is teljesíti, azt mondjuk hogy a modellgenerátor **szabályos** modelleket generál. A szabályossághoz két további tulajdonság is társul:
  - **Helyesség:** a generátor garantálja, hogy csak helyes modelleket generál.
  - **Teljesség:** ha egy modell helyes, és a mérete megfelel a modellgenerálási feladatnak, akkor a generátor előbb-utóbb elő is állítja
- **Skálázhatóság:** Több modellgeneráló eszköz létezik, amely képes kisebb modellek generálására, amelyek egy vagy több kritériumot teljesítenek a fentiek közül. Azonban ahhoz, hogy a modellgenerálást érdemes legyen alkalmazni, fontos, hogy a modellek elvárt méretének növekedésével elfogadható arányban nőjön a futásidő.
- **Kezdeti struktúra:** Modellgenerálást opcionálisan kezdeményezhetjük egy kiindulási modellből, ilyenkor a generált modellek részmodellként (részgráfként) kell hogy tartalmazzák a kiindulási modellt.
- **Diverzitás:** a modellgenerátor törekedjen arra, hogy a generált modellek között nagy különbségek legyenek.

3.2. Modellnövesztés szabály-alapú tervezésitér-bejárással

A modell generálásra használt keretrendszerek változatos módszereket használnak, ezek közül külön kiemelhetjük a széles körben elterjedt logikai megoldókat [10, 13], amelyek



3.1. ábra. Tervezésitér-bejárás alapú modellgenerálás lépései és bemenetei

képesek helyes modellek generálására, azonban gyengén skálázódnak (hozzávetőlegesen 20-60 objektum), illetve a szabály alapú véletlen modellgenerátorokat [6, 4], amelyek viszont alkalmatlanok a helyes modellek előállítására.

A szabályalapú tervezési-tér bejáró algoritmusok, mint például a VIATRA-DSE jó alternatívát kínál a bonyolult kényszerekkel megadott modellek generálására, mivel a helyesen megírt viatra mintákként megadott bonyolult kényszerek ellenőrzése jól skálázódik, és képes szabályok alapján modellek előállítására. Ezen felül a VIATRA-DSE - megfelelő bemenetek mellett - képes garantálni, hogy a kimeneti modell minden bementi kényszernek megfelel, vagyis a modellek helyesek. Továbbá a bejárási stratégia függvényében képes lehet egymástól jelentősen eltérő modellek generálására. Megfelelően választott bejárási stratégiával és gráfmintákkal módszer képes lehet az összes helyes modell generálására is.

A 3.1. ábra összefoglalja a tervezésitér-bejárás alapú modellgenerálás alapvető struktúráját. A módszernek a következő bemenetei vannak:

- **Metamodel:** Meghatározza a generálandó modell típusát. A kimenetek szabványos jólformált modelljei lesznek a metamodellnek, amelyek akár a natív szerkesztőben is megjeleníthetőek. Egyes feladatoknál szűkíthetjük a generálást a metamodellnek egy releváns részére, ilyenkor a generált modell csak a kijelölt részből készít modellelemeket.
- **Scope:** Meghatározza a generátor álltal előállított modellek elvárt méretét. A dolgozatban a modellek méretét objektumok minimális számában mérjük.
- **Stratégia:** meghatározza a modellgenerálás stratégiáját. A stratégiák változtatásával akár megváltoztathatjuk a generált modellek jellegét. A stratégiákat a 4. fejezet mutatja be.
- **Kezdeti modell:** A generálást indíthatjuk egy kezdeti modellből is, ilyenkor a generátor ezt egészíti ki.

A tervezésitér-bejárás alapú modellgenerálás alapötlete, hogy a metamodellből (és stratégiától függően a jólformáltsági kényszerekből) **transzformációs szabályokat** állítunk elő amelyeknek végrehajtásának sorozatával érhető el a generált modell. A transzformációs szabályokon kívül **célkényszerek előállítása** is szükséges, elfogadási feltételként szolgál a generálás során: amennyiben a feltétel teljesül egy generálás alatt álló modellre, az megjelenik a kimeneten is. Legvégül a **célfüggvény** heurisztikus értéke meghatározza, hogy egy félkész megoldás milyen közel van a célkényszer teljesüléséhez. Egy helyesen meghatározott célfüggvény segít a keresőalgoritmusnak a megtalálni a célkényszernek megfelelő modellek generálásában.

Amennyiben akadályok (memória, futásidő) lépnek fel a megvalósítás során, a metrikák vagy globális kényszerek megadásával javíthatunk a végrehajtás hatékonyságán. Illetve ha a valósághű modell definícióját meg tudjuk adni egy adott tartományra (pl. nem szeretnénk, hogy a vasútmodellben egynél több párhuzamos szakasz fusson két pont között), akkor ezeket kényszerekkel megadva kiválogathatjuk a feltételeknek leginkább megfelelő generált megoldásokat.

Ha nem csak egy modellt szeretnénk generálni, hanem egy benchmark összes modelljét, akkor a DSE lehetőséget kínál arra, hogy a kimeneti modellek számának egytől eltérő számot adjunk meg. Ezzel könnyíti a modellgenerálást és kizárja annak a lehetőségét, hogy kétszer ugyanazt a megoldást találjuk meg, hiszen a futtatás során tárolja a bejárt állapotokat.

3.3. Modellnövesztés példa

A 3.2 ábrán egy egyszerű tíz elemből - egy régió, két váltó és hét szegmens - modell generálásnak lépéseit figyelhetjük meg. A részábrák mindegyike egy-egy végrehajtási lépést ábrázol.

A modellen téglalappal jelöltem a szegmenseket, rombusszal a váltókat és lekerekített sarkú téglalappal a régiót. A narancsszínnel jelzett elemek azok, amelyekre a gráfmintát illesztve végrehajtható a kiválasztott transzformáció. Az alább generált modell a fenti szabályok alkalmazásával adódó legegyszerűbb helyes modell. Az ábrákat balról-jobbra és fentről lefelé kell olvasni.

A első lépésben az ábrán csak egy régió található, amely minden a példa során továbbiakban létrehozott elemet tartalmazni fog, ezért az ábrán mint befoglaló síkidom jelenik meg, hogy elkerüljük a tartalmazási élek behúzását. A második-negyedik lépésben az új elemek hozzáadása történik a create Segment és a create Switch szabályok segítségével. Az utolsó öt ábrán a továbbiakban hozzáadott élek jelennek meg.

A fenti modellgenerálás során a leállási feltételeket két csoportba oszthatjuk. Az egyik esetben a TrackElementek maximális számát adjuk meg, amelyet a végrehajtás során nem haladhatunk meg:

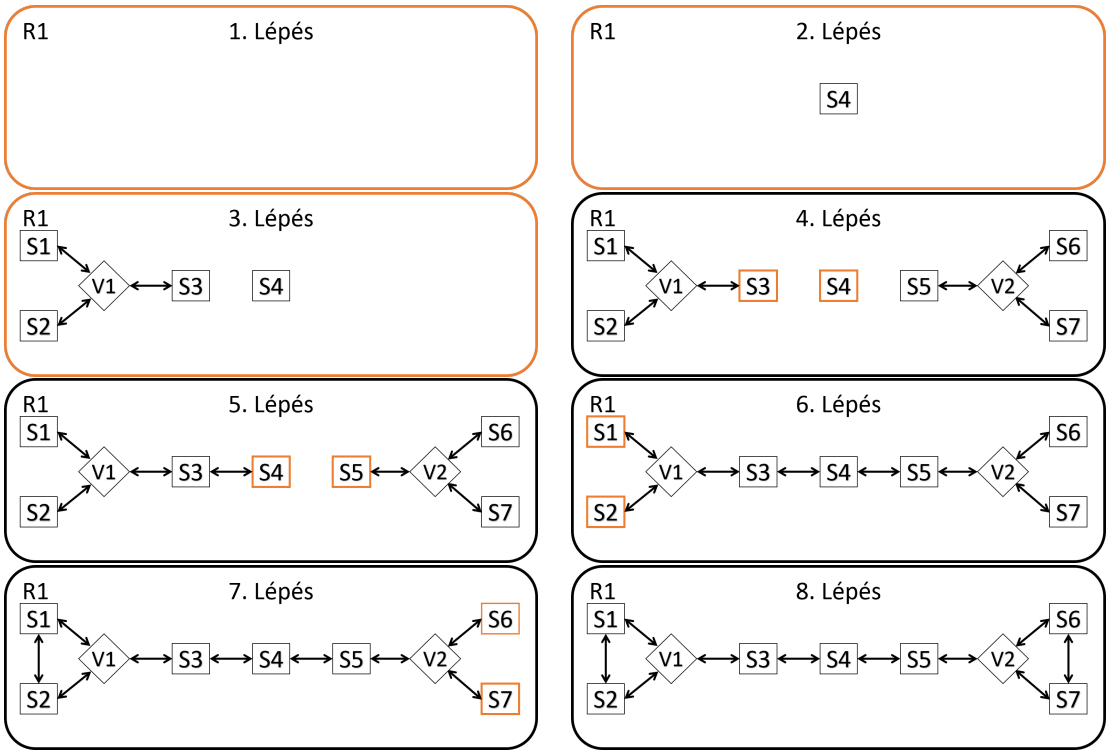
```
pattern StopAfterTrackElementNum10(){
  num == count find RailwayElements(_element);
  check(num == 10);
}
```

A másik csoportba tartozó leállási feltételek azt biztosítják, hogy a generált modell szabályos lesz. Ide tartoznak pl. a switch-ek és szegmensek szomszédszámát meghatározó követelmények.

A trackelementek maximális számát minden esetben meg kell adni, azonban egyes transzformáció szabályok szükségtelenné tehetik adott kényszerek meglétét. Például az általunk használt példában a váltók szomszédainak száma mindig három. Nem lehetséges olyan módon konstruálni a modelleket, hogy ettől eltérjünk, így az erre vonatkozó gráfminta megadását elhagyhatjuk, ha a feladat egyszerűsítésére törekszünk.

3.4. Modell dekompozíció

Érdemes megfigyelnünk, hogy vannak transzformációs szabályok, melyek sorrendje felcserélhető. A fenti példa esetében mind az elemek hozzáadásának, mind pedig az összekötésüknek a sorrendje irreleváns a generált modell szempontjából. A keresésitér-bejárás során



3.2. ábra. Az ábrán egy általános modellnövesztési példa látható, amit a korábban definiált négy növesztési szabállyal meg lehet valósítani. A kiindulási modellünk egy darab régióból áll és a modellgenerálás végén egy tíz elemből álló példánymodellt kapunk, amely két váltót, hét szegmenst és egy régiót tartalmaz.

érdemes szem előtt tartani, hogy az illeszkedések számának megnövelésével jelentősen nő az algoritmus memóriaigénye, ezért célszerű csökkenteni a redundanciát.

A memóriaigény minden további hozzáadott transzformációnál két okból jelentkezik. Az első, hogy minél több illeszkedés van egy gráfon belül, annál hosszabb listákban tároljuk az összes lehetséges illeszkedést. A listából végül csak egy elemet választunk ki (azt az illeszkedést, amelyen végrehajtjuk a hozzá tartozó transzformációs szabályt), ami a listák implementációjából következően nem eredményez optimális elérést. Az illeszkedések számának növekedése esetén arra is számíthatunk, hogy a futtató keretrendszer programfuttatás helyett a futásidő legnagyobb részében memóriaműveleteket végez.

A lassulás másik oka - az egyes bejárási stratégiáknál különböző mértékben problémát okozó - bejárási stratégiák. Pl. egy DFS stratégia esetén sokkal nagyobb lassulást okoz a keresési-tér bővülés, mint Hill-climbing stratégia használatakor. Nagyobb metamodellek vagy nagyobb számú transzformációk esetén érdemes megfontolni a feladat dekompozícióját, hogy minimalizáljuk a több illeszkedésből eredő lassító hatást. A feladat dekompozíciója során az eredetileg meghatározott problémát kisebb, egymásra épülő, kevesebb transzformációs szabállyal rendelkező részekre bontjuk, majd a részfeladatokat egymás után végrehajtva a részek után kapható és az eredetileg elérhető megoldáshalmaz fedi egymást.

Példa Ha a korábban megadott 2.2 modellen szeretnénk bemutatni a dekompozíciót, akkor két részre oszthatjuk a modellt. Ezek a részek legfeljebb egyirányú függőséget tartalmazhatnak, hiszen az elsőre generált modell lesz a második modell bemenete. Az első részben külön tudjuk választani az elemek generálására szolgáló lépéseket és a TrackElementek kapcsolatainak kialakítását célzó transzformációkat. Ebben az esetben a 3.2 ábrán látható első négy lépés az elsőnek végrehajtott keresés része lesz, míg az utolsó négy transzformáció a másodikba kerül.

4. fejezet

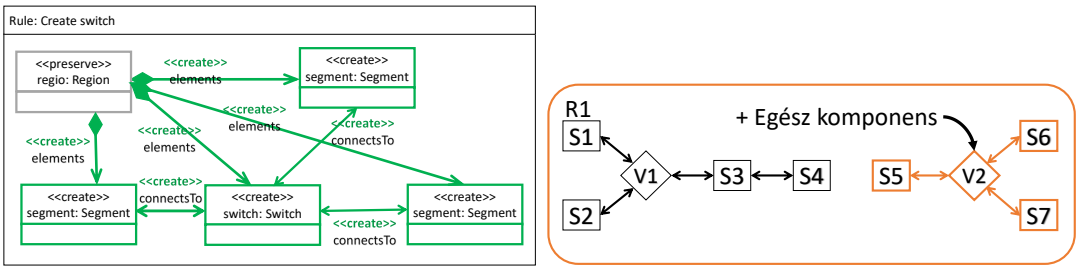
Modellnövesztési technikák

Szabályalapú modellnövesztés esetén több többféle stratégiát is alkalmazhatunk modellek felépítésére. A fejezetben bemutatok különböző modellnövesztési stratégiákat, definiálom a stratégiákhoz tartozó transzformációs szabályokat és elfogadási feltételeket, majd összefoglalom előnyeiket és hátrányaikat.

4.1. Komponens alapú növesztés

Általános leírás: A növesztési stratégia alapötlete, hogy helyes, kész komponensek beszúrásával érjük el a végső modellt. A transzformációs szabályok tehát elkészített komponenseket tartalmaznak, a baloldaluk pedig meghatározza a komponens beszúrásához szükséges objektumokat és szülőket.

Példa Vegyünk egy olyan kész négyelemű komponenst, amely egy váltóból, és három hozzá tartozó szegmensből áll. A 4.1. ábrán látható a komponenst beszúró transzformációs szabály, amely befűzi a kész modellrészletet egy meglévő régióhoz.



4.1. ábra. Példa a modellkomponens szabállyá általánosítására, és transzformáció végrehajtására.

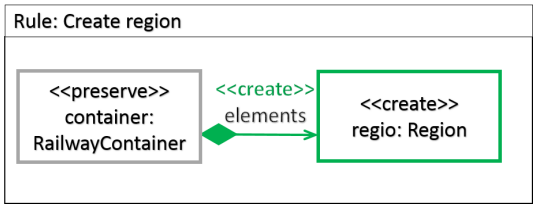
Példa

- A create Region transzformáció [4.2a ábra] létrehoz egy régiót, amely tartalmazhatja a hozzá tartozó vasúti szakaszokat. A create Region szabálynak mindig egy illeszkedése létezik, mivel a container csak egy példányban van jelen és más paraméter nem befolyásolja az illeszkedést.
- A create Segment transzfomráció [4.2b ábra] létrehoz egy új szegmenst (new), és szomszédként hozzákapcsolja egy már létező szegmenshez (old), így a nyílt végű szegmensek száma mindig állandó marad. Az 2.4 ábrán lévő példánymodell esetén három részgráfra illeszthető a minta. Ezek mind tartalmazzák az o4-es

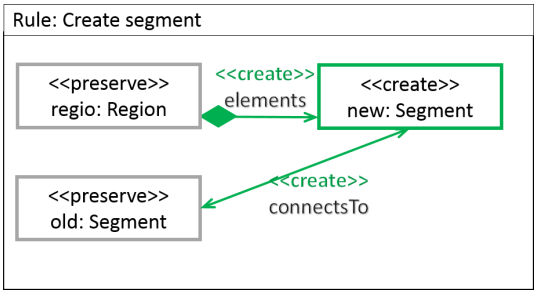
Régió elemet, illetve mindegyik részgráf tartalmaz egy-egy szegmenst, az ábrán lévő három közül.

- A connect Trackelemnts transzformáció [4.2d ábra] összeköt két egymástól független szegmenst, ha azoknak legfeljebb egy-egy szomszédjuk van. Ha az 2.4 ábrán lévő példánymodellen szeretném megvizsgálni, hogy hányszor és hol lehetséges a transzformáció elvégzése, akkor azt láthatjuk, hogy 6 illeszkedést találunk, amik rendre az o2-o5, o5-o2, o2-o7, o7-o2, o5-o7 és az o7-o5 nevű elemeket foglalják magukba. Mivel az elemek egymással felcserélhetőek az illeszkedésen belül, ezért az gráfminták az egyes elemekből álló lehetőségek összes kombinációját meg fogják adni. Ezért ha sok elemmel készítünk gráfmintákat érdemes lehet az id-k alapján sorrendezni az elemeiket.

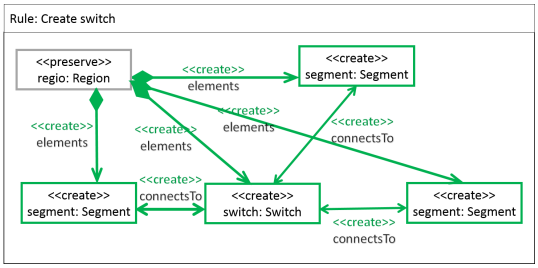
- A create Switch transzfomráció [4.2c ábra] létrehoz egy váltót, és a hozzá tartozó szegmenseket. Mivel a szegmenseket a váltóval egyszerre hozzuk létre, így a transzformációval gyorsabban növelhetjük a modellt, de számolnunk kell vele, hogy minden create Switch transzformáció során hárommal nő a lezáratlan élek száma, amely további illeszkedési pontok kialakítását teszi lehetővé. Ha túl sok ilyen pont van, akkor az eclipse memória hiány miatt nem lesz képes kezelni és leáll. A create Switch minta előfordulási száma megegyezik a régiók számával, mivel minden régió esetén pontosan egy illeszkedést találunk.



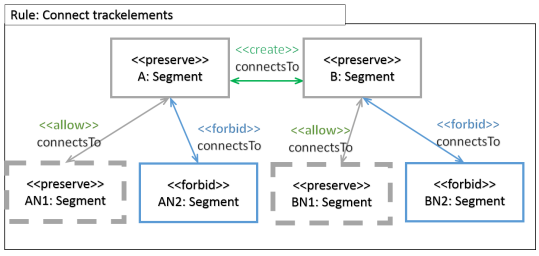
(a) Régió létrehozásai



(b) Szegmens létrehozása



(c) Váltó létrehozása három hozzá kapcsolódó szegmenssel



(d) Két Segment összekapcsolása

4.2. ábra. Az ábra a generatív modellnövesztés transzformációs szabályait mutatja be.

Előnyök: A módszer minimális interakcióval jól általánosítható kész példánymodellekből. Mivel ismert komponenseket fűz össze, a kimeneti modellek felépítése jól átlátható és nem tartalmaz nem várt struktúrákat (ezért benchmark modellek generálására alkalmas).

Hátrányok: A módszer nem általánosítható automatikusan, mindenféleképpen szükséges manuálisan kijelölni a modellrészleteket, és általánosítani növesztési szabállyá. Ezen

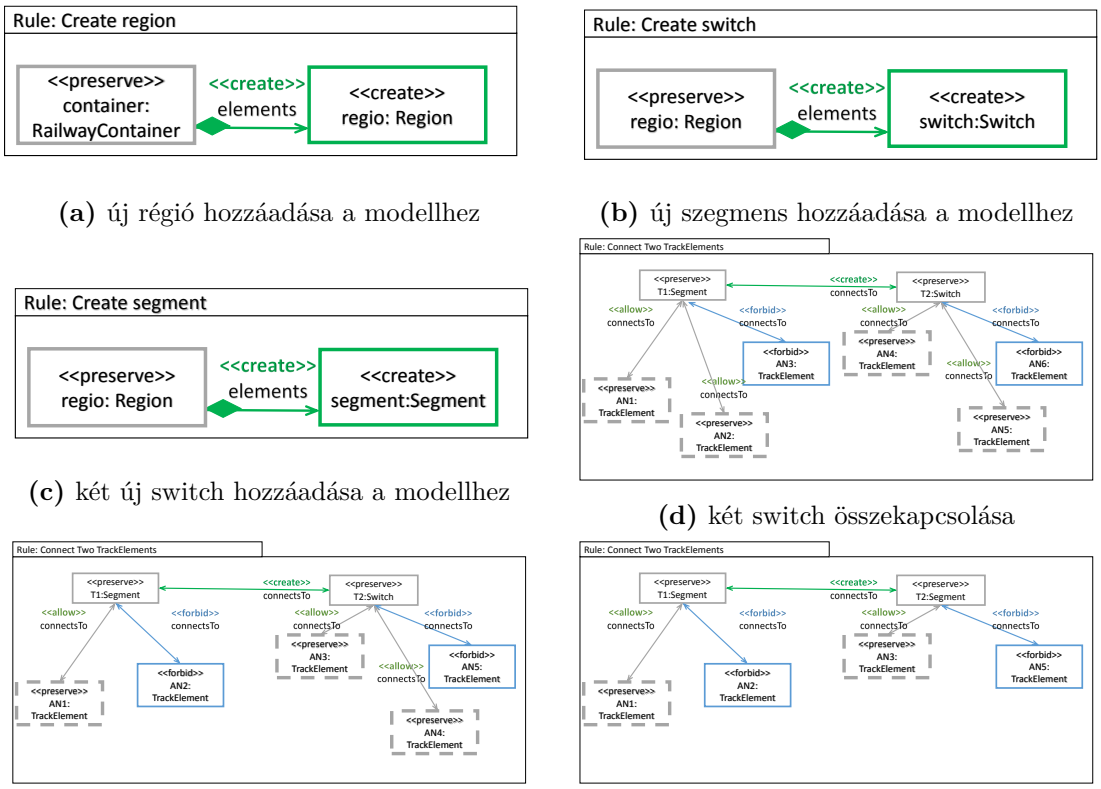
kívül, mivel ismert részgráfokat fűzünk össze, a módszer nem alkalmas tesztelésre, mivel az újonnan generált esetek nem tartalmaznak újfajta struktúrákat, így nem növelnék a tesztesetek fedettségét.

A teljesítményt rontja, hogy sok illeszkedés lesz minden transzformációra a modellen belül, hiszen kész komponenst sokféleképpen elhelyezhetünk egy félkész modellben.

Továbbá, mivel a módszer nem veszi figyelembe a jólformáltsági kényszereket, a módszer nem garantál helyes kimeneteket a végrehajtás során, illetve a teljességre nem is törekszik.

4.2. Elemenkénti modellnövesztés

Általános leírás: Minden transzformáció elemi növesztési szabályokat fogalmaz meg a modelleken, és minden egyes lépés egyetlen elem beszúrást, vagy egy új referencia létrehozását definiálja. Ennek következtében a kész modellek atomi lépések sorozatával építhetőek fel. A modellgenerálás során különválasztjuk az elemek generálását és kapcsolataik kialakítását.



(a) új régió hozzáadása a modellhez

(b) új szegmens hozzáadása a modellhez

(c) két új switch hozzáadása a modellhez

(d) két switch összekapcsolása

(e) segment és switch összekapcsolása

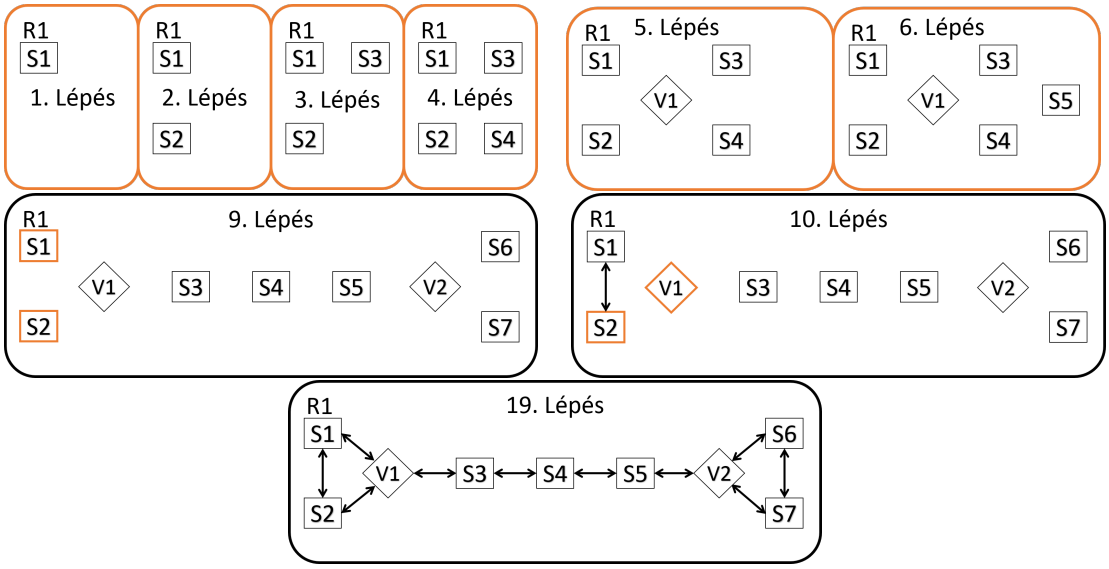
(f) két segment összekapcsolása

4.3. ábra. Az ábra a generatív modellnövesztés transzformációs szabályait mutatja be.

Előnyök: A módszer automatikusan általánosítható tetszőleges metamodellből, illetve tetszőleges kiinduló modell esetében alkalmazhatóak. A generálás és kapcsolatépítés különválasztásával a keresési-tér mérete is csökken.

Hátrányok: Aránylag sok illeszkedés lesz minden transzformációra a modellen belül, illetve sok transzformációt kell végrehajtani, hogy eljussunk ugyan abba az állapotba, ahova egy másik módszer esetén 1-2 lépésben jutnánk. Továbbá a modell nem lesz helyes

a végrehajtás során, lehetséges, hogy a modellgenerálás kezdetén olyan állapotokba jutunk, amelyekből nem tudunk sikeres modelleket generálni.

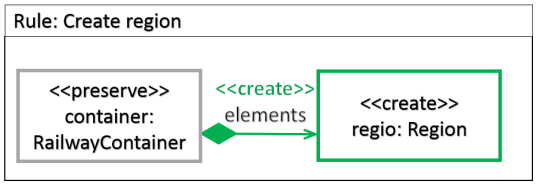


4.4. ábra. Az elemenkénti modellnövesztés az egyik legtöbb transzformációt igénylő módszer, mivel a lépései nagyon kis változtatásokat idéznek elő a modellen.

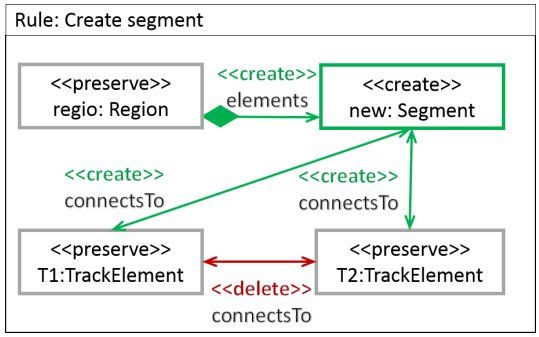
4.3. Generatív nyelvtan

Általános leírás: A keresésitér-bejárás során az egyik legnagyobb probléma, hogy ha a stratégiától nem követeljük meg minden transzformáció után a modell helyességét, akkor azt csak célkényszerként ellenőrizhetjük. Emiatt az algoritmus a futása során többször kénytelen egyes transzformációkat visszavonni, amikor felismeri, hogy az aktuális állapotból nem érhető el helyes állapot.

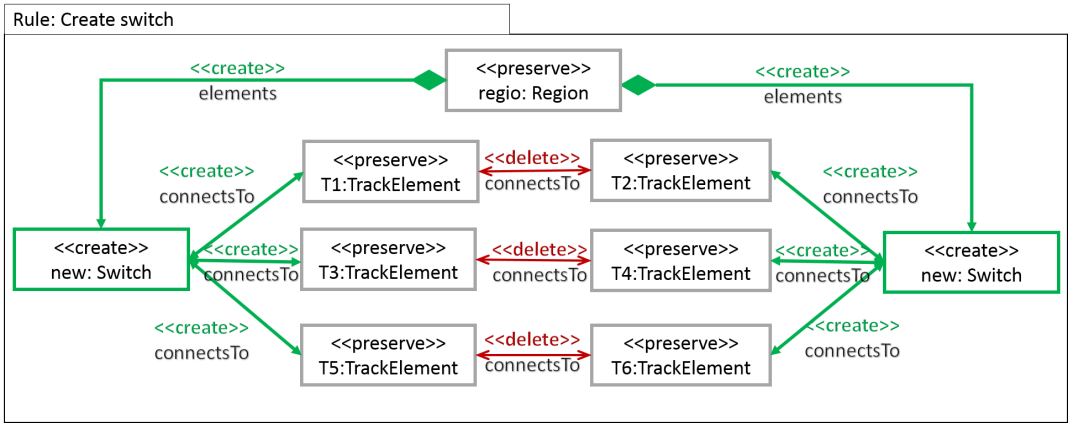
A generatív nyelvtan alapötlete, hogy úgy csökkentjük a keresésitér-bejárás során a visszalépések számát, hogy a transzformációk megfelelő megadásával elkerüljük, hogy a modell helytelen állapotba kerüljön. Tehát olyan szabályok elkészítése a cél, amelyek alkalmazásával csak helyes állapotok generálhatóak.



(a) új régió hozzáadása a modellhez



(b) új szegmens hozzáadása a modellhez



(c) két új switch hozzáadása a modellhez

4.5. ábra. Az ábra a generatív modellnövesztés transzformációs szabályait mutatja be.

Példa Vasúti hálózatok esetén az alábbi három minta generálja a helyes modelleket:

- Régió hozzáadása a modellhez: Régiókat minden feltétel nélkül adhatunk hozzá a modellhez (4.5a)
- Modell bővítése szegmenssel: Ha a modellhez szegmenst szeretnénk hozzáadni, akkor keresnünk kell szomszédos TrackElementet (szegmens vagy váltó). A két elem közötti kapcsolat helyére egy új elemet illesztünk, amely mind a két korábban szomszédos TrackElementtel kapcsolatban áll. (4.5b)
- Két váltó hozzáadása a modellhez: Amennyiben helyes modellt szeretnénk generálni, akkor észre kell vennünk, hogy nem adhatunk hozzá a modellhez páratlan számú váltót. Ezért két váltót illesztünk a modellbe a szegmens esetéhez hasonlóan, annyi különbséggel, hogy itt három pár TrackElementet választunk ki, és párok tagjai a két váltó egy-egy szomszédai lesznek. (4.5c)

Ezt beláthatjuk úgy, hogy a szomszédokat kapcsolódási pontoknak tekintjük, amik közül (eltekintve a további megkötésektől) bármelyik kettő összeköthető egymással. Ebben az esetben mivel minden váltónak három szomszédos elemmel kell kapcsolatban lenni, míg minden szegmensnek kettővel, a kapcsolódási pontok száma 2 * szegmensek száma + 3 * váltók száma. Ha a váltók száma páratlan, akkor ez egy páratlan szám lesz. Viszont ha egy kapcsolatot létre szeretnénk hozni, ahhoz két kapcsolódási pont kell, amihez páros számú elem szükséges.

Tehát, ha két váltót szeretnénk hozzáadni a modellhez, akkor hat kapcsolódási pontot (szomszédos elemet) kell számukra biztosítani. Mivel a modellben csak olyan

elemek állnak, amelyeknek pontosan az elvárt számnak megfelelő szomszéduk van, ezért három kapcsolatot érdemes kiválasztani. A kapcsolatokat felbontjuk és három elem az egyik, három elemet a másik váltó szomszédja lesz. (4.5c)

Előnyök: A megközelítés előnye, hogy a transzformációs szabályok bármilyen kombinációját alkalmazva a generált modell minden lépés végrehajtása után helyes marad. Ennek következtében nem szükséges incquery kényszerekkel ellenőrizni a modell helyességét, illetve elkerülhetjük a helytelen sorrendben történő szabályalkalmazások miatti visszalépéseket.

Hátrányok: A megközelítés hátránya, hogy a transzformációs szabályok, sok bemeneti paramétert tartalmaznak. Pl. a két switch beillesztés esetén az incquery mintának hét bemeneti paramétere van, ami azt jelenti, hogy a lehetséges illeszkedések száma (ha nem számoljuk az elemek permutációjával keletkező további illeszkedéseket):

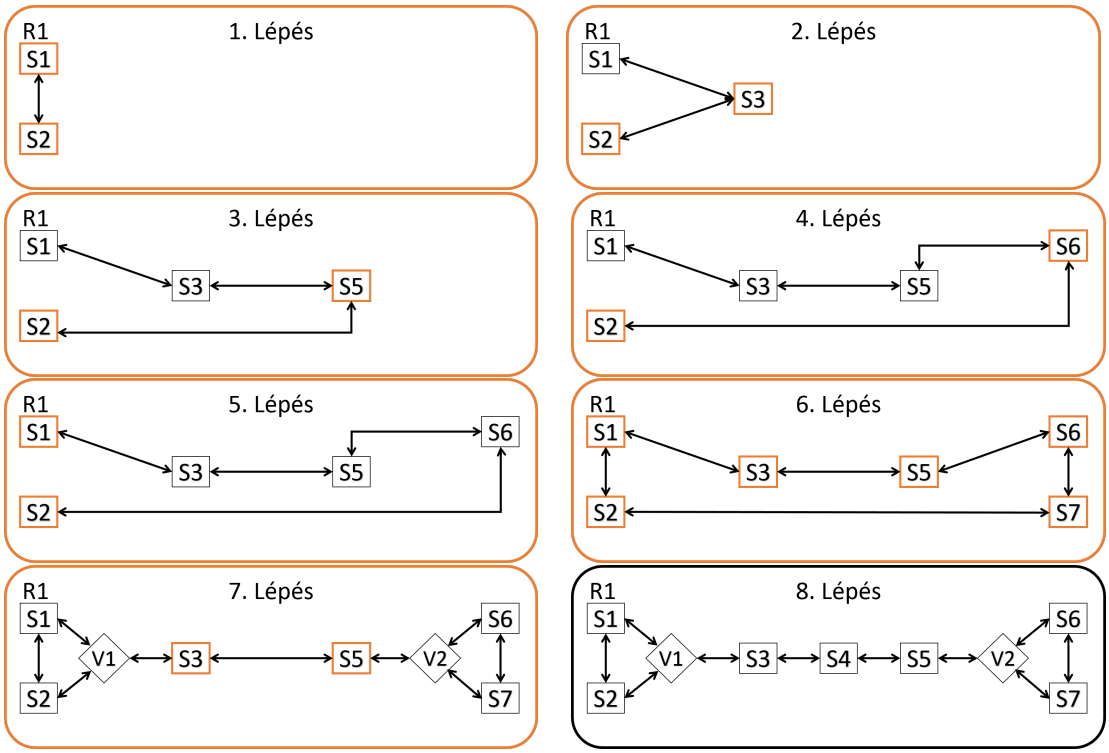
$$(\binom{|Regions|}{1} + \binom{|Regions|}{2}) * \binom{|Connections|}{3} \tag{4.1}$$

ahol a |Regions| a régiók számát jelenti, a |Connections| pedig az egymással két szomszédos TrackElementet összekötő élek számát adja meg vagyis a

$$szomszédosságiélekszáma = \frac{|váltók| * 3 + |szegmensek| * 2}{2} \tag{4.2}$$

Ha kiszámoljuk, hogy ez megközelítőleg hány illeszkedést jelent, már 60 váltó és egy régió esetén 117 480 illeszkedést jelent. Az illeszkedések nagy száma miatt a modell nem valószínű, hogy jól skálázódna.

Feltételezett méret: A modell mérete az illeszkedések nagy száma miatt nem lesz jól skálázódó, főleg nagyszámú váltó esetén.

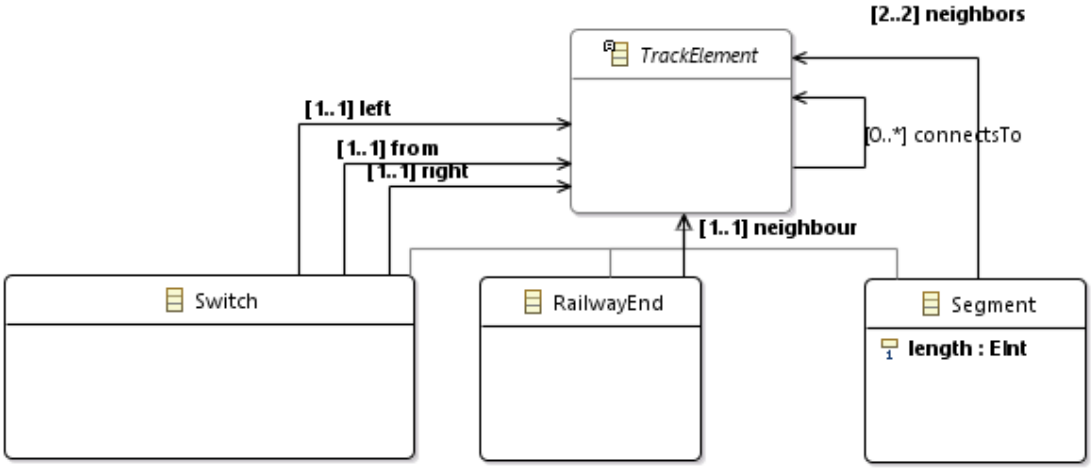


4.6. ábra. Az ábra a generatív nyelvtannal mutatja be a vasúti növesztés módszerét

4.4. Generatív nyelvtan helyettesítő objektumokkal

Általános leírás: Az előző módszer problémája a nagy mennyiségű illesztés volt, kis számú modell esetén is. A következőkben több olyan módszert mutatunk be, amelyekkel lehetséges az illeszkedések számát korlátozni.

Az első ilyen módszerben bevezetünk néhány új ecore objectet (RailwayEnd), amivel megváltoztatjuk a metamodellt (Ábra 4.7). Az ecore objectek, ha bizonyos osztályok közös ősosztályból származnak, úgy, hogy a kapcsolataik számában eltérés tapasztalható, akkor bevezetnek egy új Objectet, amelynek csak egy kapcsolata van. Így a több elemet igénylő elem hozzáadások több független transzformációs szabályra szedhetők szét az új osztály segítségével.



4.7. ábra. A korábban használt metamodellt a RailwayEnd nevű elemmel egészítem ki, amely egy szomszéddal rendelkező TrackElementként definiálható.

Az vasúti zárószakasz lehetővé teszi, hogy a modellképzés során ne kelljen két váltót beilleszteni egy transzformáció során. Így elég csak egy váltót beilleszteni a modellbe.

Ezzel a fenti képlet az alábbiak szerint módosul:

$$(\binom{|Regions|}{1} + \binom{|Regions|}{2}) * \binom{|Connections|}{2} \tag{4.3}$$

ami a fenti 60 váltó esetén "csak" 4005 lehetséges illeszkedést definiál. Azonban, a megvalósítás során további megszorításokat teszünk, nevezetesen nem lehet olyan helyre beilleszteni éleket, ahol nincs zárószakasz legalább az egyik oldalon. Ezzel tovább csökken az egyes lépések után végrehajtható transzformációk száma.

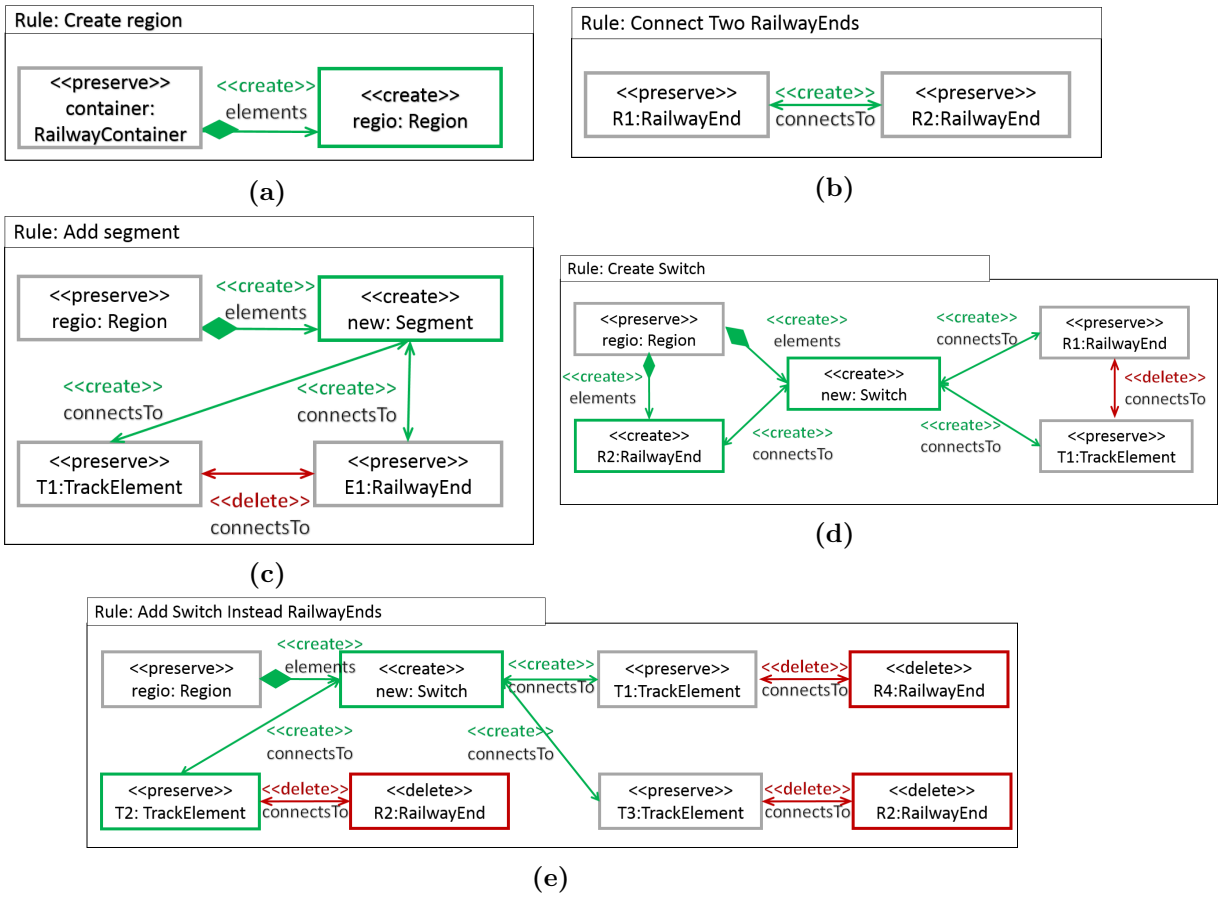
A módszer egy lehetséges változata, ha az RailwayEndek számát fix constansként adjuk meg. Ebben az esetben arra kell figyelni, hogy a fix számot úgy adjuk meg, hogy legalább egy módon le tudjuk generálni az általunk elvárt modellt.

Példa Vasút modellben használt transzformációk lehetnek:

- régió hozzáadása a modellhez: Az előző módszerhez hasonlóan itt is ugyan azt a transzformációt alkalmazhatjuk. (Ábra 4.8a)
- szegmens hozzáadása a modellhez A: A transzformáció során a két létező RailwayEndet Szegmenssé alakítjuk és összekapcsoljuk őket szomszédokká. (Ábra 4.8b)
- szegmens hozzáadása a modellhez B: Az előző módszerrel ellentétben, itt nem csökken a RailwayEndek száma. Egy TrackElement és a szomszédos RailwayEnd közé beillesztünk egy új szegmenst. (Ábra 4.8c)
- váltó hozzáadása a modellhez A: A szegmens hozzáadáshoz hasonlóan a váltókat is kétféle módon adhatjuk hozzá a modellhez. Az első lehetőség, hogy a transzformáció során egy RailwayEnd és a vele szomszédos TrackElement lesz a váltó két szomszédja, illetve létrehozunk egy új RailwayEndet. (Ábra 4.8d)

- váltó hozzáadása a modellhez B: A második lehetőség, hogy három létező RailwayEnd megszüntetésével a pontok szomszédait nevezzük a váltó három szomszédjának. (Ábra 4.8e)

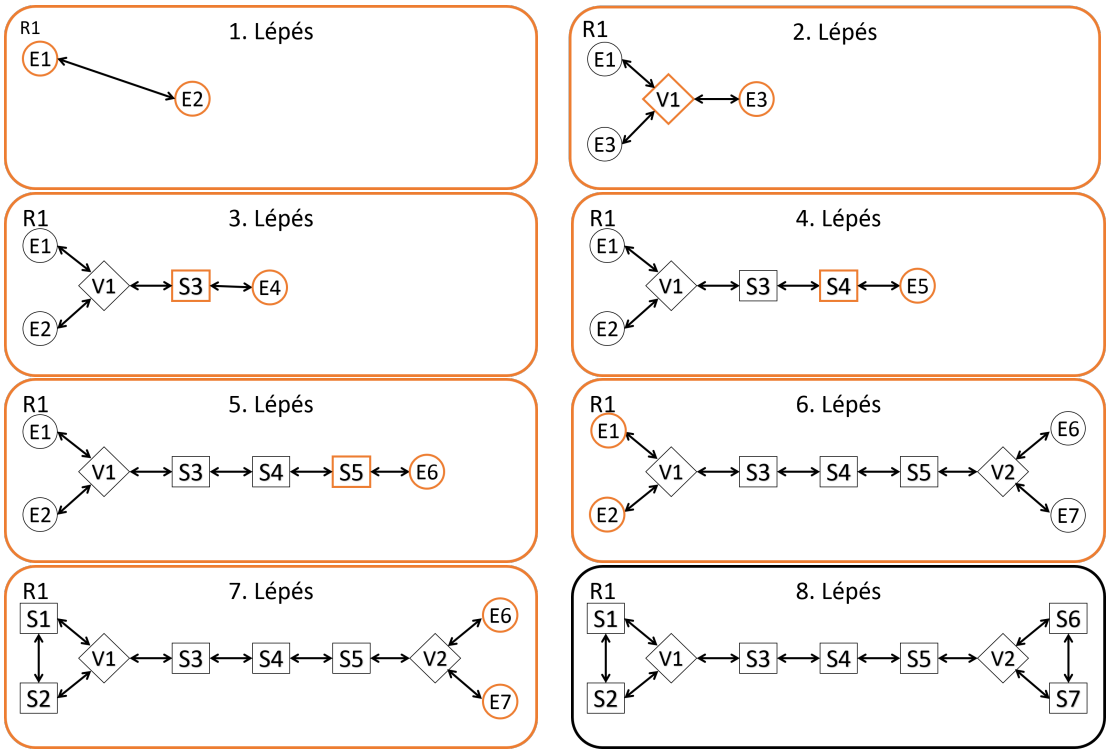
Előnyök: Az előző módszerhez hasonlóan a transzformációk itt is úgy vannak megírva, hogy helyes modellt kapjunk. Viszont minden egyes transzformáció végrehajtásához szükség van RailwayEnd-ekre, amelyek számát globális kényszerekkel alacsonyan lehet tartani (pl. arányszámokat állíthatunk be), így karbantartható a modellben található illeszkedések száma.



4.8. ábra. A railwayEnd új osztállyal történő növesztés transzformációs szabályai

Hátrányok: Be kell vezetni egy új elemet a modellbe. Míg az előző módszerek sem voltak automatikusan általánosíthatók, az új elemek bevezetése még inkább megnehezíti, hogy valamilyen módon az INCQUERY minták automatikus generálására törekedhessünk. Továbbá megfelelő globális kényszereket kell írni, amelyek nem gátolják nagyobb modellek esetén a gráf növekedését.

Feltételezett méret:A modell feltételezett legnagyobb mérete többszöröse lesz az előző megoldásban elértnél, mivel az illeszkedések száma alacsonyabb.



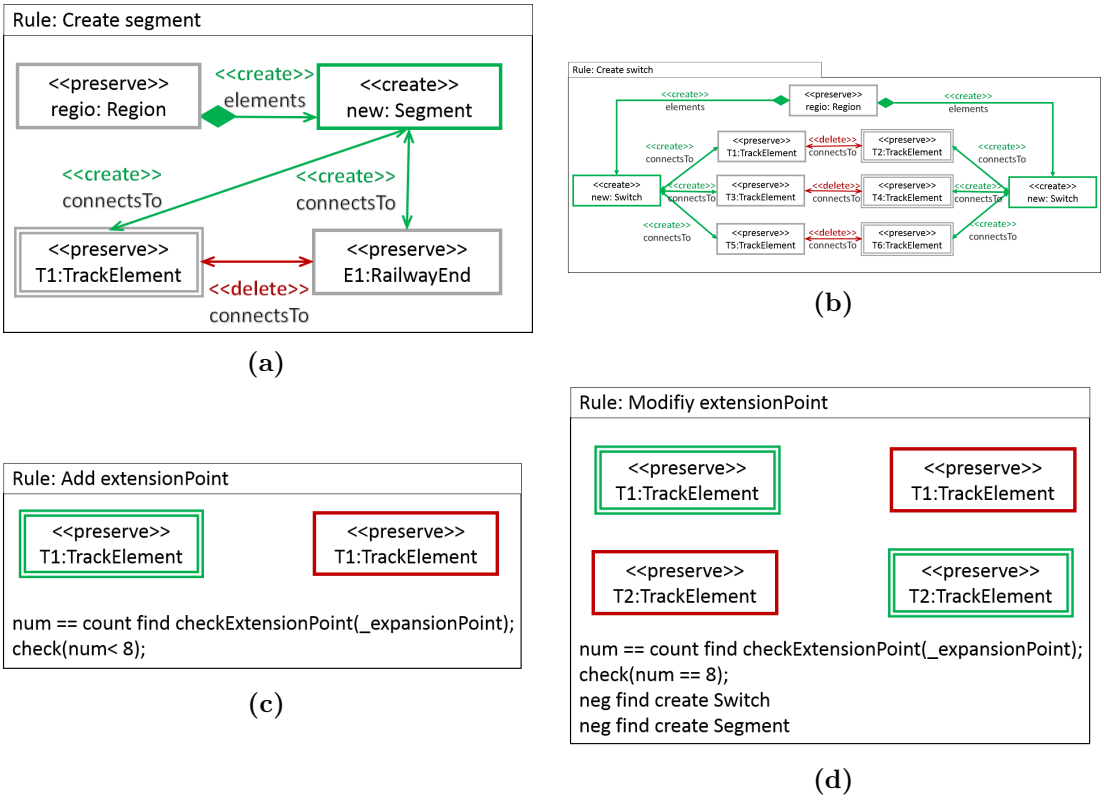
4.9. ábra. A railwayEnd új osztállyal történő növesztés transzformációs szabályai

4.5. Szélső pontokon keresztül történő modellnövesztés

Általános leírás: Az előző módszer hátránya, hogy további metamodell elemeket kell bevezetni a modellezés során. A szélsőpontokon történő modellnövesztés során nem vezetünk be további elemeket, hanem helyette egy változót adunk az elemekhez, amely mutatja, hogy valamelyik elem növesztési pont-e vagy sem. Minden transzformáció legalább egy növesztési pontot tartalmaz.

A növesztési pontok számának maximumát a futtatás elején beállítjuk. Ameddig nem érjük el ezt a maximális számot, addig minden újonnan hozzáadott elem növesztési pont lesz a modell számára. Ha nincs elég növesztési pont és még nem értük el a meghatározott maximumot, akkor adhatunk hozzá új pontokat. Ha elértük a meghatározott maximumot, akkor egy növesztési pont helyett egy másikat választhatunk.

Ha hatékony implementációt szeretnénk készíteni, akkor úgy kell kialakítani a transzformációkat, hogy bárhogy választjuk ki a lehetőségek közül az növesztési pontokat, mindig képesek legyünk egy helyes megoldás felé történő növesztésre. Vagy fordítva a növesztési pontok maximális számát úgy érdemes meghatározni, hogy legalább annyi legyen, mint amennyi a legtöbb pontot igénylő transzformációhoz szükséges.



4.10. ábra. A növesztési pontok kijelölésével történő növesztés transzformációs szabályai. A kétszeres éllel jelölt elemek az aktuálisan kiválasztott növesztési pontok.

Példa Vasút modellben használt transzformációk lehetnek:

- Régió hozzáadása: a korábbi módszerekhez hasonlóan itt is lehetőség van régiók hozzáadására
- Szegmens hozzáadása: A szegmensek hozzáadása két Trackelement között lehetséges, ahol legalább egy extensionpointként funkcionál (Ábra 4.10a).
- Váltó hozzáadása: a váltók hozzáadása esetén olyan Trackelementek közötti kapcsolatokat keresünk, amiknél legalább egy elem extensionpoint. (Ábra 4.10b)
- Növesztési pont hozzáadása: Amennyiben a növesztési pontok száma nem éri el az előre meghatározott maximális számú növesztési pontok számát, akkor (Ábra 4.10c)
- Ha nem nevezhetünk ki újabb növesztési pontot, mivel elértük a maximális számot, akkor változtatjuk, hogy melyik elem lesz a növesztési pont. (Ábra 4.10d)

Mivel a példa modell összesen 10 elemből áll, ezért a mintapélda generálásának lépései megegyeznének a generatív nyelvtan ábráján látható lépésekkel A 4.6

Előnyök: Az előző megoldásnál, ha biztosak szerettünk volna lenni benne, hogy nem okoz problémát az elemek száma, akkor különböző előre meghatározott arányokkal kellett kordában tartani az RailwayEnd típusú elemek arányát. Ebben az esetben könnyebb logikusan végiggondolni, hogy pontosan mennyi növesztési pontra van szükség egy-egy elem beillesztéséhez.

Hátrányok: Minden esetben már a transzformációk alatt sok elemzést igényel, hogy jól legyenek megírva a transzformációs szabályok.

4.6. Modellgenerálási stratégiák összefoglalása

	Szabályos	Teljes	Skálázódó	Modell javítás
Komponens alapú növesztés	I	N	N	N
Generatív modellnövesztés helyettesítő objektumokkal	I	I	I	N
Generatív modellnövesztés	I	N	N	N
Elemenkénti modellnövesztés (Helyes modell)	I	I	N	I
Elemenkénti modellnövesztés (Nem helyes modell)	N	I	I	N
Szélső pontokon keresztüli modellnövesztés	I	I	I	N

4.11. ábra. A fenti összefoglalja a különböző generálási stratégiák előnyei és hiányosságait

A táblázat soraiban a modellgenerálási módszerek találhatóak. Az oszlopoknál a helyesség azt jelenti, hogy a generált modell minden bemeneti kényszert kielégít. A teljes, hogy minden a metamodellnek és a kényszereknek megfelelő modell generálására képes. A modell javításra képes módszerek alkalmasak nem helyes bemeneti modellek modellnövesztéssel történő helyessé tételére.

5. fejezet

Értékelés

A különböző modellgenerálási módszerek összeméréséhez egy kezdeti mérési környezetet állítottam össze. A mérések célja az alábbi kérdések megválaszolása volt:

K1 Mekkora az alábbi módszerekkel generálható legnagyobb modell?

K2 Melyik a legnagyobb generált helyes modell?

K3 Az általunk bemutatott teljes vagy a helyes modellt generáló módszerek tudnak nagyobb modellt generálni?

A méréseket a dolgozatban bemutatott trainbenchmark példán végeztem. A méréseket végző számítógépben Intel(R) Core(TM) i5-6200 CPU @ 2.30GHz processzor van. A méréseket a nyílt forráskódú eclipse szoftverkeretrendszerben végeztem. Az eclipse indításakor (64 bites gép és eclipse) 4GB memóriát használtam.

Az táblázat fejlécében a mért modellek minimális mérete található. Azért választottam a minimálisan mérhető méretet, mert egyes módszerek esetén pontosan a kívánt méretnek megfelelő modell generálásánál a szükséges idő többszöröződik, míg másik módszereknél nem tapasztalható jelentős különbség, így végül nem csak a generálás sebességét mérnénk.

	Objektumok száma								
	10	20	30	60	100	500	1000	4000	6000
Komponens alapú növesztés	737,59	742,96	12873,44	-	-	-	-	-	-
Generatív modellnövesztés helyettesítő objektumokkal	338,82	494,49	535,29	601,12	769,53	1107,22	2926,59	1189,61	-
Generatív modellnövesztés	312,51	11398,81	61618,66	-	-	-	-	-	-
Szélső pontokon keresztüli modellnövesztés	125,65	869,46	2721,71	-	-	-	-	-	-
Elemenkénti modellnövesztés (Helyes modell)	154,34	51,33	135,18	711,98	824,78	1074,79	1289,28	1759,35	-
Elemenkénti modellnövesztés (Nem helyes modell)	38,69	56,48	65,99	116,59	148,49	1120,39	2896,68	4993,04	7118,58

5.1. ábra. A fenti táblázat leírja, hogy mit várunk el a modelltől

Két kiinduló modellünk van. Az első modellben két régió tartalmaz négy váltót amelyek mindegyike össze van kötve a másik hárommal. A második kiinduló modellünk esetén két szegmens és két RégióEnd elem található a két régióban, szintén mindegyik összekötve a másik kettővel. A második bemeneti modellt csak az vasúti zárószakasszal történő növesztés esetén használjuk, a többi módszer az első bemenetből indul ki. Azért szükséges

két különböző bemeneti modell létrehozása, mert a vasúti zárószakaszos növesztés nem tud elindulni kezdeti RégióEnd elem nélkül.

A sorok elején a hat darab futtatott módszer elnevezése található. Ezek a korábban tárgyalt

- Egyszerű elemes modellnövesztés
- Elemenkénti modellnövesztés (helyes modell)
- Elemenkénti modellnövesztés (a modell helyessége nem garantált)
- Generatív modellnövesztés railwayEnd nélkül
- Generatív modellnövesztés railwayEnd felhasználásával
- Szélső Pontokon keresztüli modellnövesztés

A táblázat adatai nanoszekundumban van mérve. Minden mérést ötször végeztünk el, és a kapott eredmények mediánját vettük a táblázat elkészítésekor. A vonallal jelölt cellák esetén a program 3 percen belül nem talált megoldást egyik futtatás során sem.

A mérések alapján az alábbi következtetésekre jutottunk:

- A módszerek mindegyike képes 25-30 nagyságú modellek generálásásra.
- Három módszer képes 60 méretű modellek létrehozására, ebből kettő helyes modelleket generál (Elemenkénti modellnövesztés, Generatív nyelvtan railwayEnd osztállyal) egy módszerre pedig igaz a teljesség (Elemenkénti modellnövesztés).
- Egy módszer képes a helyes modellek létrehozására 1000 méretű modellek esetén.
- A mérések vagy gyorsan, másodpercekben mérhető idő alatt találnak megoldást, vagy több óra alatt sem valószínű, hogy találnak.

Tehát a válaszok a kérdésekre

K1 A legnagyobb generált modell 6000 elem fölött tudott sikeresen modellt generálni viszont többszöri próbálkozásra sem érte el a 7000-es méretet.

K2 A legnagyobb általunk generált helyes modell 4000 pontból állt. Mindez mind elemenkénti mind pedig generatív modellnövesztéssel sikeresen kimérhető volt.

K3 Az általunk generált legnagyobb modellt az elemenkénti modellnövesztés, amennyiben nem ellenőrizzük kényszerekkel a modell helyességét. Ez a megközelítés teljes modellek generálására alkalmas.

A mérésekről további információ a https://github.com/1122username/modelgenerationByDSE honlapon található.

6. fejezet

Kapcsolódó kutatások

Ebben a fejezetben röviden áttekintem a tesztadat generálás főbb megközelítéseit, majd ismertetek néhány már létező modellnövesztési módszert.

Az egyik leginkább elterjedt módszer a kombinatorikus tesztadat generálás specifikáció alapján. Ebben az esetben a program vagy metódus bemeneteit előre megadott lehetséges értékek megadásával lehet különböző teszteseteket generálni, amely sok esetben elég jól lefedi a tesztelni kívánt mechanizmusokat. Egy másik tesztgenerálási techinka egy szimbolikus végrehajtó használata, amely a forráskód végig játszásával kényszereket generál az egyes bementi adatokra és ez alapján jól használható tesztadatokat generál.

A keresésalapú tesztadat generálás hosszú múltra tekint vissza, már a hetvenes években is születtek erről publikációk [14]. Szélesebb körben a kilencvenes években lett aktív kutatási területté, és a kimerítő kereséseket felváltotta a metaheurisztikus keresési módszerek használata, mint például a hegymászó algoritmus, szimulált lehűtés, illetve a genetikus algoritmusok. Ezen technikákat foglalja össze Phil McMinn [12], aki négy részre osztja a tesztadat generálási megközelítéseket:

1. **Strukturális (fehér doboz) alapú tesztelés:** egy szimbolikus végrehajtó végig játsza a forráskódot és az egyes bemenetekre kényszereket generál.

2. **Funkcionális (fekete doboz) alapú tesztelés:** a tesztadatokat kizárlóag a bemeneti adatok lehetséges értékeiből és a program vagy metódus specifikációja alapján származtatják.

3. **Szürke doboz tesztelés:** az előző két techinka ötvözése.

4. **Nem funkcionális tesztelés:** ez esetben a tesztelendő rendszer extra-funkcionális jellemezői szerint generálunk tesztadatokat, mint például legrosszabb vagy legjobb esetbeli végrehajtási idő (worst (best) case execution time - WCET, BCET), erőforrás kihasználtság, biztonság tesztelése (például fuzz testing), stb.

A modellnövesztési módszerek leginkább a modellező eszközök és nyelvek tesztelését szolgálja. Az eddigi módszerekben közös, hogy elsősorban bizonyos kényszerek betartására fókuszálnak azaz konzisztens modellek létrehozására törekednek. A sokszínű, realisztikus és nagy méretű modellek generálása, illetve a modellező eszközök konkrét funkcióinak és működésének figyelembe vétele másodlagosak bár vannak kivételek: E. Brottier [4] kifejezetten modelltranszformációk teszteléséhez generál teszt modelleket. A jelenlegi módszerek alapvetően a generálás folyamatában különböznek, melyek szerint három kategóriába sorolhatóak:

- **Logikai következtetőkön alapuló módszerek.** Ezen módszerek lényege, hogy a generálni kívánt modellre megfogalmazott jól-formáltsági kényszereket SAT/SMT

megoldókra fordítják le. Ilyen módszerre épít a Formula [10], az Alloy [9] és a Clafer [3] is. Ezen módszerekben közös, hogy nehezen skálázódnak, a SAT alapú módszerek kb. 60 elemű modelleket képesek létrehozni értelmes időn belül, míg az SMT alapú módszerek kb. 20 elemig skálázódnak.

- **Szabály alapú módszerek.** Ezek a módszerek, hasonlóan a bemutatotthoz, szabályokat alkalmaznak a modell generálására, de ezeknél csak lokális jól-formáltsági kényszerek megadása lehetséges. [4, 6, 18]. A megközelítés velejárója, hogy nagyságrendekkel jobban skálázódik, mint a logikai következtetőkön alapuló módszerek és lényegesen különböző modelleket is képes generálni.

- **Iteratív módszerek.** Az iteratív módszerek több lépésben generálják a modellt, mint például [13], de ebben a konkrét megközelítésben a modell mérete kötött.

7. fejezet

Konklúzió

7.1. Összefoglalás

A dolgozatban bemutattam egy **új modellgenerálási módszert**, amely képes tetszőleges metamodellhez **példánymodellek automatikus előállítására**. Először a metamodellből és a jólformáltsági kényszerekből **gráftranszformációs lépéseket** állítottam elő, amelyek modelleken növesztési szabályokat fogalmaznak meg. A transzformációs szabályok alkalmazását egy **tervezési-tér bejárás** alapú algoritmussal vezéreltem, amely képes a transzformációs szabályok alkalmazásának sorozatával példánymodellek szintetizálására. A modellek előállítására **több stratégiát** is bemutattam, és értékeltem őket **helyesség** és **teljesség** szempontjából.

Mérnöki eredményeim közé tartozik, hogy megvalósítottam egy **kísérleti prototípust**, amellyet **integráltam az Eclipse fejlesztőkörnyezethez**. A prototípus képes az **EMF modellező nyelvek**hez szabványos modellek előállítására, amely a modellezőeszközök de facto szabványának nevezhető. Ezáltal a modellgenerátor **közvetkenül alkalmazható a legnépszerűbb modellezőeszközökön**, és a modellek akár a saját szerkesztőfelületükben megtekinthetőek. A munkámat a **TraninBenchmark validációs esettanulmányon** szemléltettem. Végül **mérésekkel igazoltam** a modellgenerálási technikám skálázhatóságát, amely helyes modellek generálása esetén **közel két nagyságrenddel meghaladja** a szakirodalomban található logikai következtetők teljesítményét.

A modellgenerátornak két fő felhasználási területe a **tesztbemenet generálás** modellezőeszközök tesztelésére, és a **benchmark modell generálás** modellezőeszközök teljesítménymérésére. Az általam bemutatott technika mindkét esetben jól alkalmazható.

7.2. Jövőbeli tervek

A jövőben két továbbfejlesztési lehetőséget fontolunk megvalósítani:

1. **Stratégiák kombinálása:** A modellgenerálási feladatokat több független lépésben is végre lehet hajtani [1], amelyben a különböző lépéseket különböző stratégiával érdemes végrehajtani. Ezáltal tovább növelhető a generálás skálázhatósága, és ötvözhetőek a különböző stratégiák előnyei.

2. **Realisztikus modellek generálása:** Számos mérőszám áll rendelkezésre valósághű modellek jellemzésére [20]. A generálási folyamat vezérlésével elérhető lehet, hogy a generálás kimeneteként realisztikus modellek készüljenek, ami mindenféleképpen izgalmas iránynak mutatkozik.

Köszönetnyilvánítás

A dolgozat MTA-BME Lendület kutatócsoport támogatásával készült.

Irodalomjegyzék

[1] *Iterative and incremental model generation by logic solvers* (konferenciaanyag), Eindhoven, The Netherlands, 2016. 04/2016.
URL `http://www.etaps.org/index.php/2016/fase/fase-accepted`.

[2] Hani Abdeen – Dániel Varró – Houari Sahraoui – András Szabolcs Nagy – Csaba Debreceni – Ábel Hegedüs – Ákos Horváth: Multi-objective optimization in rule-based design space exploration. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* (konferenciaanyag). 2014, ACM, 289–300. p.

[3] Kacper Bak – Zinovy Diskin – MichałAntkiewicz – Krzysztof Czarnecki – Andrzej Wasowski: Clafer: unifying class and feature modeling. *Software & Systems Modeling*, 2013., 1–35. p.

[4] E. Brottier – F. Fleurey – J. Steel – B. Baudry – Y. Le Traon: Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In *17th International Symposium on Software Reliability Engineering, 2006. ISSRE '06.* (konferenciaanyag). 2006. Nov, 85–94. p. ISSN 1071-9458.

[5] Emf tutorial. `http://eclipsesource.com/blogs/tutorials/emf-tutorial/`. Accessed: 2016-10-27.

[6] F. Fleurey – J. Steel – B. Baudry: Validation in model-driven engineering: Testing model transformations. In *International Workshop on Model, Design and Validation* (konferenciaanyag). 2004. Nov, 29–40. p.

[7] Henshin. `https://www.eclipse.org/henshin/`. Accessed: 2016-10-27.

[8] Benedek Izsó – Gábor Szárnyas – István Ráth – Dániel Varró: Train benchmark technical report. 2014.

[9] Daniel Jackson: Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11. évf. (2002) 2. sz., 256–290. p.

[10] Ethan K Jackson – Gabor Simko – Janos Sztipanovits: Diversely enumerating system-level architectures. In *Proceedings of the 11th ACM Int. Conf. on Embedded Software* (konferenciaanyag). 2013, IEEE Press, 11. p.

[11] Anneke G Kleppe – Jos B Warmer – Wim Bast: *MDA explained: the model driven architecture: practice and promise*. 2003, Addison-Wesley Professional.

[12] Phil McMinn: Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14. évf. (2004. június) 2. sz., 105–156. p. ISSN 0960-0833.
URL `http://dx.doi.org/10.1002/stvr.v14:2`. 52 p.

[13] Aleksandar Milicevic – Joseph P. Near – Eunsuk Kang – Daniel Jackson: Alloy*: A general-purpose higher-order relational constraint solver. In *37th IEEE/ACM Int. Conf. on Software Engineering, ICSE* (konferenciaanyag). 2015, 609–619. p.

[14] C. V. Ramamoorthy – S. B. F. Ho – W. T. Chen: On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2. évf. (1976. Dec) 4. sz., 293–300. p. ISSN 0098-5589.

[15] Grzegorz Rozenberg (szerk.): *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. River Edge, NJ, USA, 1997, World Scientific Publishing Co., Inc. ISBN 98-102288-48.

[16] Oszkár Semeráth – Ágnes Barta – Ákos Horváth – Zoltán Szatmári – Dániel Varró: Formal validation of domain-specific languages with derived features and well-formedness constraints. *Software & Systems Modeling*, 2015., 1–36. p.

[17] Oszkár Semeráth – András Vörös – Dániel Varró: Iterative and incremental model generation by logic solvers. In *International Conference on Fundamental Approaches to Software Engineering* (konferenciaanyag). 2016, Springer, 87–103. p.

[18] Sagar Sen – Naouel Moha – Benoit Baudry – Jean-Marc Jézéquel: Meta-model Pruning. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)* (konferenciaanyag). Denver, Colorado, USA, 2009. Oct.

[19] Dave Steinberg – Frank Budinsky – Ed Merks – Marcelo Paternostro: *EMF: eclipse modeling framework*. 2008, Pearson Education.

[20] Gábor Szárnyas – Zsolt Kővári – Ágnes Salánki – Dániel Varró: Towards the characterization of realistic models: Evaluation of multidisciplinary graph metrics. In *ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS 2016* (konferenciaanyag). Saint Malo, France, 2016. Acceptance rate: 23.7%.

[21] Gábor Szárnyas – Oszkár Semeráth – István Ráth – Dániel Varró: The ttc 2015 train benchmark case for incremental model validation. In *Transformation Tool Contest* (konferenciaanyag). 2015. 07/2015.

[22] The trainbenchmark github repository. `https://github.com/FTSRG/trainbenchmark`. 2016 Okt.

[23] Viatra-dse. `https://wiki.eclipse.org/VIATRA/DSE`. Accessed: 2016-10-27.