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

Modellvezérelt automatatanulás

Szakdolgozat

Készítette
Elekes Márton

Konzulens
Tóth Tamás

2017. december 8.

Tartalomjegyzék

SZAKDOLGOZAT-FELADAT

Elekes Márton
szigorló mérnökinformatikus hallgató részére

Modellvezérelt automatatanulás

Napjainkban egyre elterjedtebben alkalmaznak tanuló algoritmusokat nem teljesen specifikált problémák megoldására. Tanuló algoritmusok egy fajtája az automatatanulás módszere, amely alkalmas megfigyelésekből egy végesautomata-modellt előállítani. Az algoritmus véges ábécé fölötti bemeneti szekvenciákat felhasználva építkezik, folyamatosan finomítva a megtanult viselkedéseket reprezentáló automatát.

Összetett rendszerek tanuláshoz szükséges a külvilágból érkező események leképezése a bemeneti ábécére. A külvilág eseményei ugyanakkor gyakran nem véges értékkészletűek, és összetett relációk, kapcsolatok jellemzik őket. Annak érdekében tehát, hogy az automatatanulás mégis sikeres lehessen, absztrakciót kell alkalmazni, amely lehetővé teszi, hogy a külvilágból érkező komplex eseményeket meg tudjuk tanulni.

A hallgató feladata egy olyan automatatanuló keretrendszer fejlesztése, amely képes absztrakcióval támogatni a külvilágból származó összetett információ feldolgozását.

A hallgató megoldásának a következőkre kell kiterjednie:

- Mutassa be az automatatanulás problémáját.
- Elemezze a bemeneti ábécé absztrakción keresztül történő előállításának lehetőségeit.
- Tervezzen meg egy megoldást modellalapú, absztrakcióval támogatott automatatanulás támogatására.
- Implementálja a megtervezett tanuló rendszert.
- Igazolja esettanulmány segítségével a megközelítés hatékonyságát.

Tanszéki konzulens: Tóth Tamás, tudományos segédmunkatárs

Budapest, 2017. október 8.

……..
Dr. Dabóczi Tamás
tanszékvezető

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

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

HALLGATÓI NYILATKOZAT

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

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

Budapest, 2017. december 8.

Elekes Márton
hallgató

Kivonat

Összetett kiberfizikai rendszerek tervezésére széles körben alkalmaznak modellvezérelt tervezést, amely fejlett modellező nyelvek bevezetésével megkönnyíti a rendszerek megvalósítását. A modellek alkalmasak formális módszerekkel való ellenőrzésre, ami biztonságkritikus feladatkörben - például okos járművek esetében - kiemelten fontos. Továbbá automatikus kódgenerátorok alkalmazásával a rendszer legfőbb komponensei automatikusan előállíthatóak, beleértve a komponensek helyes viselkedéséért felelős monitorozást is.

Azonban számos szoftverkomponensnek – különös tekintettel a korábban vagy külső felek által fejlesztetteknek – nem áll rendelkezésre megfelelően precíz modellje, ami megakadályozza a formális módszerek alkalmazását. Továbbá az elvárt viselkedés specifikációjának hiányában a monitorkomponensek sem képesek detektálni az esetleges hibákat.

Dolgozatom célja egy olyan specializált algoritmus kidolgozása, mely - megfigyelve egy komplex állapottal rendelkező rendszer viselkedését - a megfigyelt események alapján képes formális modelleket szintetizálni. A tanulás támogatására az irodalomból ismert korszerű automatatanuló algoritmusokra támaszkodom. Célom ezen algoritmusok kiterjesztése a mérnöki tervezésben gyakran alkalmazott absztrakció hatékony kezelésével, ezáltal lehetővé téve a fókuszált tanulást.

Megközelítésem újszerűségét a modellalapú bemeneti nyelv és ennek automatatanuló algoritmusokkal való kombinációja adja. Munkám során elkészítettem egy modellalapú automatatanuló keretrendszert komplex állapottal rendelkező szoftverkomponensek mérnöki modelljének szintetizálására. Gráfmintaillesztő nyelvet használok a tanulás fókuszálására, azaz az absztrakció definiálására. A definiált absztrakció alapján az automatatanuló algoritmus előállítja a rendszer formális modelljét. Az elkészült szoftver felhasználását esettanulmányokkal demonstrálom.

A módszerem által lehetővé válik olyan szoftverkomponensek viselkedésének megtanulása, amelyekre eddigiekben nem volt lehetőség komplexitásuk, illetve adatfüggő viselkedésük miatt. Az így létrejövő specifikációból a komponens viselkedését formálisan ellenőrizhetjük, anomáliadetektáló monitort származtathatunk, valamint dokumentációt, illetve teszteket generálhatunk.

Abstract

Model-driven engineering is a widely used approach for designing complex cyber-physical systems that is based on the use of high-level system modeling languages. Models can be used for formal verification, which is important for safety-critical systems, e.g. vehicular automation. Moreover, the components of the system can be generated using automatic code generation, including monitoring components that continously check the expected behavior.

However, for numerous software components - mainly legacy and third-party components - there is often no precise specification given. The lack of a formal specification prevents the use of formal verification. Without a specification the expected behavior cannot be monitored. The construction of complex systems is especially challenging.

The goal of my work is to create a specialized algorithm to synthesize formal models of complex systems by observing the events of a black-box system. For this purpose I use state-of-the-art automaton learning algorithms. My goal is to extend these algorithms to focus the learning to efficiently capture the abstraction techniques used in engineering.

The novelty of my approach is a model-based input language and its combination with automaton learning algorithms. I developed a model-based automaton learning framework for synthesizing formal models of software components with complex states. I use a graph query language to focus the learning on important properties of the target component, while excluding unimportant details. Therefore the automaton learning algorithm produces the formal model of the target system. I demonstrate my approach with case studies.

As a result of my work, it is possible to learn the behavior of software components that have not been possible yet due to their complexity or data-dependent behavior. Additionally, the model produced during the learning can be used as documentation, to derive monitors for anomaly detection, or to create test inputs automatically.

1. fejezet

Bevezetés

Kontextus. Kiberfizikai rendszerek olyan, jellemzően fizikai közegbe ágyazott intelligens rendszerek, amelyek szenzorokon keresztül észlelik környezetüket, majd beavatkozókon keresztül visszahatnak arra. Komplex viselkedéseik miatt különösen fontos olyan módszerek választása a fejlesztés során, amelyek segítenek elkerülni a hibákat. A jó minőségű modellek nagyban hozzájárulnak a komplex kiberfizikai rendszerek helyességéhez és megbízható működéséhez. Emiatt biztonságkritikus kiberfizikai rendszerek fejlesztésére széles körben használnak modellvezérelt tervezést, amely során lehetőség nyílik a formális módszerek eszköztárát felhasználni a rendszertervek ellenőrzésére már a fejlesztés korai fázisaiban is. Emellett a formális modellekből előállított monitorok segítségével a futásidőben bekövetkező anomáliák is kiszűrhetőek.

Problémafelvetés. Precíz modellek alkotása nehéz feladat, különösen kiberfizikai rendszerek esetén, mivel ilyen rendszerek viselkedése tipikusan komplex módon függ a környezettől. Ráadásul a valós rendszerek adatfüggő viselkedéssel rendelkeznek, amit szintén nehéz precízen modellezni. Emellett gyakran gondot okoz az is, hogy bizonyos komponensek belső viselkedése nem ismert: vagy nem áll rendelkezésre dokumentáció, vagy még a forráskód sem ismert (például egy zárt, külső beszállítótól érkező komponens esetén). Ezekben az esetekben nincsen lehetőség pontos rendszerterveket készíteni, amely meggátolja, hogy formális ellenőrzés segítségével megbizonyosodhassunk a tervek helyességéről.

Célkitűzés. Dolgozatom célja egy olyan módszer megalkotása, mellyel kritikus kiberfizikai rendszerek viselkedésmodelljét tudjuk automatikusan előállítani tanuló algoritmusok segítségével. Célom, hogy akár komplex, gráfstruktúrával jellemezhető rendszerek viselkedését is tanulhatóvá, megismerhetővé tegyem. Emellett fontos célom az is, hogy a komplexitás kezelésének módja a felhasználó által állítható legyen.

Kontribúció. Dolgozatom előzménye egy, a szerzőtársammal közösen készített TDK-dolgozat [8]. Dolgozatomban bemutatok egy véges automata alapú viselkedési modelleket tanuló megközelítést. Szerzőtársammal közösen implementált tanulóalgoritmust gráf alapú modellek változásainak tanulására tettem képessé. Az új tanuló algoritmus:

- gráfmintákat használ a tanulás számára releváns viselkedések definiálására,
- gráfmintaillesztő algoritmusok alapján állítja elő a tanulás során felhasznált eseményeket, továbbá
- gráfminták alapján hajtja végre az automatatanulás során szükséges transzformációkat.

Legjobb tudomásom szerint az én megközelítésem az első, amely hatékonyan kezeli a gráf- és adatjellegű viselkedéseket viselkedésmodellek szintézise során.

Hozzáadott érték. Megközelítésem által rendszerek szélesebb köre válik a formális analízis által vizsgálhatóvá. Az általam adott algoritmus segítségével komplex rendszerek adatjellegű, és akár gráfszerű viselkedései is tanulhatóvá, megismerhetővé válnak.

A dolgozat felépítése. Dolgozatom kifejtését a 2. fejezetben az előismeretek áttekintésével kezdem. Általánosan bemutatom az automatákhoz és automatatanuláshoz szükséges alapismereteket, majd az absztrakció bevezetéséhez szükséges ismereteket a tulajdonsággráfokról, illetve a gráfmintákról. A 3. fejezetben az általam kidolgozott keretrendszert ismertetem, valamint bemutatok egy esettanulmányt, melyre alkalmaztam a keretrendszert. A 4. fejezetben a tanulás során használt absztrakciót, a 5. fejezetben a szerzőtársammal közösen megvalósított tanulóalgoritmust mutatjuk be elméleti szinten. Az absztrakció és a tanuló algoritmus gyakorlati megvalósításáról a 6. fejezetben írok. A dolgozatomhoz kapcsolódó munkákat a 7. fejezetben fejtem ki. Végül a 8. fejezetben összefoglalom az elvégzett munkát, és kifejtem a jövőbeli terveket.

2. fejezet

Előismeretek

Ebben a fejezetben a dolgozat további részeinek megértéséhez szükséges elméleti előismeretekről lesz szó.

2.1. Alapfogalmak, jelölések

Ebben az alfejezetben a dolgozat során használt alapfogalmakat, jelöléseket sorolom fel. Ezek nagyrészben megegyeznek a formális nyelvek elméletéből ismert jelölésekkel [3], azonban az automatatanulás témaköréből további fogalmakat is ismertetek.

Az automaták témakörében egyértelműen definiálva van, hogy az *ábécé*, *karakter*, *szó* és *nyelv* fogalmak mit jelentenek. Mindig fontos megállapítani, hogy egy automata milyen ábécé felett van értelmezve.

Definíció 1 (Ábécé, karakter). Egy tetszőleges, nem üres, véges halmazt ábécének nevezünk. Jelölése: Σ. A Σ ábécé elemeit betűknek, avagy karaktereknek nevezzük. ▪

Definíció 2 (Bemeneti esemény). Dolgozatomban a bemeneti esemény egy karaktert jelent, hiszen az automatatanulás során a tanuló algoritmus a rendszerrel bemeneti eseményeken keresztül kommunikál. ▪

A karakterekből képezhető sorozatokat szavaknak nevezzük.

Definíció 3 (Szó). Egy szó a Σ ábécé elemeiből, karaktereiből képezhető véges sorozat. Az ω szó hosszát $|\omega|$ jelöli. A Σ ábécén képezhető összes szó halmazát Σ^* jelöli. ▪

Definíció 4 (Üres szó). Üres szónak nevezzük azt a szót, mely nem tartalmaz egyetlen karaktert sem. Jelölése: ϵ. ▪

Definíció 5 (Lefutás). A lefutás a bemeneti események sorozata a rendszer egy adott végrehajtása során. Dolgozatomban a lefutás és szó szavakat szinonimaként használom. ▪

Az ábécéből képezhető szavak egy részhalmazára nyelvként hivatkozunk.

Definíció 6 (Nyelv). Egy Σ ábécé feletti L nyelvnek nevezzük az ábécé elemein képezhető összes szó egy (nem feltétlenül véges) részhalmazát ($L \subseteq \Sigma^*$). ▪

2.2. Véges automata

A véges automata [3, 10] az egyik legegyszerűbb számítási modell, amely annak eldöntésére használható, hogy az egyes szavak az adott nyelv elemei-e. Ezek a nyelvek akár rendszerek biztonságos működését jelentő lehetséges lefutások halmazai is lehetnek. A véges automatát általában DFA-val rövidítjük (Deterministic Finite Automaton).

Definíció 7 (Véges automata). A véges automatát a $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, F \rangle$ struktúra írja le, ahol

- Q az automata állapotainak véges, nem üres halmaza,
- Σ az automata ábécéjének véges, nem üres halmaza,
- $\delta : Q \times \Sigma \to Q$ az automata állapotátmeneti függvénye,
- $q_0 \in Q$ a kezdőállapot és
- $F \subseteq Q$ az elfogadó állapotok halmaza. ▪

Egy véges automata működése a következőképpen írható le egy adott $\omega = a_1 a_2 \dots a_n \in \Sigma^*$ szón:

Definíció 8. Az $r_0, r_1, r_2, \dots, r_n$ $(r_i \in Q)$ állapotsorozat az $a_1 a_2 \dots a_n$ szóhoz tartozó számítás, ha $r_0 = q_0$ és $r_i = \delta(r_{i-1}, a_i)$ minden $i = 1, 2, \dots, n$ esetén. ▪

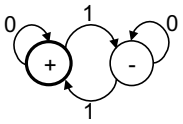
Tehát az automatát a $q_0 = r_0$ állapotból indítva rendre beolvassuk a karaktereket, és a következő állapotba a beolvasott karakter és a jelenlegi állapot által meghatározott átmeneti függvény szerint lépünk.

Egy szót elfogad az automata, ha a legutolsó beolvasott karaktere után az automata egy elfogadó állapotba ér. Ellenkező esetben az automata nem fogadja el, vagyis elutasítja a szót. Egy szó eleme az automata által leírt nyelvnek, ha az automata elfogadja azt. Az alábbiakban a vonatkozó precíz definíciókat tekintjük át.

Definíció 9. Az $\mathcal{M}$ automata elfogadja $\omega \in \Sigma^*$ szót, amennyiben $|\omega| = n$ és az ω szóhoz tartozó számítás végén r_n állapot egy elfogadó állapota az automatának, vagyis $r_n \in F$. Egyébként $\mathcal{M}$ nem fogadja el, más néven elutasítja ω szót. ▪

Definíció 10 (Véges automata nyelve). Az $\mathcal{M}$ véges automata nyelve azon ω szavak összessége, melyeket $\mathcal{M}$ elfogadja. Jelölése: $L(\mathcal{M})$. A korábbiakból következik, hogy $L(\mathcal{M}) \subseteq \Sigma^*$. ▪

Példa 1. *Legyen $\Sigma = \{0, 1\}$, vagyis az ábécé karakterei legyenek 0 és 1. Legyen M egy olyan automata, melynek $L(\mathcal{M})$ nyelvébe azok a szavak tartoznak, amelyekben páros darab 1 karakter szerepel.*



2.1. ábra. Az 1. példában leírt automata ábrázolása

A 1. példában leírt automatát a 2.1. ábrán látható módon ábrázolhatjuk grafikusan. Dolgozatomban ezt az ábrázolásmódot fogom a későbbiekben is alkalmazni. Ennek a következő elemei vannak:

Állapot. Az $\mathcal{M}$ automata állapotait körök jelölik.

Állapotátmenet. Az $\mathcal{M}$ automata állapotátmeneteit nyilak jelölik, melyek az átmenet kezdőállapotából a végállapotába mutatnak. Az állapotátmenet karakterét a nyílra írjuk rá.

Elfogadó állapot. Az $\mathcal{M}$ automata elfogadó állapotait a + jelzés jelöli.

Nem elfogadó állapot. Az $\mathcal{M}$ automata nem elfogadó állapotait a − jelzés jelöli.

Kezdőállapot. Az $\mathcal{M}$ automata kezdőállapotát, az r_0 állapotot vastagított körvonal jelöli.

2.2.1. Minimális véges automata

Belátható az, hogy egy adott L nyelvre létezhet több automata is, melyek az L nyelv szavait fogadják el. Az 1. példában szereplő nyelvvel a 2.2. ábra mindkét automatájának nyelve megegyezik. Ezeknek az automatáknak nem feltétlenül ugyanakkora az állapottere, vagyis nem ugyanannyi állapotból állnak. Ezen automaták közül a legkevesebb állapottal rendelkezőt minimális automatának nevezzük.

(a) Minimális véges automata **(b)** Nem minimális véges automata

2.2. ábra. Véges automata állapottere

Definíció 11 (Állapottér). $\mathcal{M}$ véges automata állapotterének mérete alatt az állapotainak számát értjük, vagyis Q elemeinek a számát. ▪

Definíció 12 (Minimális véges automata). Egy L nyelvhez tartozó minimális automata egy olyan $\mathcal{M}$ automata, melyre igaz, hogy $L(\mathcal{M}) = L$ és az ilyen automaták közül $\mathcal{M}$ állapottere a legkisebb. ▪

2.2.2. Teljes és hiányos véges automata

Sokszor, főként kényelmi és átláthatósági szempontokból, nem szokták az állapotátmeneti függvényt minden lehetséges állapotra és karakterre ábrázolni (pl. ha egy adott karakterre ugyanabban az állapotban fog maradni az automata). Abban az esetben, ha hangsúlyozni akarjuk, hogy nem hiányos automatáról van szó, akkor a 7. definícióban szereplő automatát teljes véges automatának nevezzük.

A 2.3. ábrán látható, hogy az 1. példában leírthoz hasonló, páros számú 1 karaktert elfogadó nyelvet hogyan lehet egy teljes, illetve egy hiányos automatával ábrázolni. Az ábra azt is mutatja, hogy ebben az esetben indokolt is lehet hiányos automatát alkalmazni, hiszen a páros számú 1 karaktert tartalmazó szavak elfogadásához a 0 karakterekkel nem kell foglalkozni.

(a) Teljes véges automata **(b)** Hiányos véges automata

2.3. ábra. Véges automata teljessége

Definíció 13 (Teljes véges automata). Teljes véges automatának nevezünk egy olyan $\mathcal{M}$ véges automatát, amelyben minden $q \in Q$ állapotra, minden $a \in \Sigma$ karakterre a δ állapotátmeneti függvény definiálva van. ▪

Definíció 14 (Hiányos véges automata). Egy $\mathcal{M}$ nemteljes véges automatát hiányos véges automatának nevezünk. ▪

Hiányos véges automaták esetén többféle megoldás is lehetséges, amikor egy ω szó számítása során olyan karakter következne, amelyre az automata adott állapotában nincsen állapotátmenet értelmezve. Egyik lehetőség, hogy a számítás elakad. Ebben az esetben az automata nem fogadja el az ω a szót. Másik lehetséges megoldás, hogy az automata figyelmen kívül hagyja (hurokélként értelmezi) a nem definiált átmenetet. A két megoldás különbözik, pl. a 2.3. ábrán látható automata az $1, 0, 1$ szót az előbbi értelmezés szerint nem fogadná el, az utóbbi értelmezés szerint igen.

2.2.3. Determinisztikus és nemdeterminisztikus véges automata

A nyelvek leírását sokszor megkönnyíti, ha nem csak elhagyhatunk átmeneteket, de azt is megengedjük, hogy ne legyenek egyértelműek azok. Ez azt jelenti, hogy egy állapotból egy karakterre nem csak egy lehetséges állapotba juthat az automata. Az ilyen, nem egyértelmű állapotátmeneteket tartalmazó automatákat nemdeterminisztikus automatáknak nevezzük, míg a csak egyértelmű állapotátmenettel rendelkezőket determinisztikus automatáknak.

A 2.4. ábra ábrázol egy az 1. példában definiált nyelvre adott determinisztikus és nemdeterminisztikus automatát.

(a) Determinisztikus véges automata **(b)** Nemdeterminisztikus véges automata

2.4. ábra. Véges automata determinisztikussága

Definíció 15 (Determinisztikus véges automata). Determinisztikus véges automatának nevezzük a 7. definícióban definiált véges automatát. Az $\mathcal{M}$ véges automatában, minden $q \in Q$ állapotból, minden $a \in \Sigma$ karakterre legfeljebb egy állapotátmenet van definiálva. ▪

Definíció 16 (Nemdeterminisztikus véges automata). Nemdeterminisztikus véges automata definíciója abban tér el a 7. definíciótól, hogy $\delta : Q \times (\Sigma \cup \varepsilon) \to 2^Q$. Azaz az automata δ állapotátmeneti függvény szerint $q \in Q$ állapotból és $a \in \Sigma$ karakter hatására több Q-beli állapotba, azaz az állapotok egy halmazába kerülhet. Továbbá ε hatására, karakter nélkül is új állapotba léphet. ▪

Egy nemdeterminisztikus véges automata akkor fogad el egy szót, ha létezik benne olyan lefutás, amelyben a szó utolsó karaktere elfogadó állapotba juttatja.

Általában amikor véges automatáról beszélünk, akkor determinisztikus véges automatára *(deterministic finite automaton)* gondolunk. Ezért is használjuk a DFA rövidítést.

2.3. Automatatanulás

Black box rendszernek nevezünk egy olyan rendszert, melynek pontos belső működése nem ismert, csupán a külvilággal való kommunikációja. Az automatatanulás célja egy adott black box rendszer működésének feltérképezése az elérhető információk alapján. Az automatatanulás eszközt biztosít arra, hogy a rendszer lefutásait vizsgálva megtanuljunk egy automatát, amely jól modellezi annak belső működését.

Általában egy tanuló algoritmustól elvárjuk, hogy az általa megtanult hipotézismodellre a következők igazak legyenek:

- **Determinisztikus.** Az automata későbbi felhasználása (tesztesetek generálása, anomáliadetektálás, stb.) során előnyös, ha az automata determinisztikus. Ezt a kritériumot általában tudja teljesíteni egy tanuló algoritmus.
- **Teljes.** A teljesség azért fontos, hogy minden állapotban tudjuk, hogy egy bemenetre hogyan viselkedne a rendszer. Ez a kritérium nem tud mindig teljesülni. Főként passzív tanulás (2.3.3. fejezet) esetén fordulhat elő, hogy az algoritmusnak kevés információ áll rendelkezésére.
- **Minimális.** Szemléletesség miatt fontos, illetve ilyenkor kevesebb memória szükséges az automata tárolásához. Általában nem tudja garantálni a minimális automata megtanulását egy algoritmus, de törekszik rá.

Ezen kívül gyakorlati szempontból, nem csak a megtanult automatára, de az algoritmusra nézve is vannak elvárásaink:

- **Minimális futásidő.** Szeretnénk, ha egy algoritmus minél kevesebb idő alatt, minél kevesebb kérdéssel, az automata minél kevesebbszer történő fölösleges módosításával tudna tanulni. Ezt sem tudja általában garantálni egy tanuló algoritmus, de különféle optimalizációkkal lehet törekedni rá.

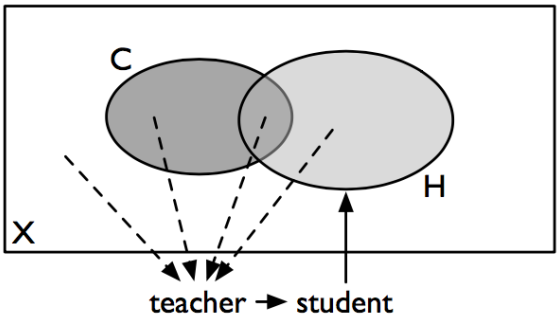
Többféle algoritmus létezik, változatos formalizmusokat és rendszereket támogatva. A különböző algoritmusok többek között az előállított automata típusában, illetve a tanulás módjában térhetnek el.

2.3.1. Alapfogalmak

A tanuló algoritmusok a vizsgálat során a rendszernek adott input események sorozatát és az azokra adott output reakciókat vizsgálja. A legegyszerűbb algoritmusok esetében a kimenet csupán azt az információt hordozza, hogy az adott bemeneti sorozat érvényes lefutást jelent-e. Ezáltal a tanulás eredményeként egy véges automata alapú modellt áll elő, mely jól modellezi a rendszer működését. Minden tanulás egy tanulóból, studentből (s) és egy tanárból, teacher (t) áll. A tanuló célja, hogy kikövetkeztessen egy hipotézist (H) a rendszer (C) működéséről.

Legyenek a rendszernek küldhető üzenetek, vagyis a rendszer lehetséges inputjai a tanulás ábécéjének karakterei. Nevezzük az ábécéből előállítható szavak halmazát (Σ^*) X-nek. A tanár a tanulót X elemeivel tanítja, elárulva a rendszer arra adott válaszát is (2.5. ábra).

Definíció 17 (Rendszer). Egy C rendszer által definiált nyelv részhalmaza X-nek, amennyiben C bemeneti ábécéjéből előállítható szavak halmaza X. Egy $x \in X$ szó egy pozitív (lefutási) példa, amennyiben $x \in C$, ellenkező esetben egy negatív példa. ▪



2.5. ábra. A tanulásban résztvevő komponensek

Definíció 18 (Hipotézis). H hipotézis egy hipotézismodellje C rendszernek X fölött. Egy $x \in X$ szó igaznak értékelődik ki H szerint, amennyiben $x \in H$, ellenkező esetben hamis. Egy H hipotézismodell helyes C-re nézve, amennyiben $x \in X$ akkor és csak akkor igaz H-ban, ha x pozitív példa C-ben. Vagyis ha H és C nyelve megegyezik. ▪

Definíció 19 (Tanár). C rendszer tanára egy olyan orákulum, mely megmondja egy $x \in X$ elemről, hogy x szó C szerint egy pozitív vagy negatív példa-e. Vagyis visszatérési értéke egy címkézett (x, b), ahol b egy logikai érték, mely akkor *igaz*, ha $x \in C$, ellenkező esetben *hamis*. ▪

Definíció 20 (Tanuló). Egy s tanuló maga a tanuló algoritmus, ami megtanulja a H hipotézist, t tanár segítségével, akinek van tudása a C rendszerről. A tanuló célja, hogy találjon egy helyes H hipotézist, melyre $x \in H$ akkor és csak akkor, ha $x \in C$. ▪

Véges automatát tanuló algoritmusból is sokféle létezik. A különböző algoritmusokat [12] az alábbi szempontok alapján szokás megkülönböztetni:

- **Aktív vagy passzív tanulás.** Aktív tanulás során az algoritmus, ha szükséges, feltehet további kérdéseket a rendszer működését illetően (pl. megkérdezheti egy lefutásról, hogy pozitív-e), ezzel bővíteni tudja a tudását. Ezzel ellentétben a passzív tanulás során az algoritmusnak egy előre megadott lefutáshalmazzal kell dolgoznia, melyet később nincs lehetősége bővíteni.

- **Online vagy offline tanulás.** Online tanulás során minden információhoz csak egyszer tud hozzáférni az algoritmus, ekkor kell beépítenie a modellbe. Így ezen algoritmusok működése igen időkritikus. Offline tanulás során az információ a rendszerről tárolva van. Emiatt az algoritmus többször is lekérdezheti őket, illetve akár előfeldolgozáson is áteshetnek.

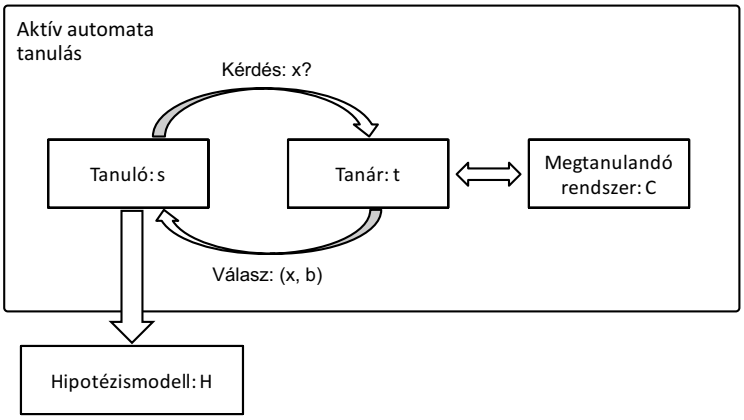
- **Csak pozitív, vagy pozitív és negatív elemek.** Néhány algoritmus csak pozitív elemekkel dolgozik, vagyis csak olyan $x \in X$ példákat kap a tanárától, melyekre az (x, b) pár b része *igaz*. Azonban általában egy tanulás során pozitív és negatív elemek is rendelkezésre állnak.

- **Megengedett-e olyan állapot a hipotézismodellben, mely nem tartalmaz információt.** Néhány tanuló algoritmusnál nem követelmény, hogy a kapott H hipotézisautomata minden állapotról eldönthető legyen, hogy elfogadó vagy elutasító. Ilyen például akkor lehetséges, ha kevés információnk van a lefutásokról, kevés az elemünk.

Az irodalomban gyakran összekeverik az aktív és az online, valamint a passzív és az offline tanulást. Ez azért van, mert a gyakorlatban ezek a tulajdonságok gyakran egyszerre jelennek meg a tanulóalgoritmusokban. Az aktív és passzív tanulás közötti különbséget egy-egy egyszerű példán keresztül szemléltetjük (2. és 3. példa).

2.3.2. Aktív automatatanulás

Az aktív tanulóalgoritmus eleinte nem rendelkezik információval a megtanulandó rendszerről. Ilyenkor a kezdeti hipotézismodellje egy nagyon általános automata (általában egy X minden elemét elfogadó automata). A tanulás során az algoritmus kérdezgeti a megfigyelt rendszert. A kérdéseire kapott válaszok alapján tudja pontosítani a rendszert és felépíteni arról az új hipotézismodellt (2.6. ábra).



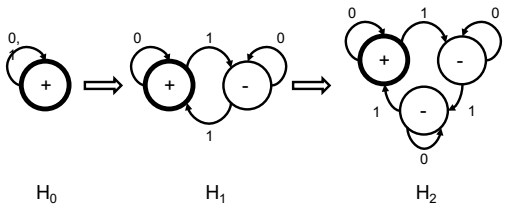
2.6. ábra. Aktív automatatanulás

Az algoritmus kétféle lekérdezést tud megfogalmazni a tanulandó rendszer számára:

- **Membership query, avagy tagsági kérdés.** A membership query-k segítségével a rendszert meghajtja annak lehetséges bemeneteivel, majd az ezekre kapott válaszok alapján, a bemenet-kimeneti párokat építve tanulja meg a viselkedéseket. A kérdéseit az alapján választja meg, hogy miből tudja meg a legtöbb információt. Az aktuális bementet az algoritmus mindig a jelenlegi tudása alapján határozza meg.
- **Equivalence query, avagy ekvivalencialekérdezés.** Ha a tagsági kérdések alapján az algoritmus úgy véli, hogy megtanulta az automatát, akkor egy ekvivalenciatesztet hajt végre az automata és a rendszer működése között. Ennek kimenete kétféle lehet:
  - **Talál ellenpéldát.** Ha a teszt során ellenpéldára talál, annak alapján folytatja a tanulást a tagsági kérdésekkel.
  - **Nem talál ellenpéldát.** Ha azonban nem talál ellenpéldát, akkor a tanulás befejeződik, a megtanult automata jól reprezentálja a rendszert.

Példa 2. *Legyen $\Sigma = \{0, 1\}$, vagyis az ábécé karakterei legyenek 0 és 1. Legyen X továbbra is Σ^*, vagyis a karakterekből képezhető összes szó halmaza. Legyen a C által elfogadott nyelv azon x elemek halmaza, ahol $x \in X$, és ahol az x szóban szereplő 1 karakterek száma hárommal osztható.*

Ha megfigyelnénk egy aktív tanuló algoritmust, mely a 2. példában definiált rendszert tanulja és a tanulás különböző fázisaiban lekérdeznénk az aktuális hipotézismodelljét, akkor a 2.7 ábrán láthatóhoz hasonló eredményt kapnánk.



2.7. ábra. Aktív tanulás során felépített hipotézismodellek

Az aktív automatanulás széles körben használt módszer (7.1., 7.3. fejezet). Számos különböző hatékonyságú változata van, illetve többféle automata megtanulására is léteznek változatok. Az algoritmus alapötlete még 1987-ből, Angluintól [2] származik. Az algoritmus a kezdeti hipotézismodellt úgy határozza meg, hogy lekérdezi a rendszer üres szóra adott reakcióját. Ezt követően ekvivalenciakérdést fogalmaz meg a tanárának. Ha ellenpéldát kap válaszul, akkor az ellenpélda alapján tagsági kérdéseket fogalmaz meg. Miután az összes tagsági kérdésére kapott választ, újból megvizsgálja az ekvivalenciát. Ezt mindaddig folytatja, míg nem kap ellenpéldát. Az algoritmus működését az 1. pszeudokód írja le.

Algoritmus 1: Angluin-féle aktív automatatanulás

Input: C rendszer és t tanára, aki választ ad a mem() tagsági és az eq() ekvivalencia kérdésekre

Output: H hipotézismodell C rendszer viselkedéséről

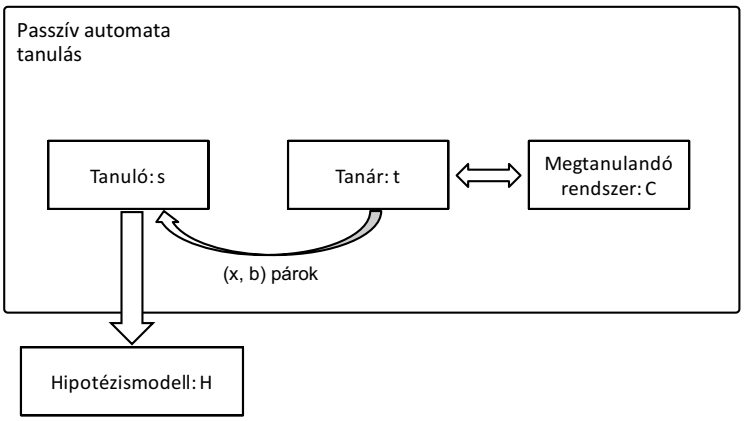
```
Kérdezze meg mem(ϵ) tagsági kérdést, hogy a kezdőállapotról kiderüljön, hogy
elfogadó vagy nemelfogadó.
Állítsa elő egy kezdeti H hipotézismodellt, mely ezt az egy állapotot tartalmazza.
while eq(H) ellenpéldával tér vissza do
    while van kérdés a rendszer felé do
        Kérdezze mem(seq) tagsági kérdést a tanártól, ahol seq a kérdéses szó.
    end
    Vizsgálja meg az ekvivalenciát: eq(H).
end
return H
```

2.3.3. Passzív automatatanulás

A passzív tanulás főként abban különbözik az aktív tanulástól, hogy annak nincs lehetősége kommunikálni a megtanulandó rendszerrel (2.8. ábra). Az algoritmus egy előre megkapott információhalmazból dolgozik, nem tehet fel további tagsági kérdéseket, amelyekkel több tudást tudna szerezni.

Passzív tanulás során az algoritmus bemenetként kap számos, eddig már előfordult, megfigyelt lefutást. Ezeket a lefutásokat a tanulás tanárától kapja, így azzal az információval is rendelkezik, hogy igazak, vagy sem a megtanulandó rendszerre nézve. Az algoritmus belőlük eleinte egy lefutási fát (21. definíció) épít, majd azt megpróbálja determinisztikusan összevonni, minimalizálni [19].

Definíció 21 (Lefutási fa). A lefutási fa, röviden APTA (Augmented Prefix Tree Acceptor) egy fa automata reprezentációja a tanulás során felhasznált pozitív és negatív elemeknek. Minden egyes elem, lefutás megtalálható a fában, annak egy útvonalaként, mely a fa gyökeréből indul. A pozitív és a negatív lefutásokhoz tartozó útvonalak végét jelző állapotok rendre elfogadóak vagy nem elfogadóak.



2.8. ábra. Passzív automatatanulás

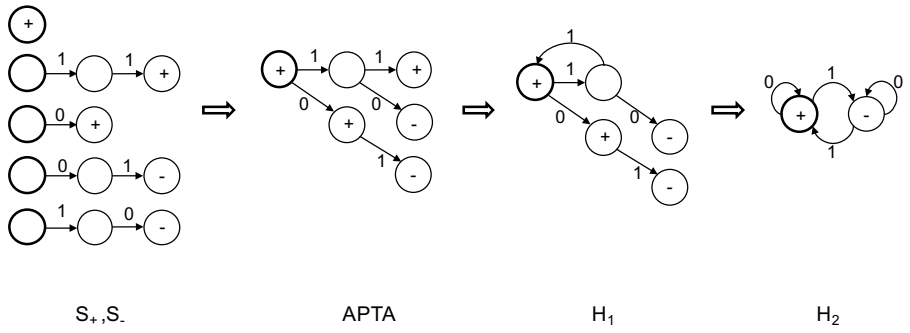
Az APTA így tulajdonképpen egy hiányos véges állapotgépnek felel meg, melynek nem minden állapotáról dönthető el, hogy elfogadó-e. ▪

Példa 3. *Legyen* $\Sigma = \{0, 1\}$*, vagyis az ábécé karakterei legyenek 0 és 1. Legyen* X *továbbra is* Σ^**, vagyis a karakterekből képezhető összes szó halmaza. Legyen a* C *által elfogadott nyelv azon* x *elemek halmaza, ahol* $x \in X$*, és ahol az* x *szóban szereplő* 1 *karakterek száma kettővel osztható.*

A 3. példában leírt rendszer megtanulási folyamatát a 2.9. ábra szemlélteti. A tanulás első fázisában a tanuló megkapja a tanártól a pozitív és negatív lefutásokat. Ezek a pozitív és negatív lefutások most legyenek rendre a következők:

- $S_+ = \epsilon, 11$
- $S_- = 0,\ 10,\ 01$

Ezek után az algoritmus felépíti ezeknek megfelelően a lefutási fát, majd ameddig tud, kiválaszt az automatában olyan állapotokat, amelyeket összevonva nem jut ellentmondásra, és így csökkentheti az automata méretét.



2.9. ábra. Passzív tanulás során felépített hipotézismodellek

A passzív algoritmusoknak is sokféle változata létezik, különböző típusú automaták megtanulására. A 2. pszeudokód írja le általánosan egy passzív tanuló algoritmus működését. Ennél konkrétabb megvalósításokról lesz még szó az 5. fejezetben.

Algoritmus 2: Passzív automatatanulás

Input: S pozitív és negatív lefutások halmaza
Output: H DFA, ami kicsi és konzisztens S-sel

```
H = apta(S)
while van lehetőség összevonni do
    Összevon két állapotot.
end
return H
```

2.4. Tulajdonsággráf

Adatmodell. A gráf alapú tudásbázis reprezentálására tulajdonsággráfot használunk, amely egy irányított gráf, címkézett csúcsokkal és típusos élekkel. A csúcsok és élek elláthatóak továbbá tetszőleges tulajdonságokkal, így a tulajdonsággráf alkalmas komplex rendszerek állapotának ábrázolására.

Definíció 22 (Tulajdonsággráf). A tulajdonsággráfot (*property graph*) a

$$G = \langle V, E, \mathsf{src\_trg}, L_v, L_e, l_v, l_e, P_v, P_e \rangle$$

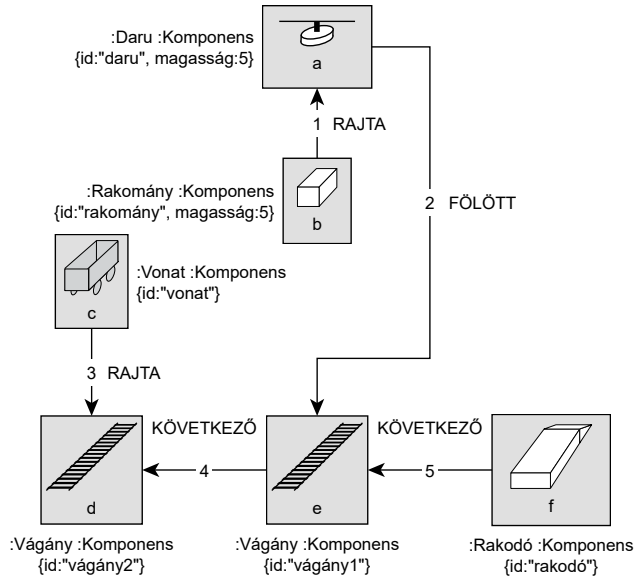
struktúra írja le, ahol V a csúcsok halmaza, E az élek halmaza és $\mathsf{src\_trg} : E \to V \times V$ az élekhez a kezdő- és a végpontjukat hozzárendelő függvény. A csúcsokat címkével, az éleket típussal látjuk el:

- L_v a csúcsok címkéinek halmaza, $l_v : V \to 2^{L_v}$ minden csúcshoz *címkék egy halmazát* rendeli.
- L_e az élek típusának halmaza, $l_e : E \to L_e$ minden élhez *típust* rendel.

A tulajdonságok definiálásához $D = \bigcup D_i$ a D_i elemi tartományok uniója és NULL jelöli a NULL, üres értéket.

- P_v a csúcsok tulajdonságainak halmaza. A $p_i \in P_v$ csúcstulajdonság egy olyan $p_i : V \to D_i \cup \{\mathsf{NULL}\}$ függvény, amely egy tulajdonság értéket rendel a $D_i \in D$ tartományból a $v \in V$ csúcshoz, ha v rendelkezik a p_i tulajdonsággal, egyébként $p_i(v)$ értéke NULL.
- P_e az élek tulajdonságainak halmaza. A $p_j \in P_e$ éltulajdonság egy olyan $p_j : E \to D_j \cup \{\mathsf{NULL}\}$ függvény, amely egy tulajdonság értéket rendel a $D_j \in D$ tartományból a $e \in E$ élhez, ha e rendelkezik a p_j tulajdonsággal, egyébként $p_j(e)$ értéke NULL. [13] ▪

Példa 4. *A 2.10. ábrán látható egy vasúti rakodóállomás gráfmodellje vágányokkal, vonatokkal, rakománnyal, daruval és rakodóhellyel. A különböző komponenseket típusuk szerint címkével láttuk el, a köztük lévő fizikai viszonyokat élek jelölik. Minden csúcs egyedi* id *tulajdonsággal van ellátva, illetve a daru és a rakomány függőleges helyzetét a* magasság *tartalmazza (méterben). A gráf formálisan a 2.11. ábrán látható.*



2.10. ábra. Rakodóállomás gráfmodellje

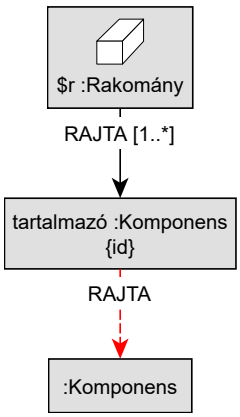
$V = \{a, b, c, d, e, f\}; E = \{1, 2, 3, 4, 5\};$
$\mathsf{src\_trg}(1) = \langle b, a\rangle; \mathsf{src\_trg}(2) = \langle a, e\rangle; \ldots$
$L_v = \{\mathsf{Daru}, \mathsf{Rakomány}, \mathsf{Vonat}, \mathsf{Vágány}, \mathsf{Rakodó}, \mathsf{Sérült}, \mathsf{Komponens}\};$
$L_e = \{\mathsf{RAJTA}, \mathsf{FÖLÖTT}, \mathsf{KÖVETKEZŐ}\};$
$l_v(a) = \{\mathsf{Daru}, \mathsf{Komponens}\}; l_v(b) = \{\mathsf{Rakomány}, \mathsf{Komponens}\}; \ldots;$
$l_e(1) = \mathsf{RAJTA}; l_e(2) = \mathsf{FÖLÖTT}; \ldots;$
$P_v = \{\mathsf{id}, \mathsf{magasság}\}; P_e = \{\};$
$\mathrm{id}(a) = \mathsf{'daru'}; \mathrm{id}(b) = \mathsf{'rakomány'}; \ldots$
$\mathrm{magasság}(a) = 5; \mathrm{title}(b) = \mathsf{NULL}; \ldots$

2.11. ábra. Rakodóállomás formális tulajdonsággráfja

2.5. Gráfminta

Lekérdező gráfmintákat alkalmazunk, hogy a gráfban tárolt komplex információkból kigyűjtsük a vizsgálatunk szempontjából relevánsakat. A lekérdező gráfminták a csúcsok és a köztük lévő élek, azok címkéi, típusa és tulajdonságai alapján fogalmaznak meg megkötéseket a G tulajdonsággráfra, és az ezeknek megfelelő, azaz a mintára illeszkedő gráf részletek alapján rendezett n-eseket adhatnak vissza. Jelölje $\mathcal{T}$ a lehetséges n-esek halmazát. A $t \in \mathcal{T}$ rendezett n-es az alábbiakat tartalmazhatja: csúcsok, élek, útvonalak; címkék, típusok, tulajdonságok; elemi értékek: számok, *stringek*, logikai értékek (a lehetséges elemi értékek halmaza: $\mathcal{P}$); az előbbiekből képzett listák vagy asszociatív tömbök. Gráfminták segítségével nemcsak lekérdezések, hanem modellmódosító műveletek is megfogalmazhatóak: csúcsok, élek létrehozása, összekötése, törlése; tulajdonságok beállítása.

Példa 5. *A 2.12. ábrán látható egy gráfmintával megfogalmazott lekérdezés, amely egy adott* r *paraméterhez megkeresi, hogy melyik* Komponens *tartalmazza. Így lehet meghatározni, hogy melyik vágányon, illetve a darun van-e éppen az adott rakomány. Keressük az olyan* Komponens*-eket, amelyek* r*-ből egy vagy több* RAJTA *élen keresztül elérhetőek, de belőlük már nem megy ki* RAJTA *él* Komponens *címkéjű csúcsba (piros szaggatott él). A feltételeknek megfelelő csúcshoz tartozó* id *tulajdonságot szeretnénk megkapni.*



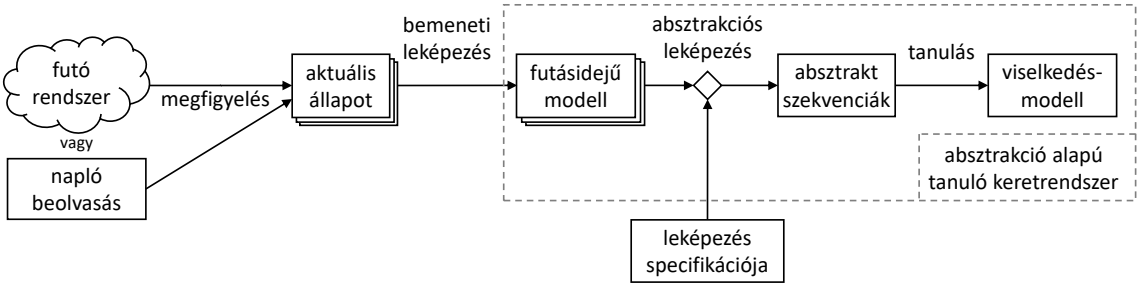
2.12. ábra. Gráfminta példa

3. fejezet

A keretrendszer bemutatása

A megvalósított keretrendszer képes egy futó rendszer megfigyelései alapján annak viselkedésmodelljét elkészíteni, így olyan rendszerekhez is készíthetők formális modellek, amelyek komplexitása, illetve adatfüggő viselkedése ezt korábban megakadályozta.

A keretrendszer felépítése a 3.1. ábrán látható. A rendszer működése során megfigyeljük annak aktuális állapotát, amelyet egy gráf alapú futásidejű modell reprezentál. Ezen a gráfon célzott tanulást hajtunk végre, amelyhez a felhasználó gráfminták segítségével definiálhatja a gráf – tanulás szempontjából – releváns részeit. A releváns részeket tartalmazó absztrakt szekvenciákat már fel lehet használni a tanulás során, hogy előállítsuk a megfigyelt rendszer viselkedésmodelljét. Így lehetővé válik komplex, parametrikus rendszerek egy kiemelt részletének formális vizsgálata.

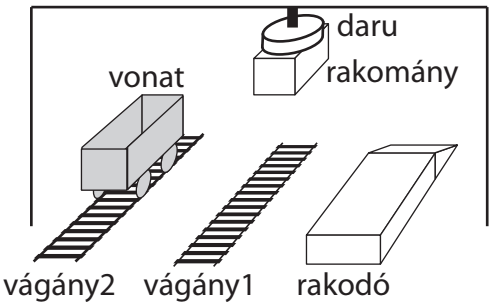


3.1. ábra. Absztrakció alapú tanuló keretrendszer

3.1. Esettanulmány

Esettanulmányként egy vasúti állomást használunk, ahol vonatok érkeznek rakománnyal, amelyet egy daru segítségével tud a kezelő áthelyezni a kocsik között. Működés során különböző szenzorok segítségével figyeljük meg, hogy az egyes komponensek milyen állapotban, milyen helyzetben vannak. Ezeket az információkat egy gráfban tartjuk nyilván. Például ilyen gráf lehet az objektumok és a köztük lévő asszociációk alapján készített modell (például UML objektumdiagram). A működés során megfigyelünk veszélyes helyzeteket, például ha egy rakományt nagy magasságban engedtek el. Ezekben a helyzetekben vészleállást és helyreállítást követően folytatódik a munka.

A megfigyelt információkat rögzítjük, és ezek alapján viselkedésmodellt készítünk. A rendszer lefutásait megkülönböztetjük aszerint, hogy történt-e baleset a lefutás során vagy nem, így el lehet különíteni a rendszer szabályos és szabálytalan viselkedéseit.



3.2. ábra. Vasúti esettanulmány áttekintő ábrája

3.1.1. Példa

A 3.1. táblázatban egy üres állapotból (4.2. ábra) indított példa lefutás látható, ahol egy rakománnyal érkező vonatról egy másik, üres vonatra pakolja át a daru a rakományt, majd mindkettő elhagyja az állomást.

	Változás		Magyarázat
1.	vonat1	rakomány1-et be	A vonat1 a rakomány1-gyel bejött.
2.	vonat2	be	
3.	daru	rakodó→vágány1	A daru vízszintesen a rakodótól a vágány1 fölé ment.
4.	daru	le	A daru függőlegesen a vonatig leereszkedett.
5.	daru	felvesz: rakomány1	A daru felvette a rakományt.
6.	daru	fel	
7.	daru	vágány1→vágány2	
8.	daru	le	
9.	daru	lerak: rakomány1	A vágány2 fölött áll, így a vonat2-re rakta a rakományt.
10.	daru	fel	
11.	vonat2	rakomány1-et ki	
12.	vonat1	ki	
Szabályos lefutás			Nem történt baleset, ezért szabályos a lefutás.

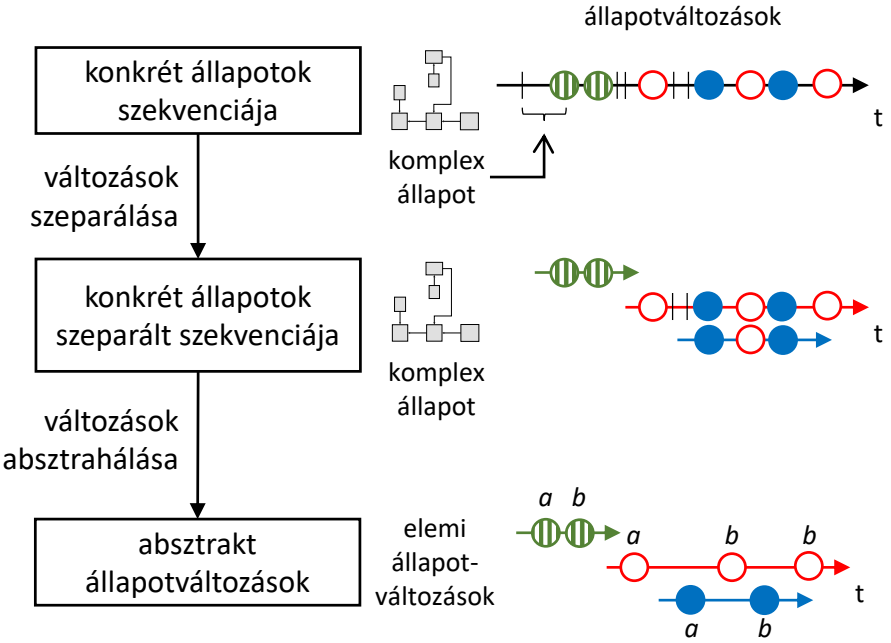
3.1. táblázat. Vasúti állomás egy lefutása

4. fejezet

Absztrakció

A rendszer megfigyelése során keletkező állapotszekvenciák önmagukban nem alkalmasak viselkedésmodell előállítására. Mivel ezek a lefutások túl részletesek, így egyéb, a vizsgálat tárgyát nem képező működéseik is lehetnek. Ezen felül a vizsgálandó működés időben párhuzamosan, több példányban is futhat. Nehézségekbe ütközik továbbá egy komplex gráfmodell esetén az állapotok közti egyenlőség vizsgálata is, amelyre azonban a tanulás során szükség van.

A 4.1. ábrán láthatók az absztrakció lépései. A rendszer állapotán annak sémájának ismeretében gráfmintákat fogalmazunk meg. Első lépésként elkülönítjük a gráf bizonyos elemeit, hogy a párhuzamosan futó, független folyamatokat külön szekvenciákra vágjuk. Ezután a szeparált szekvenciákon végrehajtjuk a mintaillesztéseket, és az illeszkedések változásaiból álló szekvenciákat képzünk belőlük, amelyek már felhasználhatóak a tanuló algoritmus bemeneteként.



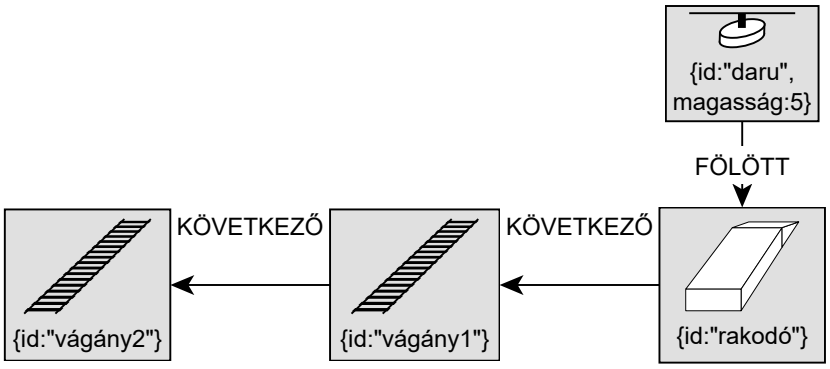
4.1. ábra. Absztrakció lépései

4.1. Megfigyelés

Az állapot megfigyelése során a mintailleszkedések változásait szeretnénk észlelni. Ez úgy lehetséges, ha minden vizsgált állapotra kiértékeljük az összes mintát. Amennyiben tudunk értesülni arról, hogy egy változás történt, akkor közvetlenül az értesítés után futtathatjuk a mintákat, így biztosan minden változást észlelünk. Ha tudjuk, hogy a gráfban milyen változás történt, és a gráfmintaillesztést inkrementálisan végezzük el, akkor az összes minta újbóli lefuttatása helyett csak a változás által érintett mintákat kell (részben) újraszámolni, így csökkenthető a számításigény.

Definíció 23 (Gráflefutás). Egy rendszer megfigyelése során mintavételezzük az aktuális állapotot leíró futásidejű modellt. A mintavételezések így előálló sorozata az $\omega = (G_1, G_2, \ldots, G_l)$ gráflefutás, amelynek minden karaktere egy, az állapotot leíró tulajdonsággráf. A lefutással ellentétben itt az egyes karakterek egy nem feltétlenül véges halmazból kerülnek ki. ▪

Példa 6. *A vasúti esettanulmány egy lehetséges gráflefutása látható a 4.1. táblázat második oszlopában. Az üres rendszerben megjelenik egy vonat egy rakománnyal, amelyet a daru egy másik üres vonatra pakol át, majd mindkettő vonat elhagyja a rendszert. A mintavételezések helyett a táblázatban csak a gráfon történt változásokat tüntettük fel. A kiindulási állapot a 4.2. ábrán látható.*



4.2. ábra. Gráflefutás kiindulási állapota

4.2. Lefutások szétválasztása

A párhuzamosan futó folyamatok elkülönítésére gráfmintát írunk fel. Nevezzük ezt *szeparáló mintának*. A lefutásokat ezen minta illeszkedései mentén vágjuk szét. Egy új illeszkedés megjelenésekor új szekvencia kezdődik, amely az illeszkedés megszűnéséig tart. A konkrét illeszkedést a további mintákban felhasználjuk arra, hogy csak az abban a szeparált szekvenciában releváns változásokat vizsgáljuk. Ha egy szekvenciában egymás után előfordul több vizsgálandó szakasz, a szekvenciát a szeparálás segítségével rövidebb szakaszokra bontjuk. Ez a lépés a tanulás hatékonyságát is javítja, mivel várhatóan rövidebb, nagyobb részben egyező szekvenciákat kapunk, így ezek összevonását a tanulás korai fázisában megtehetjük.

Definíció 24 (Szeparáló minta). Szeparáló mintának hívjuk a $q_s : \{G_1, G_2, \ldots, G_l\} \to 2^{\mathcal{T}_s}$ gráfmintát, amely a gráflefutást a vizsgálandó folyamat szerint eltérő $t \in \mathcal{T}_s$ illeszkedések mentén külön szekvenciákra bontja. $T_s \subseteq T$ jelöli a szeparáló minta tényleges illeszkedéseinek halmazát. ▪

Definíció 25 (Szeparált gráflefutás). $\langle \omega_s, t \rangle \in S(q_s, \omega)$ jelöli az ω gráflefutás q_s szeparáló minta szerinti szeparált lefutásból (ω_s) és a q_s minta hozzá tartozó illeszkedéséből (t) álló párok halmazát, ahol $\omega_s = (G_{k+1}, G_{k+2}, \ldots, G_{k+m})$ szeparált lefutás az $\omega = (G_1, G_2, \ldots, G_l)$ lefutás egy folytonos része és $\forall G_i \in \omega_s : t \in q_s(G_i)$, de $t \notin q_s(G_k), t \notin q_s(G_{k+m+1})$, ahol $G_k, G_{k+m+1} \in \omega$, azaz nem létezik ω_s-et tartalmazó bővebb folytonos lefutás, amelynek minden elemére q_s minta illeszkedik t illeszkedéssel. ▪

Példa 7. *A vasúti esettanulmányban a rakományok lehetséges mozgását vizsgáljuk. Ahhoz, hogy az egy időben előforduló rakományokat megkülönböztessük, a lefutásokat rakomány szerint szétválasztjuk. A 4.3. ábrán látható minta illeszkedései a rendszerben aktuálisan előforduló rakományok (*Rakomány *címkéjű csúcsok). Egy illeszkedés megjelenésekor új szeparált szekvencia kezdődik, amely annak megszűnéséig tart.*

4.3. ábra. Szeparáló minta

4.3. Változások absztrakciója

Következő lépésként a komplex állapotokat kell a tanulás során is felhasználható, könnyen összehasonlítható karakterekké alakítani, miközben a nem releváns információkat is ki kell szűrni. Ehhez a szűréshez megfogalmazhatunk gráfmintákat, amelyek a gráf vizsgálatunk szempontjából releváns részeire fogalmaz meg megkötéseket. Ezt a lépést már a szeparált lefutásokon vizsgáljuk, így a minta felhasználhatja a szeparáló minta lekötéseit, így szeparált lefutásonként más-más releváns részeket tud vizsgálni.

Az általános gráfmintákkal szemben, amelyek csúcsokat vagy éleket is visszaadhatnak, ebben a lépésben olyan mintákat alkalmazunk, amelyek a csúcsok és élek címkéit, illetve típusát vagy bizonyos tulajdonságait adják vissza, azaz már elemi típusokat (egész, logikai, *string*). Erre azért van szükség, mert az általános gráfminták rendezett n-esek *multihalmazát* adják vissza (azaz egy elem előfordulásainak számát is tároljuk), amelyek között az összehasonlítás körülményesebb. Továbbá a minták eredményében a csúcsokat és éleket azok belső azonosítói reprezentálják, amelyek futásról futásra változnak, így eltérő lefutásokat eredményeznének, illetve nem hordoznak a gráf szempontjából értékelhető információt. Az összehasonlíthatóságra azért van szükség, mert a tanulás lépés során a lefutásokban szereplő karakterek egyezése és különbözősége befolyásolja a lehetséges állapot-összevonásokat. Ennélfogva az absztrakciós minták csak egy elemi típusú értékkel vagy az illeszkedés hiányával térhetnek vissza. Ez nem csökkenti a minták kifejezőerejét, mivel aggregációs és *string*-műveletek segítségével a szükséges típusú értékek előállíthatóak.

Továbbá szükséges, hogy az absztrakciós minta a lehetséges végtelen sok gráfot elemi értékek véges tartományára képezze le. A mintavételezés véges sok lefutást rögzít, majd a tanulás során ebből véges automatát állítunk elő. Amennyiben végtelen sok lehetséges érték szerepelhetne a lefutásokban, a megtanult automata biztosan hibás lenne, mivel végtelenül sok vizsgálandó, azaz az absztrakció során nem absztrahált viselkedés hiányozna belőle. Tehát szükséges, hogy a gráf tetszőlegesen bonyolult struktúráját, illetve intervallumabsztrakció felhasználásával a számértékeket a vizsgálat szempontjából szükséges mértékben, de mindenképpen véges tartományra absztraháljuk az absztrakciós minta felhasználásával.

Definíció 26 (Absztrakciós minta). Absztrakciós mintának hívjuk a $q_a : \{G_1, G_2, \ldots, G_l\} \times T_s \to \mathcal{P}_f \subset \mathcal{P} \cup \{\varepsilon\}$ gráfmintát, amely egy gráflefutásbeli gráfon a szeparáló minta illeszkedése szerint végez mintaillesztés, amelynek eredménye az illeszkedés elemi értékek egy véges részhalmazára való leképzése vagy az illeszkedés hiányát jelző ε szimbólum. ▪

Definíció 27 (Illeszkedések lefutása). Illeszkedések lefutásának hívjuk az $\mathrm{M}(\langle q_a^1, q_a^2, \ldots, q_a^n\rangle, \omega_s, t) = \omega_m = (\langle p_1^1, p_1^2, \ldots, p_1^n\rangle; \ldots; \langle p_l^1, p_l^2, \ldots, p_l^n\rangle)$ lefutást, ahol $\omega_s = (G_1, G_2, \ldots, G_l)$ szeparált gráflefutás minden G_i elemére végrehajtva a q_a^j absztrakciós mintákat a $p_i^j = q_a^j(G_i, t) \in \mathcal{P}_f$ elemi értéket kapjuk. ▪

A mintavételezés során előfordulhat, hogy egy állapotot több alkalommal is rögzítünk. Azonban a mintavételezés számától a megalkotott viselkedésmodellnek függetlennek kell lennie, amennyiben minden változást rögzítettünk. Ebből következően nem a minták adott állapotbeli illeszkedéseit kell felhasználni a lefutásokban, hanem ezen illeszkedések megváltozását, és ezen változásokból alkotni a tanulás során felhasználandó karaktereket.

Definíció 28 (Absztrakt lefutás). Absztrakt lefutásnak hívjuk az $\mathrm{A}(\langle q_a^1, q_a^2, \ldots, q_a^n\rangle, \omega_s, t) = \omega_a$ lefutást. $C = (\langle c_1^1, c_1^2, \ldots, c_1^n\rangle; \ldots; \langle c_l^1, c_l^2, \ldots, c_l^n\rangle)$ a $p_i^j \in \mathrm{M}(\langle q_a^1, q_a^2, \ldots, q_a^n\rangle, \omega_s, t)$ elemi értékek változásaiból képzett lefutás, ahol $(p_0^j = \varepsilon)$, továbbá minden $i \in [1, l]$ és $j \in [1, n]$ esetén

$$c_i^j = \begin{cases} \langle p_{i-1}^j, p_i^j\rangle, & \text{ha } p_{i-1}^j \neq p_i^j \\ 0, & \text{egyébként} \end{cases}$$

▪

Ezen felül $\omega_a = C \setminus \{\langle 0, \ldots, 0\rangle\}$, azaz a lefutásból kihagyjuk azokat a karaktereket, amelyekben semmilyen változás nem szerepel.

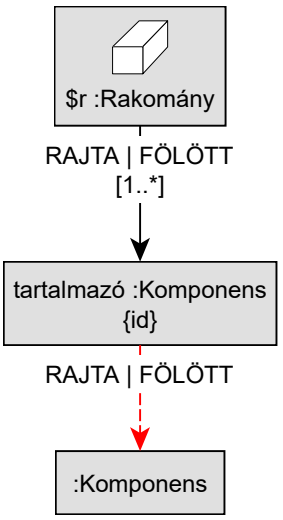
Példa 8. *Vizsgáljuk, hogy a rakomány a pálya mely elemén van. Felhasználhatjuk a szeparáló mintában az ehhez a lefutáshoz tartozó* r *rakományt paraméterként. A 4.4a. ábrán látható minta megállapítja, hogy az* $r *paraméterként kapott rakomány milyen komponensen vagy milyen komponens felett van. Azaz melyik az a* Komponens*, amely elérhető egy vagy több* RAJTA *vagy* FÖLÖTT *élen keresztül, de belőle már nem megy ki* RAJTA *vagy* FÖLÖTT *él egy* Komponens *címkéjű csúcsba. Az adott* tartalmazó *komponens* id *tulajdonságát adja vissza. További absztrakciós lépésként megtehető, hogy az* id *tulajdonság helyett az* l_v(tartalmazó) *címkehalmazt használjuk fel, ha a vizsgálat során elhanyagoljuk, hogy pontosan melyik vágányon van a rakomány, elég csak az, hogy valamelyik vágányon vagy a rakodón van.*

A 4.4b. ábrán szereplő minta azt vizsgálja, hogy az r *rakomány a darun van-e, azaz kapcsolódik-e* Daru *címkéjű csúcshoz* RAJTA *élen keresztül.*

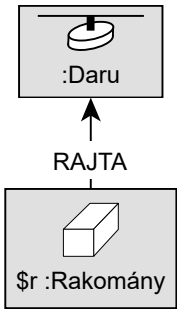
Emellett az r *rakomány függőleges helyzetét is szeretnénk vizsgálni. Ehhez felírtuk a 4.4c. ábrán szereplő mintát, amely csak akkor illeszkedik a rakományra, ha annak a* magasság *tulajdonsága 3 méternél nem kisebb. (A szűrő feltétel dőlt betűvel szerepel.)*

4.4. Lefutások osztályozása

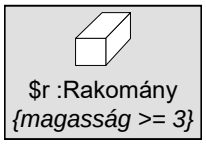
A tanulás során osztályozott, *elfogadó* és *nem elfogadó* lefutásokra van szükség. Amennyiben a rendszer állapotában, az egyes szeparált szekvenciákra vonatkozóan rendelkezésre áll, hogy elfogadó vagy nem elfogadó-e a lefutás, akkor egy gráfminta segítségével az osztályozás automatikusan elvégezhető. *Pozitív feltétel* felírása esetén a



(a) Rakomány vízszintes helyzete



(b) Kapcsolódó daru keresése



(c) Rakomány függőleges helyzete

4.4. ábra. Absztrakciós gráfminták

lefutás elfogadó, ha volt a mintának illeszkedése a lefutás során, egyébként nem elfogadó. *Negatív feltétel* esetén nem elfogadó, ha volt illeszkedés, egyébként elfogadó.

Definíció 29 (Osztályozó minta).
Pozitív osztályozó mintának hívjuk a $q_c^+ : \{G_1, G_2, \ldots, G_l\} \times T_s \to \{0, +\}$ gráfmintát. Negatív osztályozó mintának hívjuk a $q_c^- : \{G_1, G_2, \ldots, G_l\} \times T_s \to \{0, -\}$ gráfmintát. Az osztályozó minta egy gráflefutásbeli gráfon a szeparáló minta illeszkedése szerint végez mintaillesztést, amelynek eredménye

$$\begin{aligned} &+, \text{ ha a gráf alapján a lefutás elfogadó,} \\ &-, \text{ ha a gráf alapján a lefutás elutasító,} \\ &0, \text{ egyébként.} \end{aligned}$$

A q_c osztályozó minta eredményét az ω_s szeparált szekvencián a $t_s \in T_s$ illeszkedés mellett így definiáljuk:

$$C(q_c^+, \omega_s, t_s) = \begin{cases} +, & \text{ha } \exists G_i \in \omega_s : q_c^+(G_i, t_s) = + \\ -, & \text{egyébként} \end{cases}$$

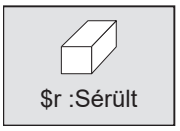
$$C(q_c^-, \omega_s, t_s) = \begin{cases} -, & \text{ha } \exists G_i \in \omega_s : q_c^-(G_i, t_s) = - \\ +, & \text{egyébként} \end{cases}$$

.

Definíció 30 (Absztrakt osztályozott lefutás). Absztrakt osztályozott lefutásnak hívjuk az $\langle \omega_a, c \rangle \in \mathrm{AC}(q_s, \langle q_a^1, q_a^2, \ldots, q_a^n \rangle, q_c, \omega), c \in \{+, -\}$ lefutás-elfogadás párt.
$\langle \omega_s, t \rangle \in S(q_s, \omega)$ a szeparált lefutások és a hozzájuk tartozó illeszkedések.

$\mathrm{A}(\langle q_a^1, q_a^2, \ldots, q_a^n \rangle, \omega_s, t) = \omega_a$ a szeparált lefutásokból képzett absztrakt lefutás.
$C(q_c, \omega_s, t_s) = c$ az egyes szeparált lefutásokhoz tartozó elfogadás értéke. ▪

Példa 9. *A szeparált lefutásokat osztályozzuk aszerint, hogy történt-e baleset a szeparált lefutáshoz tartozó rakománnyal. Az ilyen esetek a gráflefutásokban jelölve vannak. Negatív feltétel felírásával fogalmazható meg az osztályozás. A 4.5. ábrán látható minta, akkor illeszkedik a szeparált lefutás* $r *rakományára, ha a csúcson szerepel* Sérült *címke. Ekkor nem elfogadó a lefutás. Ha a lefutás során nem volt illeszkedése a negatív osztályozó mintának, akkor a lefutás elfogadó.*



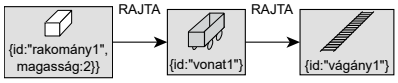
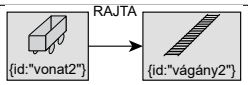
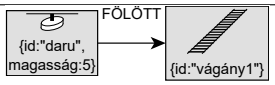
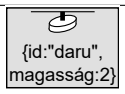
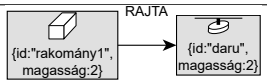
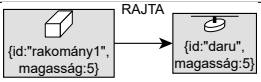
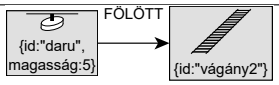
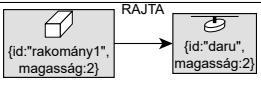
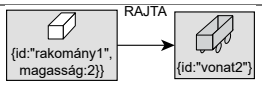
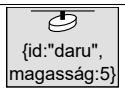
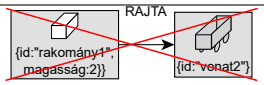
4.5. ábra. Negatív osztályozó minta

Példa 10. *A 4.1. táblázatban láthatók az absztrakciós minták illeszkedései az esettanulmányban szereplő példán. A kiindulási állapot a 4.2. ábrán látható. Az egyes sorokban a gráfon történt változásokat is ábrázoltuk. Az első sorban megjelenik a* rakomány1*, ami illeszkedik a szeparáló mintára (4.3. ábra), így egy szeparált szekvencia kezdődik a* rakomány1 *illeszkedő csúccsal. Az absztrakciós minták illeszkedésének változásai láthatóak a három jobb oldali oszlopban. A szeparált szekvencia véget ér, amint eltűnik a* rakomány1*. Mivel nem történt baleset a futás során, nem volt illeszkedése a negatív osztályozó mintának (4.5. ábra), így a szekvencia elfogadó.*

4.5. Absztrakció hatása a tanulásra

Általános esetben a következő okok miatt van szükség absztrakcióra, ha komplex információkat automatatanulás során szeretnénk felhasználni:

- **Nagy állapottér.** Aktív automatatanulás esetén a nagy állapottér azt eredményezheti, hogy az algoritmus az állapotok feltérképezése során túllépi a rendelkezésre álló méret- vagy időkorlátokat. Jelen esetben passzív tanulás során ez úgy jelentkezik, hogy a rögzített lefutások hosszúak, és a tanulás során kell ezeket összevonni, ahol lehet. Ezt a problémát a megközelítésünk a szeparáló minták bevezetésével segíti.
- **Nagy vagy végtelen ábécé.** Aktív tanulás esetén fel kell térképezni az összes állapotban, hogy az egyes karakterekre milyen állapotba kerül az automata. Ez ütközhet erőforráskorlátokba. Azonban ha valós számok szerepelnek paraméterként, akkor elvi akadálya van a tanulásnak, mivel nem lehetséges az összes karakterre feltérképezni a viselkedést. A megközelítésünkben intervallumabsztrakciót alkalmazunk az absztrakciós mintákban, hogy a valós paramétereket is kezelni tudjuk.
- **Áttekinthetetlen méretű automata.** Sikeres tanulás során is lehetséges, hogy az eredmény nem jól felhasználható például dokumentációs célra, mivel túl sok, nem releváns részletet tartalmaz, vagy hasonló viselkedések többször megjelennek. A megközelítésünkben az absztrakció definiálása ad lehetőséget a fókuszált tanulásra.

Változás		Változás a gráfban	4.4a. minta	4.4b. minta	4.4c. minta
vonat1	rakomány1-et be		szeparált lefutás kezdete		
			ε → vágány1		
vonat2	be				
daru	rakodó→vágány1				
daru	le				
daru	felvesz: rakomány1			megjelenik	
daru	fel				megjelenik
daru	vágány1→vágány2		vágány1 → vágány2		
daru	le				eltűnik
daru	lerak: rakomány1			eltűnik	
daru	fel				
vonat2	rakomány1-et ki		szeparált lefutás vége		
vonat1	ki				

4.1. táblázat. Absztrakciós minták illeszkedése a vasúti példán

Az általunk vizsgált gráffal leírható állapotok esetén feltétlenül szükség van valamiféle absztrakcióra, egyrészt lehetséges gráfok végtelen száma miatt, továbbá mivel a tanulás során szükséges az állapotátmeneteken lévő karakterek egyenlőségvizsgálata. Az egyenlőségvizsgálat gráfok esetében a gráfizomorfia-vizsgálat, amelyre nem ismert polinomiális algoritmus.

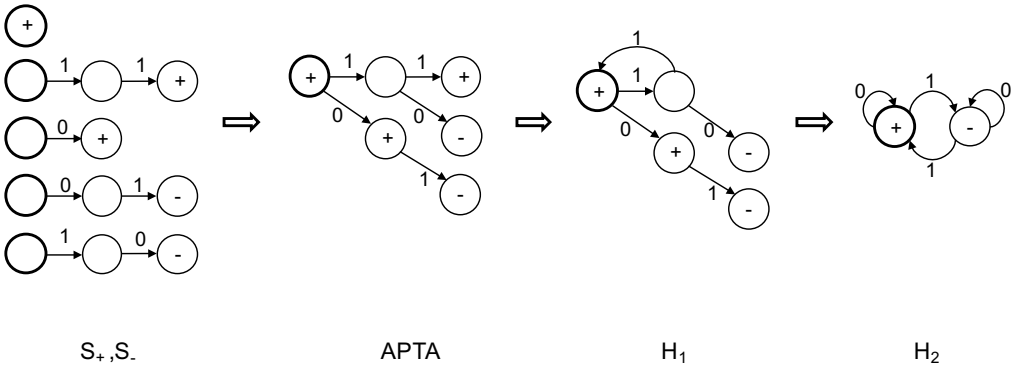
A tanulás során felhasznált lefutásokra vonatkozik az a követelmény, hogy determinisztikusak legyenek, azaz egy lefutás kizárólag elfogadó vagy elutasító legyen. Az absztrakció során információt hagyunk el, amely nemdeterminisztikus lefutásokat okozhat. Amennyiben a rendszer rögzített lefutásai determinisztikusak voltak, akkor a rendelkezésre álló információk felhasználására további absztrakciós mintákat kell definiálni.

5. fejezet

Automatatanuló algoritmus

Ebben a fejezetben bemutatom a használt tanulóalgoritmus működését, amelyet a szerzőtársammal közös TDK-dolgozatunkban már ismertettünk [8]. A dolgozathoz képest eltérés, hogy jelen szakdolgozat kizárólag időzítést nem tartalmazó rendszerek kezelésére szorítkozik, így ebben a fejezetben csak az időzítés nélküli automatatanulást mutatom be.

Munkám során egy pozitív és negatív elemeket is felhasználó, passzív, offline tanulóalgoritmust használtam, amely véges automatát képes előállítani, amelynek minden állapota vagy elfogadó, vagy elutasító. Ahogy arról már a 2.3.3. fejezetben szó volt, a passzív tanulás már meglévő lefutások halmazával dolgozik, azokból először egy APTA-t (21. definíció) készítve, majd azt összevonva azt egy véges állapotgéppé (7. definíció). A 5.1. ábrán látható egy áttekintő folyamatábra a 3. példában leírt rendszernek a megtanulásáról. Ebben a példában a cél az volt, hogy a páros számú 1 karaktert tartalmazó szavakat fogadja el a megtanult automata.



5.1. ábra. Passzív tanulás folyamata

5.1. APTA előállítása

A tanulási folyamat első lépése tehát az APTA előállítása, vagyis a lefutások összevonása egy fává. Az előállítás során feltételezzük azt, hogy a lefutásaink nem ellentmondásosak, vagyis nem létezhet két olyan lefutás, melyekben a bemeneti események és azok sorrendje azonos, azonban az egyik elfogadó, míg a másik nem elfogadó állapotba vinné az automatát.

Az APTA előállításának a menetét a 3. pszeudokód szemlélteti. A folyamat első lépéseként az összes lefutás kezdőállapotát összevonjuk, ez lesz a fa gyökere. Ezek után a gyökértől távolodva, ameddig találunk olyan állapotokat, melyek a gyökértől egyenlő távolságra vannak és onnan megegyező karaktersorozatra érhetőek el, összevonjuk. Ha

nincsenek már ilyen állapotok, készen vagyunk. A pszeudokódban $s_{i,j}$ az i-edik lefutás j-edik állapotát jelöli, a az ábécé egy karakterét, q_i pedig az APTA i-edik állapotát.

Algoritmus 3: APTA előállítása: *apta*

Input: $S = \{S_+, S_-\}$ pozitív és negatív lefutások halmaza (összesen k db)
Output: H lefutási fa, amely tartalmazza S összes lefutását

```
1  q_0 = osszevon(sum_{i=0}^{k} s_{i,0})
2  while ∃q_i, s_{a,j}, s_{b,j}, ..., s_{n,j} amelyre s_{a,j} = δ(q_i, a), s_{b,j} = δ(q_i, a), ..., s_{n,j} = δ(q_i, a)
   do
3  |  osszevon(s_{a,j}, s_{b,j}, ..., s_{n,j})
4  end
5  return H
```

Az állapotok összevonása (4. algoritmus) a következőképpen zajlik: létrehozzuk az új állapotot, belemásoljuk a régi állapotok bemenő és kimenő éleit (ha voltak), majd beállítjuk a jellemzőjét. Itt szintén feltételezzük azt, hogy a lefutásainkban nincs ellentmondás. Ezt követően töröljük a már összevont állapotokat.

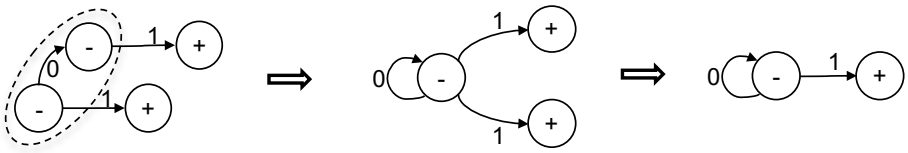
Algoritmus 4: Állapotok összevonása: *osszevon*

Input: $s_1, s_2, \ldots s_n$ összevonandó állapotok
Output: q állapot

```
1  q állapot létrehozása
2  for i ∈ s_1, s_2, ..., s_n do
3  |  if ∃δ(r, a) = s_i then
4  |  |  δ(r, a) = q állapotátmenet létrehozása
5  |  end
6  |  if ∃δ(s_i, a) = r then
7  |  |  δ(q, a) = r állapotátmenet létrehozása
8  |  end
9  |  if s_i elfogadó then
10 |  |  q beállítása elfogadóra
11 |  end
12 |  if s_i nem elfogadó then
13 |  |  q beállítása nem elfogadóra
14 |  end
15 |  s_i állapot és állapotátmenetei törlése
16 end
17 return q
```

5.2. Merge

A passzív tanuló algoritmus következő lépése, hogy az APTA-nak, ameddig csak tudja, csökkentse a méretét, illetve általánosítsa azt. Ezt úgynevezett *merge* műveletekkel tudja megtenni. Ez a művelet két lépésből áll, először a kijelölt két állapotot összevonja, majd az így okozott inkonzisztenciákat, nemdeterminizmusokat feloldja. Egy *merge* csak akkor végrehajtható, hogyha sem az összevonás, sem az azt követő inkonzisztencia feloldás során nem vonunk össze ellentétes állapotokat. Vagyis a művelet során egy elfogadó és egy nem elfogadó állapot összevonása nem megengedett.



5.2. ábra. A *merge* műveletet

A 5.2. ábrán szemléltetünk egy *merge* műveletet, illetve a 5. pszeudokódban leírjuk annak működését.

Algoritmus 5: Merge DFA tanulás során: *merge*

Input: H DFA két állapota: q q'
Output: *igaz* ha a merge lehetséges, *hamis* egyébként

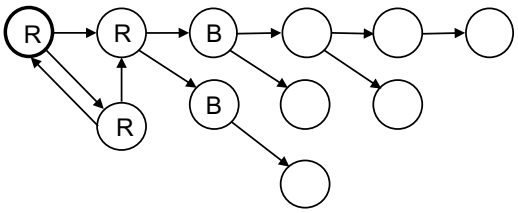
```
1  if q elfogadó és q' nem elfogadó, vagy fordítva then
2  |  return hamis
3  end
4  osszevon(q, q')
5  while H automatának van nemdeterminisztikus átmenete q_n és q'_n állapotokba do
6  |  boolean b = merge(q_n, q'_n)
7  |  if b hamis then
8  |  |  eddigi merge műveletek visszavonása
9  |  |  return hamis
10 |  end
11 end
12 return igaz
```

5.3. Piros-kék algoritmus

Annak érdekében, hogy a tanulás gyorsabb legyen, a piros-kék eljárást [19] használjuk. Ez azáltal gyorsítja a passzív automatatanulást, hogy *merge* esetén nem vizsgálja meg az összes lehetséges q és q' párost, nem próbál meg minden állapotpárt összevonni, hanem a fa gyökere felől indulva, mindig az ahhoz közelebb eső új csúcsokat próbálja összevonni a gyökérhez közelebb esőkkel. Ennek a megvalósításához a csúcsokat két színnel színezi: pirossal jelöli azokat az állapotokat, amelyek egymással már biztosan nem összevonhatók, kékkel pedig a következő jelölteket. Futás során csak az általa kékre színezett állapotokat vizsgálja meg és párosítja az összes pirosra színezett állapottal. Ezeket a színezéseket pedig a következőképpen kapjuk meg.

Az algoritmus kezdetben az automata kezdőállapotát pirosra színezi, majd ennek az állapotnak a közvetlen gyerekeit kékre. Innentől kezdve minden lépést követően a új piros állapotok közvetlen gyerekei kékek lesznek. Csak kék állapot színeződhet át pirosra – vagy azért, mert végrehajtunk egy *merge* műveletet és ilyenkor az összevont állapot piros lesz, vagy mert nincs megengedett *merge*, így a kék állapotot át kell színeznünk pirosra.

A 5.3. ábra szemlélteti az algoritmust. A pirosra színezett állapotokat R jelöli, a kékeket B. Tulajdonképpen ez a színezés azt jelenti, hogy a piros állapotokat már nem fogja módosítani az algoritmus, azok a végső automatában is benne lesznek. A kék állapotok azok, melyeket az algoritmus ebben a körben vizsgál, a nem színezettekkel pedig csak a későbbiekben fog foglalkozni. Ebből az is következik, hogy az automatatanulás akkor fejeződik be, ha az automata minden állapota piros.



5.3. ábra. A piros-kék algoritmus

5.4. Color

Az előző fejezetben megemlítettünk az átszínezés műveletét. Ezt akkor hajtjuk végre, ha nincsen olyan kék-piros állapotpár, melyet össze lehetne vonni. Ilyenkor egy kék állapotot egyszerűen pirosra színezünk. Ennek a műveletnek a neve *color*.

5.5. Metrika alapú állapot-összevonás

Az automatatanuló algoritmustól azt is elvárjuk, hogy az általa megtanult automatának az állapottere minél kisebb legyen. Erre a metrika alapú állapot-összevonás (*Evidence Driven State Merging* – EDSM) [11, 4] eljárás megoldást ad. Ennek az eljárásnak, a nevéből adódóan is az a lényege, hogy a lehetséges *merge* műveletekhez valamilyen metrikát rendel, majd azt a műveletet fogja végrehajtani, ami a legnagyobb metrikával rendelkezik. Ezt a metrikát pedig az alapján számolja, hogy milyen jónak, hasznosnak, biztosnak ítéli meg a *merge* műveletet.

Az algoritmus működése a következőképpen alakul: megnézi, hogy egy $merge(q, q')$ során mely állapotpárok lesznek majd összevonva. A metrikát eleinte $m = 0$ értékkel inicializálja, majd a q_i, q'_i állapotpárok alapján a következőképpen módosítja azt:

- ha q_i elfogadó és q'_i nem elfogadó, vagy fordítva, akkor $m = -\infty$
- ha q_i és q'_i is elfogadó, vagy mindkettő nem elfogadó, akkor m értékét növeli 1-gyel
- q_i vagy q'_i nem ismert, hogy elfogadó-e, akkor nem változtat m értékén.

Az így kapott legmagasabb értékű *merge* műveletet fogja végrehajtani az algoritmus. Ha a legmagasabb metrika a $-\infty$ lenne (vagyis nincs lehetséges $merge$), akkor *color* műveletet hajt végre. Ezzel tulajdonképpen 0 metrikát, értéket tulajdonítva a *color* műveletnek.

5.6. Az algoritmus

A fent leírt műveletek és eljárások alapján az általunk megvalósított passzív automatatanuló algoritmus a 6. pszeudokód alapján működik.

Algoritmus 6: DFA tanulóalgoritmus

Input: $S = \{S_+, S_-\}$ pozitív és negatív lefutások halmaza
Output: H kis állapotterű DFA, melyre igazak S elemei

```
1  H = apta(S)
2  q_0, a kezdőállapot pirosra színezése
3  q_0 közvetlen gyerekeinek kékre színezése
4  while H automatában van még nem piros állapot do
5  |  Lehetséges merge műveletek metrikájának a kiszámítása
6  |  if merge(r, b) rendelkezik a legmagasabb pozitív metrikával then
7  |  |  merge(r, b)
8  |  else
9  |  |  color()
10 |  end
11 |  Új piros állapotok gyerekeinek kékre színezése
12 end
13 return H
```

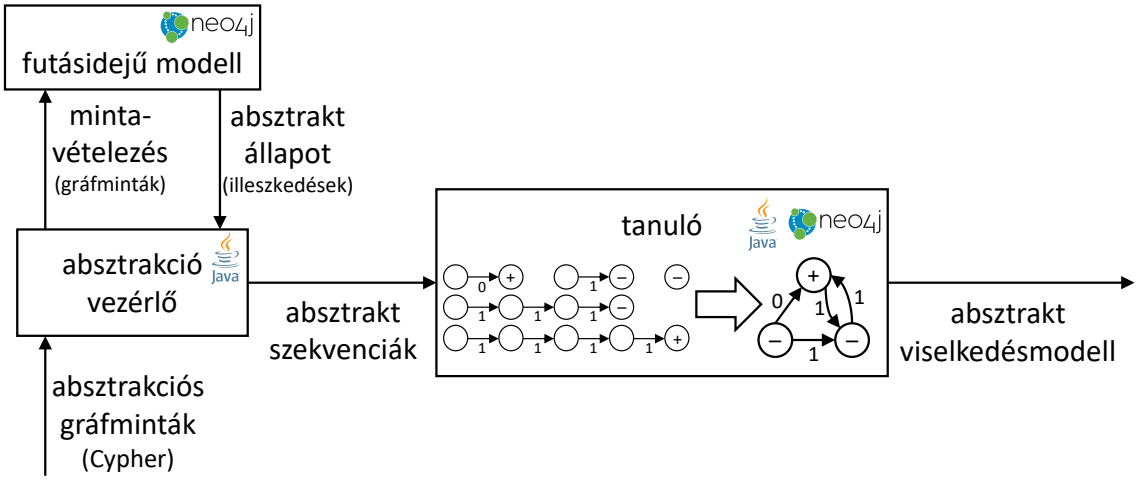
6. fejezet

Megvalósítás

Ebben a fejezetben bemutatom a keretrendszer megvalósítása során felhasznált technológiákat, a megvalósított komponenseket, illetve a fontosabb implementációs részleteket.

6.1. Architektúra

A megvalósított keretrendszer két fő részből, egy absztrakciós és egy tanuló komponensből áll. A keretrendszer felépítése a 6.1. ábrán látható.



6.1. ábra. A keretrendszer felépítése

Az absztrakciós komponens a gráf formájában rendelkezésre álló futásidejű modell mintavételezésével képez absztrakt szekvenciákat. Ehhez a felhasználó a célzott tanulást vezérlő gráfmintákat definiál, amelyeket aztán a rendszer mintavételezett állapotain gráfmintaillesztéssel lehet detektálni. A kimeneti szekvenciákat alkotó absztrakt állapotokat a minták illeszkedései határozzák meg. Erre a célra Neo4j gráfadatbázist alkalmazunk, amelyhez az absztrakciós gráfmintákat a Neo4j Cypher nyelvén lehet megfogalmazni.

Az absztrakt szekvenciák alapján a tanuló komponens képes absztrakt viselkedésmodellt alkotni. A tanulás során használt algoritmus gráf alapú modellen végez átalakításokat, amelyek hatékonyan fogalmazhatóak meg Cypher nyelv segítségével, így ennél a lépésnél is a Neo4j gráfadatbázist alkalmazzuk. A komponens a bemenetként kapott absztrakt szekvenciákat a gráfadatbázisba tölti, ezen lekérdezések segítségével módosításokat hajt végre, majd az előálló viselkedésmodellt visszatölti az adatbázisból.

A Neo4j [15] egy népszerű NoSQL tulajdonsággráf-adatbázis, amelyben a Cypher [14] lekérdező nyelven lehet gráfmintákat leírni. A Cypher egy magas szintű deklaratív lekérdező nyelv, amelyben ASCII art-hoz hasonló szintaxissal lehet gráfokat leírni.

6.2. Absztrakciós komponens

A futásidejű modell ábrázolására tulajdonsággráfot használunk, amely lehetővé teszi, hogy a vizsgált rendszer állapotának jellemzőit magában a modellben tároljuk. A Cypher nyelv segítségével komplex gráfminták is leírhatók, valamint lehetőség van paraméteres lekérdezések megfogalmazására is, így hozzájárul a lefutásokon értelmezett absztrakciók hatékony végrehajtásához.

Az absztrakció során a rendszer futásidejű modelljét mintavételezzük. Ehhez rendelkezésünkre áll egy értesítés, amely minden modellfrissítés után jelez, ekkor újra kell futtatni a gráfmintákat.

A lefutások szétválasztására a szeparáló minta minden frissítéskor lefut. Az eredményül kapott illeszkedéshalmazt összehasonlítjuk az előző frissítéskor rögzítettekkel. Az összehasonlítás alapján egy illeszkedés lehet:

- **Újonnan megjelenő.** Ebben az esetben egy új szekvenciát kezdünk.
- **Eltűnő.** Ekkor a szekvencia véget ér, beállítjuk az osztályozó mintának megfelelően, hogy elfogadó-e a lefutás.
- **Meglévő.** Ebben az esetben az illeszkedésnek megfelelő paraméterrel lefuttatjuk az absztrakciós mintákat, és ha az illeszkedések változtak, rögzítjük a szekvenciában.

Példa 11. *A 6.1. kódrészlet bemutat egy szeparáló mintát. A vasúti esettanulmányon az egyes rakományok mozgásának vizsgálatához elkülönítjük a lefutásokat rakományok szerint. A* MATCH *kulcsszóval bevezetett minta egy olyan csúcsot keres, amelynek van* Rakomany *címkéje. A csúcsokat kerek zárójelek jelölik. A minta a feltételeknek megfelelő csúcsokat adja vissza. A további mintákban az egyes szeparált lefutásoknak megfelelő rakomány csúcsokra* r *néven lehet hivatkozni.*

```
1 MATCH (r:Rakomany)
2 RETURN r
```

6.1. kódrészlet. A 4.3. ábrának megfelelő szeparáló minta.

A változások absztrahálása során az absztrakciós mintákat futtatjuk. Minden szeparált szekvencia esetén a szétválasztás során rögzített illeszkedést mint paramétert használhatjuk fel a mintában. A minták több egymás utáni frissítés során is illeszkedhetnek, így az illeszkedések megjelenését, eltűnését vagy változását rögzítjük. A lekérdezésektől elvárt, hogy egy megadott paraméterben adják vissza az eredményüket, továbbá az összehasonlíthatóság végett az eredményekben ne szerepeljenek csúcsok vagy élek, hanem csak elemi típusok, tulajdonságok, címkék, éltípusok. Továbbá elvárás, hogy 0 vagy 1 eredménye legyen, mivel így az illeszkedések összehasonlítása egyértelmű. Ez nem jelent megkötést a mintát illetően, mivel a Cypher nyelv rendelkezik aggregációs műveletekkel, ezek segítségével több illeszkedés összevonható egy eredménnyé.

Példa 12. *A 6.2. kódrészletben vizsgáljuk, hogy a rakomány a pálya mely elemén van. Felhasználjuk a szeparáló mintában az ehhez a szeparált lefutáshoz tartozó* r *rakományt paraméterként. A* WITH *kulcsszó miatt az* $r *paramétert* rakomany *néven használhatjuk fel a minta további részében. A --> karaktersorozat irányított élet jelöl. Az élre vonatkozó megkötéseket szögletes zárójelek között adhatjuk meg. A második sorban lekérdezzük a*

rakományból RAJTA *vagy* FOLOTT *típusú élek legalább egy hosszú élsorozatán át elérhető csúcsokat, és ezeket* tartalmazo*-nak nevezzük el. A harmadik sorban ezek közül eltávolítjuk azokat, amelyekből kiindul* RAJTA *vagy* FOLOTT *típusú él tetszőleges csúcsba. A* tartalmazo *csúcs* id *nevű tulajdonságával tér vissza a minta. Az absztrakciós mintára vonatkozó megkötések szerint csak egy vagy nulla illeszkedése lehet, elemi értékkel kell visszatérnie, amelyet egy adott néven (*eredmeny*) lehet elérni.*

A 6.3. kódrészletben azt vizsgáljuk, hogy az $r *paraméterben megkapott rakomány* RAJTA *típusú élen át össze van-e kötve* Daru *címkéjű csúccsal. Illeszkedés esetén a* darun string *jelenik meg eredményként.*

A 6.4. kódrészletben leírt minta akkor illeszkedik, ha az $r *paraméterben megkapott rakomány* magassag *tulajdonsága 3-nál nagyobb vagy egyenlő. Így vizsgálható a rakomány függőleges helyzete.*

```
1 WITH $r AS rakomany
2 MATCH (rakomany)-[:RAJTA|FOLOTT*]->(tartalmazo)
3 WHERE NOT (tartalmazo)-[:RAJTA|FOLOTT]->()
4 RETURN tartalmazo.id AS eredmeny
```

6.2. kódrészlet. A 4.4a. ábrának megfelelő absztrakciós minta.

```
1 WITH $r AS rakomany
2 MATCH (rakomany)-[:RAJTA]->(:Daru)
3 RETURN 'darun' AS eredmeny
```

6.3. kódrészlet. A 4.4b. ábrának megfelelő absztrakciós minta.

```
1 WITH $r AS rakomany
2 WHERE rakomany.magassag >= 3
3 RETURN 'fent' AS eredmeny
```

6.4. kódrészlet. A 4.4c. ábrának megfelelő absztrakciós minta.

A lefutások osztályozása esetén egy olyan lekérdezést kell megadni, amelynek ha van illeszkedése a szekvencia során, akkor nem elfogadó az adott szekvencia, egyébként elfogadó. Vagy lehetőség van ennek a fordítottjára is pozitív feltétel megadásával. A lekérdezés definiálásakor ügyelni kell arra, hogy az adott szekvenciára jellemző feltételt írjunk, azaz a párhuzamos szekvenciák vagy egy a rendszerben előforduló globális tulajdonság ne befolyásolja az osztályzást.

Példa 13. *A szeparált lefutásokat osztályozzuk aszerint, hogy történt-e baleset a szeparált lefutáshoz tartozó rakománnyal. A 6.5. kódrészletben leírt minta első sorában keressük a* Rakomany *címkéjű csúcsokat* rakomany *néven. Ezek közül megtartjuk azokat, amelyek egyeznek a szeparált szekvenciához tartozó* $r *rakomány paraméterrel. (Ez az összehasonlítás az adatbázisban használt belső azonosítók összehasonlítására van visszavezetve.) A szűrés során további megkötés, hogy a rakománynak rendelkeznie kell* Serult *címkével. Így vizsgálható, hogy a szeparált lefutáshoz tartozó rakomány sérült-e a lefutás során. Ha a mintának van illeszkedése, a lefutás nem elfogadó, egyébként elfogadó.*

```
1 MATCH (rakomany:Rakomany)
2 WHERE rakomany = $r AND rakomany:Serult
3 RETURN 'serult'
```

6.5. kódrészlet. A 4.5. ábrának megfelelő osztályozó minta.

Az absztrakciós lépés végén előálló absztrakt állapotváltozások olyan szekvenciák, amelyek illeszkedések változásait tartalmazzák *string* formájában, és mindegyik időbélyeggel van ellátva. Továbbá minden lefutás osztályozott, tehát elfogadó vagy nem

elfogadó. Ezek a szekvenciák alkotják a tanuló komponens bemenetét, amely a feldolgozás során gráfadatbázisba tölti a lefutásokat, majd ezen hajt végre átalakításokat.

6.3. Tanuló komponens

A tanulóalgoritmus megvalósítása szerzőtársammal közösen történt a TDK-munka során [8]. A tanulóalgoritmus során szükséges gráftranszformációkat a Neo4j gráfadatbázison valósítottuk meg. Cypher lekérdezések segítségével hajtunk végre az adatbázison lekérdezéseket, illetve módosítjuk is azt. A 6. algoritmus logikája Java nyelven van megvalósítva. Azonban ennek egyes részletei gráflekérdezéseket takarnak. Így a tanulóalgoritmus megvalósítása két nagyobb részre bontható.

- **Az algoritmus logikája.** A Java nyelven megírt kód felelős azért, hogy az adatmodellen a lekérdezések jó sorrendben fussanak le. Ebben a kódban van implementálva, hogy először a *APTA* előállításához szükséges lekérdezések hajtódjanak végre, majd a piros és kék állapotokat beszínező lekérdezések. Ezt követően pedig folyamatosan kérdezze le a *merge* és *color* műveletek metrikáit, majd válassza ki ezek közül a legjobbnak bizonyulót, és hajtsa is végre azt. Ezek a végrehajtások szintén módosító adatbázis-lekérdezéseket takarnak.

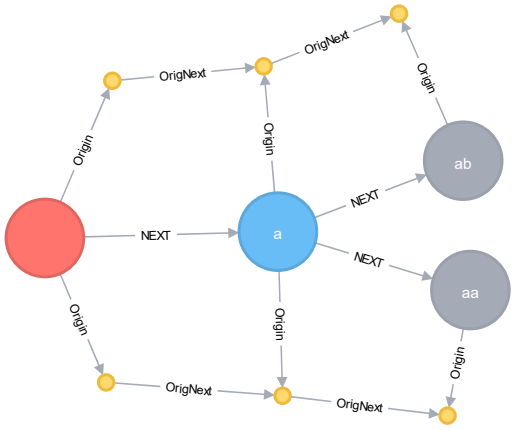
- **Adatmodell.** Megvalósításunkban mind az aktuális hipotézismodellt, mind a rendelkezésre álló lefutásokat gráfadatbázisban tároljuk.

  A 6.2. ábra szemléltet egy általunk használt adatmodellt. Az adatmodellen kétféle csúcs és háromféle él látható. Az egyik féle csúcs, mely az ábrán nagyobbként jelenik meg, a H hipotézismodell állapotát jelenti. A kisebb csúcs pedig az eredeti S szekvenciák állapotait. Vannak élek, melyekre *NEXT* van írva, ezek a H hipotézismodell állapotátmenetei. Ezen kívül használunk még *Origin* éleket, melyek a hipotézismodell állapotait a szekvenciabeli állapotaikhoz kötik hozzá. Illetve használunk még *OrigNext* éleket is, melyek a szekvenciákban jelölik az állapotátmeneteket.

  Felhasználjuk a tulajdonsággráf által kínált lehetőségeket, a gráf csomópontokra és az élekre is illesztünk címkéket, illetve tulajdonságokat. A piros és kék állapotok megjelölését például címkékkel oldottuk meg. Az állapotokon ezen kívül látszódnak a *trace* tulajdonságok, melyek azt jelentik, hogy az eredeti lefutásokban addig az állapotig milyen karakterek szerepeltek. Egy-egy állapotnak tulajdonsága még például egy *id* mező is, mely a gráfadatbázis lementésénél és visszatöltésénél játszik szerepet.

Egy-egy művelet megvalósítása általában több lekérdezés hívásából épül fel. Egy *merge* művelet metrikájának kiszámításához például először le kell kérdezni, hogy melyik piros-kék pár az, amelyiket még nem vizsgáltuk *merge* metrika szempontjából, azután az adott pár párhuzamos útvonalait kell jelölni. Össze kell kötni az összevonandó csúcsokat, és az ezen összevonások után keletkező párhuzamos útvonalak miatt összevonandó csúcsokat, amíg lehet. Majd ki kell számítani, hogy ez az összevonás lehetséges-e, és ha igen, mennyire előnyös. Ezután az egyes számítások során tett ideiglenes jelzéseket törölni kell az adatbázisból.

Az alábbiakban bemutatok néhány összetettebb adatbázislekérdezést. Erre a célra a *merge* metrikájának a kiszámítását végző lekérdezéseket választottam ki. Egy ilyen lekérdezés az algoritmusban akkor futhat le, mikor már van egy kezdetleges hipotézismodellünk. Ebben a hipotézismodellben egyaránt vannak piros és kék állapotok



6.2. ábra. Egy Neo4j-ben tárolt egyszerű H hipotézismodell és S eredeti lefutások ábrázolása

is. A Java kódból ez a lekérdezés akkor hívódik, mikor szeretnénk kiszámolni, hogy melyik műveletet lenne a legérdemesebb végrehajtani, minek a legmagasabb a metrikája.

Merge művelet metrikájának kiszámítása

A 7. algoritmus a *merge* művelet metrikájának számítási lépéseit mutatja be. A H hipotézismodell aktuális állapotában keressük azt a piros-kék állapotpárt, amelynek az összevonása a legelőnyösebb, és a legnagyobb metrika értékkel rendelkezik; keressük továbbá a metrika értékét. A későbbi összevonáshoz szükség van arra, hogy mely állapotokat melyik másik állapottal kell összevonni. A számítás során is szükség van ezekre az értékekre, így ezeket is visszaadjuk.

Minden piros-kék párt ellenőrizzük, azaz jelöljük, hogy az összevonása milyen további állapotok összevonását eredményezné. Az összevonandó állapotpárok halmazát addig bővítjük, ameddig lehetséges, majd ezen párok alapján kiszámoljuk a metrikát, és amennyiben nagyobb, mint az eddig maximum, lecseréljük a maximumot.

Az algoritmus egy-egy pontja gráfon végzett számításokat jelent, amelyek hatékony leírásához és végrehajtásához Cypher nyelvű lekérdezéseket használunk. Az algoritmus a lekérdezéseket azok eredményétől függően hajtja végre.

A 6.6. kódrészlet első 3 sorában keressük azokat a piros-kék állapotpárokat, amelyeket még nem ellenőriztünk, azaz köztük nincs Checked típusú él. Az ilyen párok közül a `LIMIT 1` sor maximum egyet hagy meg. Ezen csúcsok közé behúzunk egy új Checked élet, majd visszatérünk a csúcspárral, amelyet a további lekérdezésekben felhasználunk. Ha nincs ilyen csúcspár, akkor az eddigi maximális értékekhez tartozó paraméterekkel tér vissza az algoritmus.

```
1 MATCH (r:Red), (b:Blue)
2 WHERE NOT (r)-[:Checked]-(b)
3 WITH r, b
4 LIMIT 1
5 CREATE (r)-[:Checked]->(b)
6 RETURN r, b
```

6.6. kódrészlet. Összevonható kiinduló csúcsok keresése és jelölése.

A 6.7. kódrészletben az aktuálisan vizsgált piros-kék állapotpárt használjuk fel bemenő paraméterként (`$r`, `$b`). A 2. és 3. sorokban keressük azokat az *rp*, *bp* útvonalakat, amelyek az r, illetve a b csúcsokból NEXT élek 0 vagy hosszabb sorozatán át State címkéjű csúcsban végződnek. Azaz a piros és kék állapotból állapotátmenetek sorozatán át milyen állapotokba juthatunk.

Algoritmus 7: Összevonás művelet metrikájának számítása

Input: H hipotézismodell
Output: (r_{max}, b_{max}) összevonandó állapotpár; $P_{max} \subset Q \times Q$ tranzitívan összevonandó állapotpárok halmaza, n_{max} metrika

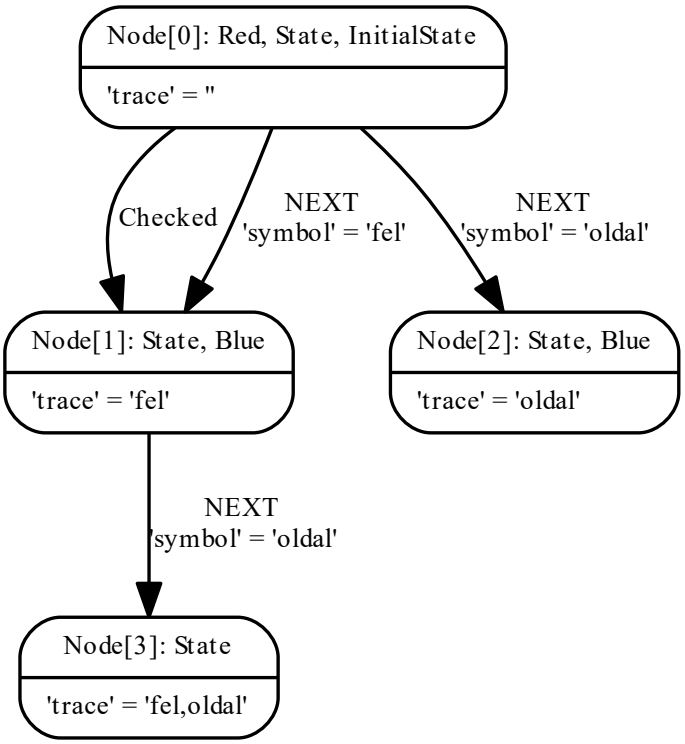
1 $n_{max} := -\infty$
2 **while** *H automatában van még összevonható (r, b) piros-kék pár* **do**
3 (r, b) ellenőrizve jelölés
4 (r, b)-ből párhuzamos útvonalakon elérhető $P \subset Q \times Q$ csúcspárok jelölése
5 **while** *ha $P' \neq \emptyset$, ahol $P' \subset Q \times Q \setminus P$ a tranzitívan összevonandó csúcspárok* **do**
6 $P := P \cup P'$
7 **end**
8 $n :=$ metrika számítása P alapján
9 **if** $n_{max} < n$ **then**
10 $(r_{max}, b_{max}) := (r, b)$
11 $P_{max} := P$
12 $n_{max} := n$
13 **end**
14 P jelölések törlése a gráfból
15 **end**
16 (r, b) párokról ellenőrizve jelölés törlése

A 4. sorban az eddigi értékeket továbbadjuk, és kiszámítjuk, hogy az útvonalak mentén az egyes átmeneteken milyen karakterek szerepeltek. Ehhez listanézetet *(list comprehension)* használunk. A `rels` függvénnyel az útvonal mentén szereplő élek listáját kapjuk meg, amelyeket a `|` jel jobb oldalán az élek symbol nevű tulajdonságainak listájává alakítunk. Ezután a két útvonalon előforduló karakterek listájának összehasonlításával biztosítjuk, hogy a két útvonalon rendre azonos karakterek szerepelnek. A két útvonal *párhuzamusan* fut, azaz az r állopotból ω szó hatására az `rp` útvonalat járjuk be, míg b állapotból ugyanezen ω szó hatására a `bp` útvonalat járjuk be. A 7. sorban az útvonalak hossza alapján indexet állítunk elő, majd a párosítjuk az `rp` útvonal 0. csúcsát a `bp` útvonal 0. csúcsával, az 1. csúcsot az 1. csúccsal stb. Ha összevonjuk az r és b csúcsokat, akkor ezek az állapotpárok azok, amelyeket szintén össze kell vonni.

A 8. sorban az állapotpárok listáját az `UNWIND` kulcsszóval kiterítjük, azaz a lekérdezés eredményének minden sorába – a meglévő oszlopok mellé – csak egy-egy állapotpár kerül. Ezután az r és b mellé az állapotpárok listája kerül, de a `DISTINCT` kulcsszó miatt minden állapotpár csak egyszer. Erre a kiterítésre azért van szükség, mivel több párhuzamos útvonalban is szerepelhet ugyanaz az állapotpár. Például a piros-kék pár minden útvonal elején szerepel.

```
1  WITH $r AS r, $b AS b
2  MATCH rp=(r)-[:NEXT*0..]->(:State)
3  MATCH bp=(b)-[:NEXT*0..]->(:State)
4  WITH r, b, rp, bp, [rr IN rels(rp) | rr.symbol] AS rss, [br IN rels(bp) | br.symbol] AS bss
5  WHERE rss = bss
6  WITH r, b, rp, bp,
7    [ i IN range(0, length(rp)) | [nodes(rp)[i], nodes(bp)[i]] ] AS pairs
8  UNWIND pairs AS pair
9  WITH r, b, collect(DISTINCT pair) AS dpairsResult
10
11 UNWIND range(0, length(dpairsResult)-1) AS i
12 WITH dpairsResult[i][0] AS rpc, dpairsResult[i][1] AS bpc, i, r, b, dpairsResult
13 CREATE (rpc)-[:IndexPair {index: i}]->(bpc)
14 SET rpc:IndexedMerge, bpc:IndexedMerge
15 RETURN r, b
```

6.7. kódrészlet. Piros-kék párból párhuzamos útvonalakon elérhető csúcsok jelölése.



6.3. ábra. A hipotézismodell egy állapota a gráfadatbázisban

A 6.7. kódrészlet első 9 sorát a 6.3. gráfon $\{\$r = 0, \$b = 1\}$ paraméterekkel futtatva a 6.1. táblázatban látható eredményt kapjuk. A gráfon látható, hogy 0-s csúcs a kezdőállapot, és jelenleg csak ez az állapot piros. Az ebből a csúcsból állapotátmeneten közvetlenül elérhető csúcsok kékek. A 6.6. minta a 0-s és 1-es csúcsok közé húzott be egy Checked típusú élet.

Az aktuális piros-kék párból párhuzamos útvonalakon haladva elérhető állapotpárok láthatók a `dpairsResult` oszlopban. A $(0, 1)$ azonosítójú csúcspárból oldal karakter mentén jutunk el a $(2, 3)$ azonosítójú csúcspárba. Ezek a csúcspárok lesznek összevonva, ha ez a piros-kék pár lesz összevonásra kijelölve, azaz a legmagasabb metrikával rendelkezik.

A 6.7. kódrészlet 11-12. sorában a már egyedi állapotpárok listáját ismételten kiterítjük, hogy a párok tagjait IndexedMerge címkével jelöljük (14. sor), és közéjük sorszámozott IndexPair típusú éleket tegyünk (13. sor). A minta a piros-kék párral tér vissza.

r	b	dpairsResult
{"trace":"", "id":"0"}	{"trace":"fel", "id":"1"}	[[{"trace":"","id":"0"},{"trace":"fel","id":"1"}], [{"trace":"oldal","id":"2"},{"trace":"fel,oldal","id":"3"}]]

6.1. táblázat. A 6.7. kódrészlet első 9 sorának eredménye a 6.3. gráfon futtatva

A 6.8. kódrészlet az összevonandó állapotpárok halmazát növeli. Az összevonások miatt keletkeznek olyan csúcsok, amelyek egy összevont állapotból párhuzamos útvonalakon elérhetőek, így szintén összevonandóak. Ezt a lekérdezést addig kell futtatni, amíg már nem talál egy új összevonandó párt sem.

A 6.8. minta 1. sorában kiválasztjuk az összevonandó elemeket, azaz azokat az IndexedMerge címkével jelölt csúcsokat (*s1*, *s2*), amelyek legalább egy hosszú IndexPair típusú élek irányítatlan sorozatával össze vannak kötve. A 2. sorban vesszük az ezekből a csúcspárokból kiinduló NEXT típusú élek legalább egy hosszú sorozatán át elérhető állapotpárokat (*s12*, *s22*). 3. sorban szereplő szűrés során az adatbázisbeli belső *id*-k összehasonlításával garantáljuk, hogy egy pár sorrendtől függetlenül csak egyszer szerepeljen. Továbbá szűrünk arra is, hogy az *s1* és *s2*-ből kiinduló NEXT típusú élek sorozatának hossza megegyezzen. Továbbá elvárás az is, hogy *s12* és *s22* ne egyezzen meg, illetve ne legyen IndexPair típusú élek 0 hosszú vagy hosszabb sorozatán át összekötve, mivel ekkor az összevonás már jelölve van, felesleges IndexPair éllel jelölni. Ezek után az 5-6. sorban ellenőrizzük, hogy a párhuzamos útvonalakon azonos karakterek szerepelnek-e. A 7. sorban listába gyűjtjük az összevonandó párokat.

A 9-10. sorban vesszük az eddigi IndexPair éleket, és megkeressük a legnagyobb kiosztott sorszámot, hogy a következő sorszámot képezzük. A 12-13. sorban a megtalált összevonandó párok listáját kiterítjük, és ellátjuk sorszámmal az előbb kiszámolt maximális értéket növelve. A 14-15. sorban behúzzuk közéjük a sorszámozott IndexPair élet, és IndexedMerge címkével jelöljük a csúcsokat. Így képezzük az összevonandó csúcsok egyre bővülő halmazát, amíg lehet.

```
1  MATCH (s1:IndexedMerge)-[:IndexPair*]-(s2:IndexedMerge),
2    s1p = (s1)-[:NEXT*]->(s12:State), s2p = (s2)-[:NEXT*]->(s22:State)
3  WHERE id(s1) > id(s2) AND length(s1p) = length(s2p)
4    AND NOT (s12)-[:IndexPair*0..]-(s22)
5  WITH s12, s22, s1p, s2p, [s1r IN rels(s1p) | s1r.symbol] AS s1ss, [s2r IN rels(s2p) | s2r.symbol] AS
       s2ss
6  WHERE s1ss = s2ss
7  WITH collect([s12, s22]) AS pairs
8
9  MATCH ()-[ip:IndexPair]->()
10 WITH max(ip.index) + 1 AS nextIndex, pairs
11
12 UNWIND range(0, length(pairs)-1) AS idx
13 WITH pairs[idx][0] AS s12, pairs[idx][1] AS s22, idx + nextIndex AS edgeIndex
14 CREATE (s12)-[:IndexPair {index: edgeIndex}]->(s22)
15 SET s12:IndexedMerge, s22:IndexedMerge
```

6.8. kódrészlet. Összevonások miatti további összevonások jelölése.

A 6.9. kódrészlet az összevonandó állapotpárok alapján számol metrikát. Az összevonandó állapotpárokat IndexPair típusú élek legalább 1 hosszú sorozata köti össze. Ezeket a párokat kéri le a 2. sor, majd a 3. sorban lévő szűrés és a 4. sorban lévő `DISTINCT` kulcsszó garantálja, hogy minden pár sorrendtől függetlenül egyszer szerepeljen.

A 9. sorban a `reduce` függvény segítségével számítunk metrikát az állapotpárok listája alapján, illetve eldöntjük, hogy egyáltalán lehetséges-e az adott összevonás. Ezt utána a `metric` nevű oszlop tárolja. A függvény első paramétere az `acc` akkumulátor, amely első eleme a metrika, második eleme pedig azt jelzi, hogy az összevonás lehetséges-e. Ennek értéke kezdetben (0, *true*). `pair` néven végigvesszük az összevonandó állapotpárok `dpairs`

nevű listáját. Az állapotok accepting és rejecting logikai típusú tulajdonsága jelzi az állapot elfogadását, vagyis azt, hogy az adott állapot elfogadó, elutasító vagy egyik sem, azaz nincs róla információ.

- Ha a pár mindkét tagjának egyezik az elfogadása (11-12. sor) és ez elfogadó vagy elutasító (13. sor), akkor megnöveljük a metrikát, mivel ez az összevonás előnyös. Az `acc` lista második elemeként az előző elemet továbbadjuk, tehát az összevonás lehetséges, ha eddig nem volt olyan, ami megakadályozta volna.
- Ha a pár egyik tagja elfogadó, míg a másik elutasító (16-17. sor), akkor nem szabad összevonni. Ekkor az akkumulátor második elemét hamisra állítja.
- Egyébként az összevonás semleges, így sem a metrika, sem az összevonás szabályosságát jelző érték nem változik (20. sor).

A 24. sorban kiszűrjük az összevonást, ha az akkumulátor végleges értéke alapján az összevonás nem lehetséges. Egyébként pedig visszaadjuk a piros-kék párt, az összevonandó párokat és a metrikát. Ezután a vezérlő kódban (7. algoritmus 9-13. sor), ha a metrika értéke az eddigi maximumot meghaladja, akkor eltároljuk a lekérdezés eredményét.

```
1  WITH $r AS r, $b AS b
2  MATCH (rpc:IndexedMerge)-[:IndexPair*1..]-(bpc:IndexedMerge)
3  WHERE id(rpc)>id(bpc)
4  WITH r, b, collect(DISTINCT [rpc, bpc]) AS dpairs
5
6  WITH
7    r, b,
8    dpairs,
9    reduce(acc = [0, true], pair IN dpairs | (
10     CASE
11       WHEN pair[0].accepting = pair[1].accepting
12         AND pair[0].rejecting = pair[1].rejecting
13         AND pair[0].accepting <> pair[0].rejecting
14       THEN [acc[0] + 1, acc[1]]
15
16       WHEN ((pair[0].accepting OR pair[1].accepting)
17         AND (pair[0].rejecting OR pair[1].rejecting))
18       THEN [0, false]
19
20       ELSE [acc[0], acc[1]]
21     END
22     )
23   ) AS metric
24 WHERE metric[1]
25 WITH r, b, metric[0] AS metric, dpairs
26 RETURN r, b, dpairs, metric
```

6.9. kódrészlet. Összevonások alapján metrika számítása.

A 6.10. kódrészlet a vizsgált piros-kék pár kapcsán történt ideiglenes jelöléseket távolítja el. Az 1-2. sor eltávolítja az összevonandó csúcsokon szereplő IndexedMerge címkét. Majd a 4. sor megszámolja, hogy hány sora volt az eddigi lekérdezésnek, így pontosan egy sornyi eredmény keletkezik a 4. sor után, függetlenül az eddigi lekérdezésektől. Ezt felhasználva lehetséges egy további lekérdezést futtatni. Az 5-6. sor törli az összevonandó párokat összekötő IndexPair típusú éleket.

```
1  MATCH (s:IndexedMerge)
2  REMOVE s:IndexedMerge
3
4  WITH count(*) AS dummy
5  MATCH ()-[e:IndexPair]->()
6  DELETE e
```

6.10. kódrészlet. Összevonás jelölések törlése.

A 6.11. kódrészlet az összes lehetséges piros-kék pár metrikájának kiszámítása után távolítja el a párok ellenőrzését jelző Checked típusú éleket.

```
1 MATCH ()-[c:Checked]-()
2 DELETE c
```

6.11. kódrészlet. Ellenőrzött kiinduló csúcs jelölése törlése.

Az algoritmus az összes lehetséges piros-kék pár metrikáját kiszámolta, ezután visszatér a maximális metrikával, a hozzá tartozó összevonandó piros-kék párral és a többi összevonandó állapotpárral. A metrika alapján lehet eldönteni, hogy a tanulóalgoritmus során melyik lépést érdemes végrehajtani.

6.4. Példa lefutás

A vasúti esettanulmányon a 6.2. szakaszban definiált absztrakció felhasználásával a 6.3. táblázatban látható lefutások állhatnak elő. A lefutások során egyszerűsített elnevezéseket használtunk, amelyek a 6.2. táblázatban látható elemi változásoknak felelnek meg.

Az 1. lefutás egy sikeres lefutást mutat, ahol az egyik vágányon álló szerelvényről ugyanazon a vágányon álló – feltehetőleg eltérő – szerelvényre pakolták át a rakományt. A 2. lefutás két vágány közötti mozgatást ír le. A 3. lefutás során előbb egyik vágányról a másik vágányra pakolták a rakományt, majd vissza az eredeti vágányra. Ezek szabályos, elfogadó lefutások voltak, mivel nem sérült a rakomány, nem volt veszélyes helyzet, például nagy magasságból elengedés, vagy alacsony magasságban vízszintes mozgatás.

A 4. lefutás során fenti helyzetében elengedték a csomagot, így az megsérült, elutasító a lefutás. Az 5. lefutás során veszélyes módon – felemelés nélkül, alacsony magasságban – rakodták át az egyik vágányról a másikra a csomagot, így ez egy elutasító lefutás. A 6. lefutás egy szabályos átpakolás után egy szabálytalan visszapakolást mutat be, így szintén elutasító.

be	$\langle$	$\langle \varepsilon, \mathsf{vágány1} \rangle$	, 0 ,	0 $\rangle$
megfog	$\langle$	0	, $\langle \varepsilon, \mathsf{darun} \rangle$,	0 $\rangle$
elenged	$\langle$	0	, $\langle \mathsf{darun}, \varepsilon \rangle$,	0 $\rangle$
fel	$\langle$	0	, 0 ,	$\langle \varepsilon, \mathsf{fent} \rangle \rangle$
le	$\langle$	0	, 0 ,	$\langle \mathsf{fent}, \varepsilon \rangle \rangle$
balra	$\langle \langle \mathsf{vágány1}, \mathsf{vágány2} \rangle$,	0	, 0	$\rangle$
jobbra	$\langle \langle \mathsf{vágány2}, \mathsf{vágány1} \rangle$,	0	, 0	$\rangle$

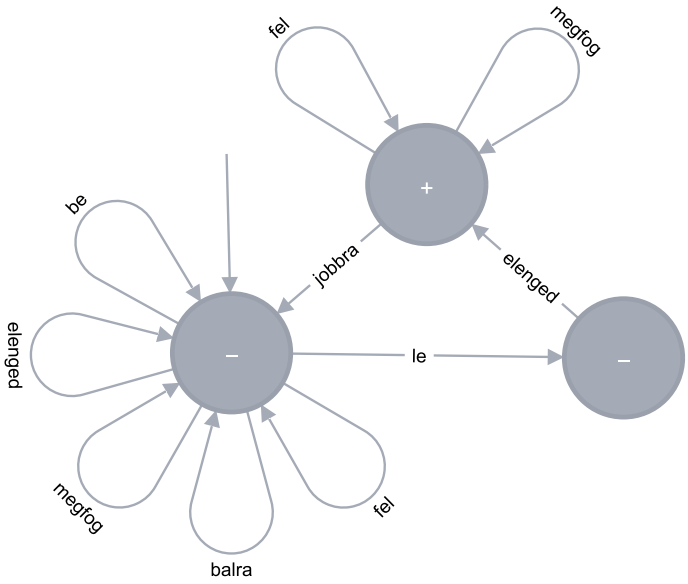
6.2. táblázat. Egyszerűsített elnevezések és elemi értékek változásai közti megfeleltetés

1.	be, megfog, fel, le, elenged	+
2.	be, megfog, fel, balra, le, elenged	+
3.	be, megfog, fel, balra, le, elenged, megfog, fel, jobbra, le, elenged	+
4.	be, megfog, fel, elenged, le	−
5.	be, megfog, balra, elenged	−
6.	be, megfog, fel, balra, le, elenged, megfog, jobbra, elenged	−

6.3. táblázat. Absztrakt osztályozott lefutások

Az automatatanulás során a 6.3. táblázatban látható lefutásokból a 6.4. ábrán látható véges automata keletkezett. Az automata alapján látható, hogy azok a kezdőállapotból kiinduló szekvenciák voltak az elfogadóak, amelyekben egymást követően szerepelt a le,

majd a elenged karakter, mivel ez a nagy magasságból leejtett, ill. a lenti állapotban elmozgatott rakomány esetén ezek nem így szerepeltek.



6.4. ábra. A tanulás során előállított véges automata

6.5. Mérési eredmények

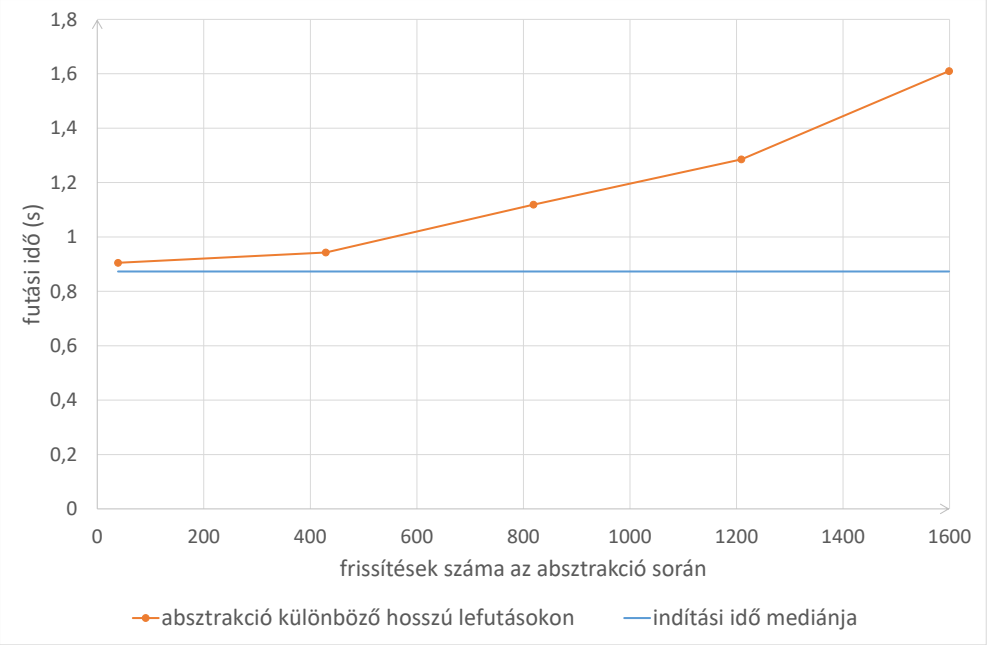
A keretrendszernek elkészült a kezdeti teljesítménymérése. A mérés során a vasúti esettanulmány lefutásait használtuk fel. Az egyes méréseket több (5-10) alkalommal futtattuk, hogy fellépő zajt, a mért értékek változékonyságát kiküszöböljük, továbbá így a program futásának elején tapasztalható gyorsulás, a szoftver "bemelegedése" nem befolyásolja számottevő mértékben az eredményt.

A vizsgálandó rendszer megfigyelése során hosszabb egybefüggő rögzített lefutások lehetnek. A 6.5. ábrán látható, hogy a lefutások hosszának függvényében hogyan változik az absztrakciós lépés futásideje. A kívánt hosszal rendelkező lefutásokat kezdeti gráflefutások ugyanazon lefutásbeli elemekkel való bővítése során állítottuk elő.

A diagramon ábrázoltuk, hogy az absztrakciós lépés egy közel állandó indítási időt követően kezdi meg tényleges futását. Mivel hosszabb lefutás esetén a hosszal arányos számban kell az absztrakciós mintákat futtatni, ezért az absztrakció jól alkalmazható hosszú lefutások szeparálására.

A 6.6. ábrán látható, hogy a tanulás fázis futásidejét a felhasznált lefutások hossza hogyan befolyásolja. A különböző méretű lefutásokat kezdeti lefutások ugyanazon lefutásbeli karakterekkel való bővítése adta. A tanulás fázis futásideje a lefutások hosszától exponenciálisan függ ($R^2 = 0,9935$). Látható, hogy az absztrakció során alkalmazott szeparálás lépéssel hosszú szekvenciák is kezelhetők, amelyeket a szeparáló mintával rövidebb szakaszokra bonthatók. Rövidebb lefutások segítségével nagy mértékben javítható a tanulás futásideje, így érdemes minél rövidebb szakaszokra bontani a lefutást a szeparáló minta segítségével.

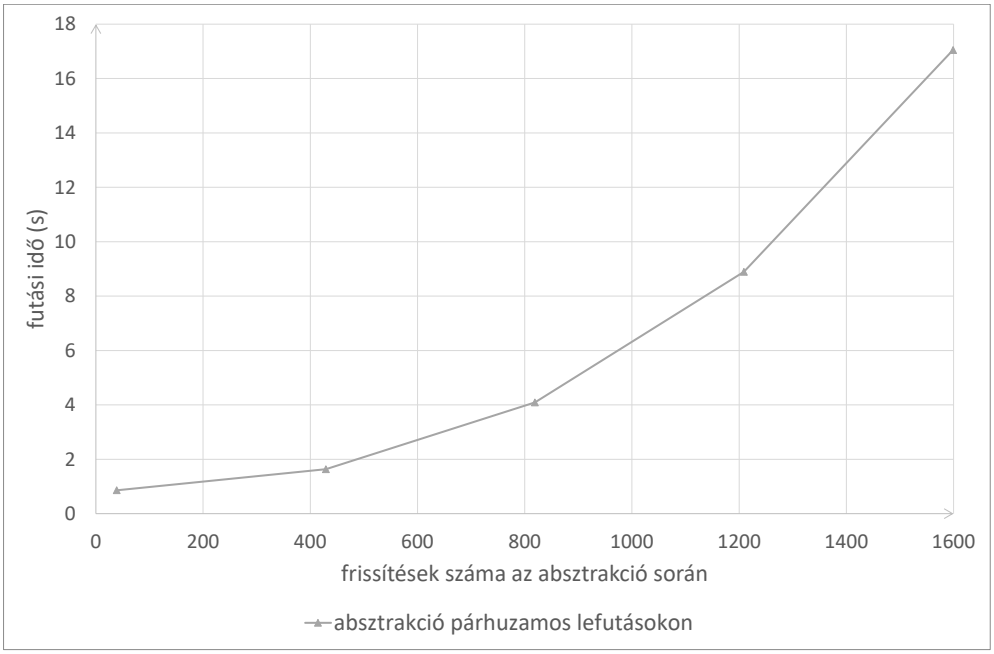
Az absztrakció segítségével nemcsak egymás után következő, hanem párhuzamosan futó viselkedések is felderíthetőek, amelyekre a bemutatott absztrakció nélkül nem volna lehetőség. A 6.7. ábrán ezek futásidejét vizsgáltuk. Párhuzamos viselkedések méréséhez a vasúti esettanulmány futása során változó gráfállapot több példányban



6.5. ábra. Absztrakciós lépés skálázódása a lefutások hosszának függvényében

– eltérő azonosítókkal – szerepel egyszerre a gráfadatbázisban. Összehasonlítva a 6.5. ábrával, látható, hogy azonos számú gráfmintaillesztés több időt vesz igénybe párhuzamos lefutások esetén, mivel ilyenkor a mintaillesztést nagyobb méretű gráfon kell végrehajtani a gráfadatbázisban.

6.6. ábra. Tanulási lépés skálázódása a lefutások hosszának függvényében



6.7. ábra. Absztrakciós lépés skálázódása párhuzamos lefutások esetében

7. fejezet

Kapcsolódó munkák

Ebben a fejezetben áttekintem az irodalomban ismert kapcsolódó munkákat. A fejezet alapjául a 2017. évi, szerzőtársammal közös TDK-munka szolgált [8]. Előző évi szerzőtársammal közös TDK-munkánk [7] is hasonló témát érintett, így az első kettő kapcsolódó munka abban is megtalálható.

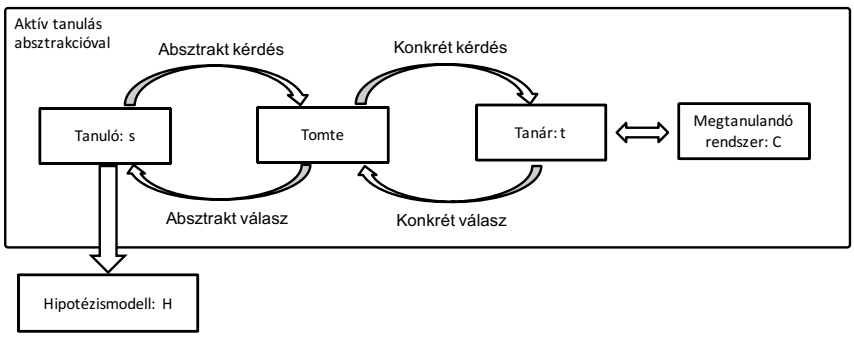
7.1. LearnLib keretrendszer aktív automatatanulásra

A LearnLib egy automatatanulásra készített keretrendszer [16]. Többféle aktív tanulóalgoritmus implementálva van benne. Véges automatákat, és ezen kívül más modelleket is lehet a segítségével alkotni a tanulás alatt álló rendszerről.

A keretrendszernek többféle kiegészítése van. Azonban nagy hátránya, hogy túl nagy állapotteret nem tud megfelelően megtanulni, komplex rendszerek tanulása esetén nem mindig jól alkalmazható. A keretrendszert determinisztikus rendszerek modellezésére fejlesztették, így nemdeterminisztikus viselkedést sem tud megfelelően kezelni. Továbbá a mi esetünkben alkalmazott passzív tanulásra sem kínál megoldást.

7.2. Tomte keretrendszer absztrakciós automatatanulásra

A Tomte keretrendszer [1] egy aktív tanulóalgoritmust használva automatikusan absztrakciót definiál automaták megtanulásához. Ez a tanulóalgoritmus bármi lehet, pl. a LearnLibben megvalósított algoritmusok egyike. A Tomte szerepe a tanulásban csupán a tanulás nyelvének manipulálása (7.1. ábra).



7.1. ábra. Aktív automatatanulás automatikus absztrakcióval

Míg általában a tanulóalgoritmusok közvetlenül tudnak kommunikálni a tanulás alatt álló rendszerrel, a Tomte egy köztes absztrakciós réteget illeszt be közéjük. Azt, hogy

az absztrakció miért is szükséges, könnyedén beláthatjuk, ha visszatekintünk a LearnLib hiányosságaira. Ugyan nem a LearnLib az egyetlen automatatanuló keretrendszer, de biztosak lehetünk abban, hogy más algoritmusok használatával sem lehetne tetszőleges méretig megnövelni a megtanulható állapotok számát. Az absztrakció bevezetésével redukálni lehet ezen állapotteret, így könnyítve az automatatanulást.

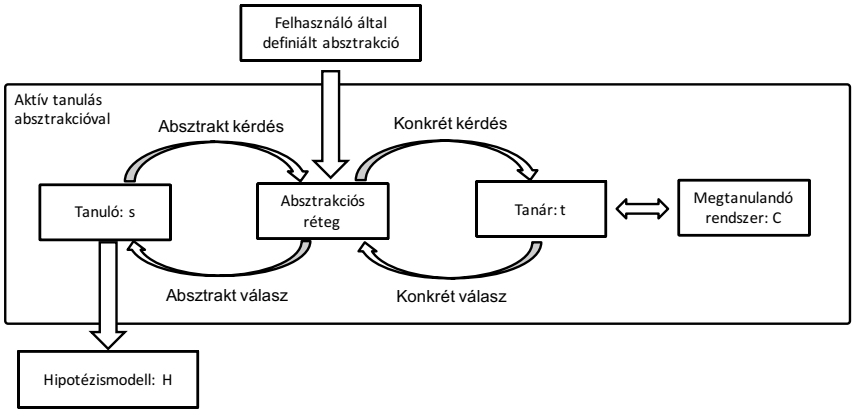
Az eszköz először egy általános absztrakciót definiál, majd ha a tanulás során nemdeterminizmust tapasztalna, akkor a nemdeterminizmust okozó ellenpélda mentén automatikusan finomítja az absztrakciót [5]. A folyamat végén a megtanult automata kellően pontosan fogja tükrözni a valós rendszer működését.

A Tomte fő újítása az, hogy az absztrakció előállítása automatikusan zajlik. Ennek egyaránt vannak előnyei és hátrányai. Nyilvánvaló előnye, hogy átveszi a felhasználótól ezt a feladatot. Hátránya azonban, hogy nem lehet az absztrakció jellegére vonatkozó elvárásokat megfogalmazni a rendszerrel szemben, nem lehet a rendszernek csak egyes funkcióira szűrni.

7.3. Absztrakcióval támogatott tanulás regressziós tesztelés támogatására

Tavalyi – szerzőtársammal közös – TDK-munkánkban [7] a LearnLib felhasználásával alkottunk egy keretrendszert. Ez a keretrendszer lehetőséget biztosít komplex rendszerek megtanulására és azokon való regressziós tesztelés automatizálására.

A Tomte keretrendszerhez hasonlóan mi is aktív tanulást használtunk a megtanulandó rendszer modellezésére. Illetve mi is a tanulás kommunikációs részében fogalmaztuk meg az absztrakciót. A Tomte eszközzel ellentétben azonban a felhasználóra bíztuk az absztrakció megfogalmazását, annak testreszabását (7.2. ábra). Az általunk kidolgozott keretrendszer nyelvi támogatást ad konfigurálható módon absztrakciók definiálására.



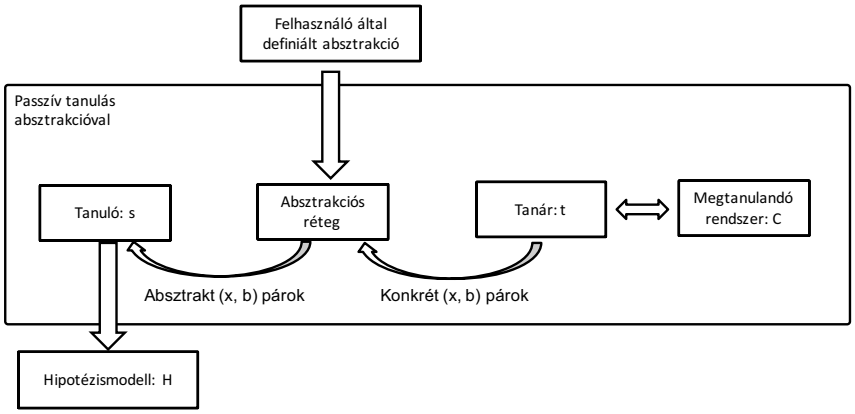
7.2. ábra. Aktív automatatanulás testreszabott absztrakcióval

Az tanulás során előállított modellek alapján teszt szekvenciákat generálunk, amelyek a fejlesztés későbbi fázisaiban alkalmazhatóak regressziós tesztelésre. Ezen kívül a keretrendszer képes ellenőrizni a rendszer különböző fejlesztési fázisokban megtanult modelljeit ekvivalenciaellenőrzéssel, így támogatva a különböző szoftververziók összehasonlítását és a változtatások ellenőrzését.

A keretrendszer azonban csak aktív automatatanulással működik. Így ha olyan rendszert szeretnénk megtanulni, mely nem áll rendelkezésünkre, csupán korábbi lefutásai ismertek, akkor nem alkalmazható. Ezen kívül hátránya még, hogy egy megszabott modellezési formalizmusban kell kifejezni a rendszer be- és kimenetét, amely jóval kötöttebb, mint a jelen dolgozatban bemutatott tulajdonsággráf alapú megközelítés.

7.4. Modellalapú automatatanulás formális modellek szintéziséhez

A megközelítésünk az előbb említett kapcsolódó munkáktól abban különbözik, hogy passzív tanulóalgoritmust és absztrakciót használva tanulja meg a megfigyelt rendszerek viselkedését. Az absztrakciót a felhasználó fogalmazhatja meg, így lehetősége van az általa relevánsnak gondolt információk kiszűrésére. A megvalósítást a 7.3. ábra ábrázolja.



7.3. ábra. Passzív tanulás absztrakcióval

Munkánk során passzív tanulóalgoritmust használunk, így az s algoritmussal a t tanár csak egyirányú kommunikációt végez. A rendszerről rögzített néhány x lefutást és annak b kiértékelését (pozitív vagy negatív) közli a tanuló felé. Ezek után a tanulóalgoritmus már csak ezek felhasználásával tud dolgozni.

A t tanár és s tanuló kommunikációjába helyeztük el az absztrakciós réteget. Ez a tanár által küldött konkrét (x, b) párokat absztrakt (x, b) párokká alakítja, majd az így kapott absztrakt üzeneteket kapja meg a tanuló. A végső H hipotézismodell pedig ezek alapján készül el.

7.5. Absztrakció mint nézeti modellek transzformációja

A bemutatott megközelítés hasonló a [17, 6] cikkekben bemutatottakhoz. A gráfabsztrakciók egy ismert csoportosítása [9] jellemzőik szerint osztályozta a transzformációs megközelítéseket. Ezek alapján a bemutatott absztrakció *gráfok*, *szintaktikus megközelítésű*, *előre funkcionális* transzformációja. *Nincsenek kiegészítő információk* a modellben, tehát az absztrakció elvégzéséhez nem szükséges a tulajdonsággráf módosítása, előkészítése. A transzformáció *cél irányban teljes*, mivel a nézeti modell minden eleme visszavezethető valamilyen forrásmodellbeli modellrészletre. A nézeti modell definíciója *egyirányú* (konkrétból absztraktba képez), és *nem Turing-teljes*. Az *állapotalapú* változások előre irányú továbbterjesztése *automatikusan* végrehajtódik, illetve *offline*, mivel a gráf változásait csak külön frissítés értesítésre dolgozzuk fel. A megközelítés támogatja tetszőleges változtatások átvezetését a forrásmodellről a célmodellre.

8. fejezet

Összefoglaló

Munkám célja komplex rendszerek formális specifikációjának előállítása volt véges automata formájában. Munkám során egy olyan módszert fejlesztettem ki, amely képes kezelni kritikus kiberfizikai rendszerek adatfüggő, gráfjellegű viselkedéseit is.

Az általam adott megközelítés egy gráfminta alapú nyelvet használ arra, hogy a tanulást fókuszálja, és a megtanulandó rendszeren az absztrakciót definiálja. Ez a nyelv használható arra, hogy a lefutásokat paraméterek mentén szeletelje, így lehetővé téve paraméteres rendszerek kezelését. A beérkező információt egy gráf reprezentációban tároljuk, amelyen a felhasználó által definiált gráfmintákat illesztjük és ez alapján állítjuk elő a tanuló algoritmus bemeneteit. A keretrendszerbe egy véges automata szintézis algoritmust illesztettem.
Elméleti eredményeim az alábbiak:

- Kidolgoztam egy módszert, amely az adat- és gráfjellegű viselkedések absztrahálásán keresztül támogatja kiberfizikai rendszerek tanulását.
- Gráfminta alapú nyelvet javasoltam az automatatanulás támogatására, amely nyelven egyrészt a bemeneti adatok specifikálhatóak, továbbá az absztrakció és a paraméterek kezelése definiálható, ezáltal támogatva a tanulás fókuszálását.
- Véges automata tanuló algoritmussal kombináltam a gráfmintaillesztő rendszert az viselkedésmodellek szintézisének támogatására.

Gyakorlati eredményeink az alábbiak:

- Megvizsgáltam kritikus kiberfizikai rendszerek automata tanuláson alapuló analízisének lehetőségét.
- Szerzőtársammal implementáltunk az irodalomban található véges automata tanuló algoritmust gráfminta-illesztő nyelv felhasználásával.
- Implementáltam a keretrendszer prototípusát.

Meg kell jegyezni, hogy az automata tanuló algoritmus egyes lépéseit Cypher nyelven formalizáltuk, amely a továbbiakban segíti, hogy későbbi implementációk is könnyebben elkészíthetőek legyenek.

Munkám során példák segítségével vizsgáltam megközelítés gyakorlati működését és használhatóságát.

8.1. Jövőbeli munka

A jövőben dolgozni fogok azon, hogy a rendszer skálázódását és korlátait is megvizsgáljam. Ehhez tervezem komplexebb ipari esettanulmányok felhasználását. Emellett a meglévő

automata tanuló algoritmuson kívül egyéb tanuló algoritmusokat is tervezem a keretrendszerbe integrálni, így várhatóan a rendszerek tágabb körét tudja majd vizsgálni.

Érdekes elméleti továbbfejlesztési lehetőség a nem megfelelő absztrakciók automatikus finomítási lehetőségének megvizsgálása, amellyel egy CEGAR [5] jellegű tanuló algoritmust készíthetnék, hasonlóan a Tomte eszközhöz [1], azonban felhasználva a Cypher nyelv adta konfigurálhatóságot az absztrakció finomítás vezérlésében.

Lehetséges algoritmikus továbbfejlesztés lenne, hogy az online feldolgozást is minél hatékonyabban támogassam, hogy inkrementális algoritmusokat használnék a gráfmintaillesztés során. Jelenleg az absztrakció során minden frissítéskor az összes absztrakciós mintát újból le kell futtatni. Ehelyett lehetséges inkrementálisan végrehajtani a gráfminta-illesztést, a változásokat csak az érintett lekérdezéseken végigvezetve. EMF-modellek felett inkrementális gráfminta-illesztésre képes például az Eclipse VIATRA keretrendszer [18].

Köszönetnyilvánítás

```
1 MATCH (hallgató:Személy), (személy:Személy)
2 WHERE hallgató = $hallgató
3   AND személy.név IN ['Gujgiczer Anna', 'Farkas Rebeka', 'Semeráth Oszkár', 'Szárnyas Gábor', 'Tóth
      Tamás', 'Vörös András']
4 CREATE (hallgató)-[:KöszönetetMond]->(személy)
```

8.1. kódrészlet. Köszönetnyilvánítás.

Az Emberi Erőforrások Minisztériuma ÚNKP-17-1-I. kódszámú Új Nemzeti Kiválóság Programjának támogatásával készült.

Irodalomjegyzék

[1] Fides Aarts – Faranak Heidarian – Harco Kuppens – Petur Olsen – Frits W. Vaandrager: Automata Learning through Counterexample Guided Abstraction Refinement. In Dimitra Giannakopoulou – Dominique Méry (szerk.): *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, Lecture Notes in Computer Science konferenciasorozat, 7436. köt. 2012, Springer, 10–27. p. URL `https://doi.org/10.1007/978-3-642-32759-9_4`.

[2] Dana Angluin: Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75. évf. (1987) 2. sz., 87–106. p.
URL `https://doi.org/10.1016/0890-5401(87)90052-6`.

[3] Budapesti Műszaki és Gazdaságtudományi Egyetem, Számítástudományi és Információelméleti Tanszék: *Nyelvek és automaták*. `http://www.cs.bme.hu/~friedl/nya/jegyzet-13.pdf`.

[4] Miguel M. F. Bugalho – Arlindo L. Oliveira: Inference of Regular Languages Using State Merging Algorithms with Search. *Pattern Recognition*, 38. évf. (2005) 9. sz., 1457–1467. p. URL `https://doi.org/10.1016/j.patcog.2004.03.027`.

[5] Edmund M. Clarke – Orna Grumberg – Somesh Jha – Yuan Lu – Helmut Veith: Counterexample-Guided Abstraction Refinement. In E. Allen Emerson – A. Prasad Sistla (szerk.): *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, Lecture Notes in Computer Science konferenciasorozat, 1855. köt. 2000, Springer, 154–169. p.
URL `https://doi.org/10.1007/10722167_15`.

[6] Csaba Debreceni – Ákos Horváth – Ábel Hegedüs – Zoltán Ujhelyi – István Ráth – Dániel Varró: Query-Driven Incremental Synchronization of View Models. In Colin Atkinson – Erik Burger – Thomas Goldschmidt – Ralf H. Reussner (szerk.): *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, VAO@STAF 2014, York, United Kingdom, July 22, 2014* (konferenciaanyag). 2014, ACM, 31–38. p.
URL `http://doi.acm.org/10.1145/2631675.2631677`.

[7] Gujgiczer Anna, Elekes Márton Farkas: *Absztrakcióval támogatott tanulás regressziós tesztelés támogatására*. TDK dolgozat. `http://tdk.bme.hu/VIK/Informacios2/Absztrakcioval-tamogatott-tanulas-regresszios`.

[8] Gujgiczer Anna, Elekes Márton Farkas: *Modellalapú automatatanulás formális modellek szintéziséhez*. TDK dolgozat. `https://tdk.bme.hu/VIK/Info/Modellalapu-automatatanulas-formalis-modellek`.

[9] Soichiro Hidaka – Massimo Tisi – Jordi Cabot – Zhenjiang Hu: Feature-Based Classification of Bidirectional Transformation Approaches. *Software and System*

Modeling, 15. évf. (2016) 3. sz., 907–928. p.
URL https://doi.org/10.1007/s10270-014-0450-0.

[10] John E. Hopcroft – Rajeev Motwani – Jeffrey D. Ullman: *Introduction to Automata Theory, Languages, and Computation - International Edition (2. ed)*. 2003, Addison-Wesley. ISBN 978-0-321-21029-6.

[11] Kevin J. Lang – Barak A. Pearlmutter – Rodney A. Price: Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm. In Vasant G. Honavar – Giora Slutzki (szerk.): *Grammatical Inference, 4th International Colloquium, ICGI-98, Ames, Iowa, USA, July 12-14, 1998, Proceedings*, Lecture Notes in Computer Science konferenciasorozat, 1433. köt. 1998, Springer, 1–12. p. URL https://doi.org/10.1007/BFb0054059.

[12] Alexander Maier: Online Passive Learning of Timed Automata for Cyber-Physical Production systems. In *12th IEEE International Conference on Industrial Informatics, INDIN 2014, Porto Alegre, RS, Brazil, July 27-30, 2014* (konferenciaanyag). 2014, IEEE, 60–66. p.
URL https://doi.org/10.1109/INDIN.2014.6945484.

[13] József Marton – Gábor Szárnyas – Dániel Varró: Formalising opencypher Graph Queries in Relational Algebra. In Marite Kirikova – Kjetil Nørvåg – George A. Papadopoulos (szerk.): *Advances in Databases and Information Systems - 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings*, Lecture Notes in Computer Science konferenciasorozat, 10509. köt. 2017, Springer, 182–196. p. URL https://doi.org/10.1007/978-3-319-66917-5_13.

[14] Neo Technology: Cypher query language. https://neo4j.com/docs/developer-manual/current/cypher/, 2017.

[15] Neo Technology: Neo4j. http://neo4j.org/, 2017.

[16] Harald Raffelt – Bernhard Steffen: Learnlib: A library for Automata Learning and Experimentation. In Luciano Baresi – Reiko Heckel (szerk.): *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, Lecture Notes in Computer Science konferenciasorozat, 3922. köt. 2006, Springer, 377–380. p.
URL https://doi.org/10.1007/11693017_28.

[17] Oszkár Semeráth – Csaba Debreceni – Ákos Horváth – Dániel Varró: Incremental Backward Change Propagation of View Models by Logic Solvers. In Benoit Baudry – Benoît Combemale (szerk.): *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016* (konferenciaanyag). 2016, ACM, 306–316. p.
URL http://dl.acm.org/citation.cfm?id=2976788.

[18] Dániel Varró – Gábor Bergmann – Ábel Hegedüs – Ákos Horváth – István Ráth – Zoltán Ujhelyi: Road to a Reactive and Incremental Model Transformation Platform: Three Generations of the VIATRA Framework. *Software and System Modeling*, 15. évf. (2016) 3. sz., 609–629. p. URL https://doi.org/10.1007/s10270-016-0530-4.

[19] SE Verwer – MM De Weerdt – Cees Witteveen: An Algorithm for Learning Real-Time Automata. In *Benelearn 2007: Proceedings of the Annual Machine Learning*

Conference of Belgium and the Netherlands, Amsterdam, The Netherlands, 14-15 May 2007 (konferenciaanyag). 2007.