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

Modellalapú automatatanulás formális modellek szintéziséhez

Készítette
Elekes Márton
Gujgiczer Anna

Konzulens
Farkas Rebeka
Tóth Tamás
Vörös András

2017.

Tartalomjegyzék

Kivonat **i**

Abstract **ii**

1. Bevezetés **1**

2. Előismeretek **3**

2.1. Alapfogalmak, jelölések 3
2.2. Véges automata 4
2.2.1. Minimális véges automata 5
2.2.2. Teljes és hiányos véges automata 5
2.2.3. Determinisztikus és nemdeterminisztikus véges automata 6
2.3. Valósidejű automata 7
2.3.1. Minimális valósidejű automata 8
2.3.2. Teljes és hiányos valósidejű automata 8
2.3.3. Determinisztikus és nemdeterminisztikus valósidejű automata 8
2.4. Automatatanulás 9
2.4.1. Aktív automatatanulás 11
2.4.2. Passzív automatatanulás 12
2.5. Tulajdonsággráf 14
2.6. Gráfminta 14

3. A keretrendszer bemutatása **16**

3.1. Esettanulmány 16
3.1.1. Példa 17

4. Absztrakció **18**

5. Automatatanuló algoritmus **22**

5.1. Algoritmus véges automatára 22
5.1.1. APTA előállítása 22
5.1.2. Merge 23
5.1.3. Red-Blue eljárás 24
5.1.4. Color 25
5.1.5. Evidence driven state merging 25
5.1.6. Az algoritmus 25
5.2. Algoritmus valósidejű automatára 25
5.2.1. TAPTA előállítása 26
5.2.2. Split 27
5.2.3. Merge 28
5.2.4. Red-Blue eljárás 29

5.2.5. Color . . . 29
5.2.6. Evidence driven state merging . . . 29
5.2.7. Az algoritmus . . . 30

6. Megvalósítás **31**
6.1. Architektúra . . . 31
6.2. Absztrakciós komponens . . . 32
6.3. Tanuló komponens . . . 32

7. Kapcsolódó munkák **37**
7.1. LearnLib keretrendszer aktív automatatanulásra . . . 37
7.2. Tomte keretrendszer absztrakciós automatatanulásra . . . 37
7.3. Absztrakcióval támogatott tanulás regressziós tesztelés támogatására . . . 38
7.4. Modellalapú automatatanulás formális modellek szintéziséhez . . . 39

8. Összefoglaló **40**
8.1. Jövőbeli munka . . . 41

Köszönetnyilvánítás **42**

Irodalomjegyzék **44**

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. Különösképpen az időzített komponensek viselkedését nehéz formalizálni.

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

Megközelítésünk újszerűségét a modellalapú bemeneti nyelv és ennek az időzített automatatanuló algoritmusokkal való kombinációja adja. Munkánk során elkészítettünk egy modellalapú automatatanuló keretrendszert időzített szoftverkomponensek mérnöki modelljének szintetizálására. Gráfmintaillesztő nyelvet használunk a tanulás fókuszálására, azaz az absztrakció definiálására. A definiált absztrakció alapján az automatatanuló algoritmus előállítja a rendszer időzített formális modelljét. Az elkészült szoftver felhasználását esettanulmányokkal demonstráljuk.

Módszerünk által lehetővé válik olyan szoftverkomponensek viselkedésének megtanulása, amelyekre eddigiekben nem volt lehetőség komplexitásuk, illetve időtől függő 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 real-time systems is especially challenging.

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

The novelty of our approach is a model-based input language and its combination with timed automaton learning algorithms. We developed a model-based automaton learning framework for synthesizing formal models of timed software components. We 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 timed formal model of the target system. We demonstrate our approach with case studies.

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

1. fejezet

Bevezetés

Kontextus Biztonságkritikus kiberfizikai rendszerek fejlesztésére széles körben használnak modellvezérelt tervezést, amely során lehetőség nyílik a formális módszerek eszköztárát felhasználni a rendszertervek ellenőrzésére a fejlesztés korai fázisaiban is már. Emellett a formális modellekből előállított monitorok segítségével a futásidőben bekövetkező anomáliák is kiszűrhetőek. Kiberfizikai rendszerek jellemzően fizikai közegbe ágyazott intelligens rendszerek, amelyek szenzorokon keresztül észlelik, majd beavatkozókon keresztül visszahatnak a környezetükre. 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.

Problémafelvetés A modellezés, precíz modellek alkotása nehéz feladat, különösen kiberfizikai rendszerek esetén. Ezen rendszerek sokszor komplex módon függenek a környezetüktől, továbbá gyakran bonyolult időzítések, időkényszerek vezérlik őket. A valós rendszerek adatfüggő viselkedéssel rendelkeznek, amit szintén nehéz precízen modellezni. Emellett gyakran gondot okoz az is, hogy bizonyos komponensek belső viselkedése nem ismert, mert vagy nem áll rendelkezésre dokumentáció, vagy még a forráskód sem ismert egy zárt, külső beszállítótól érkező komponens esetén.

Így nincs lehetőség pontos rendszerterveket készíteni, amely meggátolja, hogy formális ellenőrzés segítségével megbizonyosodhassunk a tervek helyességéről.

Célkitűzés Dolgozatunk célja egy olyan módszer megalkotása, mellyel kritikus, időzített kiberfizikai rendszerek viselkedésmodelljét tudjuk automatikusan előállítani tanuló algoritmusok segítségével. Célunk, hogy akár komplex, gráf-struktúrával jellemezhető rendszerek viselkedését is tanulhatóvá, megismerhetővé tegyük. Emellett fontos célunk, hogy az időzítéseket is kezeljük a rendszerben.

Kontribúció Dolgozatunkban bemutatunk egy időzített automata alapú viselkedési modelleket tanuló megközelítést. Az általunk kifejlesztett algoritmus több meglévő algoritmust és módszert ötvöz a viselkedési modellek szintézise során. Az új tanuló algoritmusunk:

- gráfmintákat használ a tanulás számára releváns viselkedések definiálására,
- gráfmintaillesztő algoritmusok alapján állítja elő a tanulás során felhasznált eseményeket, továbbá
- időzített automata tanulást használ a formális modellek szintézisére.

Legjobb tudomásunk szerint a mi megközelítésünk az első, amely hatékonyan kezeli a gráf- és adat-jellegű viselkedéseket időzített környezetben.

Hozzáadott érték Megközelítésünk által a rendszerek szélesebb köre válik a formális analízis által vizsgálhatóvá, az általunk adott algoritmusok segítségével komplex rendszerek adat jellegű, időzített, és akár gráfszerű viselkedései is tanulhatóvá, megismerhetővé válnak.

Dolgozat felépítése A dolgozatunkat a 2. fejezetben az előismeretek áttekintésével kezdjük. Bemutatjuk általánosan az automatákhoz és automatatanuláshoz szükséges alapismereteket, majd az absztrakció bevezetéséhez szükséges ismereteket a tulajdonsággráfokról, illetve a gráfmintákról. A 3. fejezetben az általunk kidolgozott keretrendszert ismertetjük, valamint bemutatunk egy esettanulmányt, melyre alkalmaztuk a keretrendszert. A 4. fejezetben a tanulás során használt absztrakciót, a 5. fejezetben az általunk megvalósított tanulóalgoritmust mutatjuk be elméleti szinten. Az absztrakció és a tanuló algoritmus gyakorlati megvalósításáról a 6. fejezetben írunk. Munkánkhoz kapcsolódó munkákat a 7. fejezetben fejtjük ki. Végül a 8. fejezetben összefoglaljuk munkánkat és kifejtük jövőbeli terveinket.

2. fejezet

Előismeretek

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

2.1. Alapfogalmak, jelölések

Ebben az alfejezetben az általunk használt alapfogalmakat, jelöléseket soroljuk fel. Ezek nagyrészben megegyeznek a formális nyelvek elméletéből ismert jelölésekkel [4], azonban az automatatanulás miatt ismertetünk további fogalmakat.

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. Ez az ábécé tetszőleges karakterek halmaza lehet.

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). Dolgozatunkban a bemeneti esemény egy karaktert jelent, hiszen az automatatanulás során bemeneti eseményekkel kommunikál a tanuló algoritmus a rendszerrel. ▪

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). Dolgozatunkban a lefutás és szó szavakat szinonimaként használjuk. A lefutás a bemeneti események sorozata a rendszer egy adott végrehajtása során. ▪

Valós idejű automaták esetén a karakterekhez időpontok is tartoznak, így a szavak is máshogyan definiálhatók.

Definíció 6 (Időzített szó). Időzített szónak nevezünk egy olyan $\omega_t = (a_1, t_1)(a_2, t_2)...(a_n, t_n)$ sorozatot, ahol $a_1, a_2, ..., a_n \in \Sigma$ és $t_1, t_2, ..., t_n \in \mathbb{N}$. Az ω_t időzített szó hosszát $|\omega_t|$ jelöli. A Σ ábécén képezhető összes szó halmazát Σ_t^* jelöli. ▪

Időzítetlen és időzített esetben is az ábécéből képezhető szavak egy részhalmazára nyelvként hivatkozunk.

Definíció 7 (Nyelv). Egy Σ ábécé feletti L nyelvnek nevezzük az ábécé elemein képezhető összes szó egy (nem feltétlenül véges) részhalmazát ($L \subseteq \Sigma^*$). ▪

Definíció 8 (Időzített nyelv). Egy Σ ábécé feletti időzített nyelvnek nevezzük az ábécé elemein képezhető összes időzített szó (Σ_t^*) egy (nem feltétlenül véges) részhalmazát. A nyelvet L_t-lel jelöljük. Azaz $L_t \subseteq \Sigma_t^*$. ▪

2.2. Véges automata

A véges automata[4][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 9 (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,
- $\delta : Q \times \Sigma \rightarrow Q$ 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 $\omega = a_1a_2...a_n \in \Sigma^*$ szón:

Definíció 10. Az $r_0, r_1, r_2, ..., r_n (r_i \in Q)$ állapotsorozat az $a_1a_2a_3...a_n$ szóhoz tartozó számítás, ha $r_0 = q_0$ és $r_i = \delta(r_{i-1}, a_i)$ minden $i = 1, 2, ..., n$ esetén. ▪

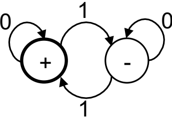
Tehát az automatát a $q_0 = r_0$ állapotból indítva rendre beolvassuk a karaktereket, és a következő állapotba a beolvasott karakter és a jelenlegi állapot által meghatározott átmeneti függvény szerint lépünk.

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 11. 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 12 (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, amikben páros darab 1-es szerepel.*



2.1. ábra. Az 1. példában leírt automata ábrázolása

A 1. példában leírt automatát a 2.1. ábrán látható módon ábrázolhatjuk grafikusan. A dolgozatunkban ezt az ábrázolásmódot fogjuk 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.

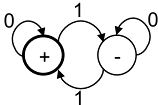
Elfogadó állapot: Az $\mathcal{M}$ automata elfogadó állapotait a $+$ jelzés jelöli.

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

Kezdőállapot: Az $\mathcal{M}$ automata kezdőállapotát, r_0-t 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. Mint ahogy a 1. példában szereplő nyelvvel az 2.2. ábra mindkét automatájának nyelve is 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 13 (Állapottér). $\mathcal{M}$ véges automata állapottere alatt az állapotainak számát értjük, vagyis Q elemeinek a számát. ▪

Definíció 14 (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 szoktak minden lehetséges állapotátmeneti függvényt 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 9. definícióban szereplő automatát teljes véges automatának szokták nevezni.

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

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

2.3. ábra. Véges automata teljessége

Definíció 15 (Teljes véges automata). Teljes véges automatának nevezzük azt az $\mathcal{M}$ véges automatát, amelyben minden $q \in Q$ kezdőállapotból $a \in \Sigma$ karakterre δ állapotátmenet definiálva van. ▪

Definíció 16 (Hiányos véges automata). Hiányos véges automatának nevezzük azt az $\mathcal{M}$ véges automatát, amelyben nincs minden $q \in Q$ kezdőállapotból $a \in \Sigma$ karakterre δ állapotátmenet definiálva. ▪

Hiányos véges automaták esetén amikor egy ω szó számítása során olyan karakter következne, amire az automata adott állapotában nincsen állapotátmenet értelmezve, akkor a számítás elekad. Ebben az esetben az automata nem fogadja el azt ω a szót.

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 17 (Determinisztikus véges automata). Determinisztikus véges automatának nevezzük azt az $\mathcal{M}$ véges automatát, amelyben minden $q \in Q$ kezdőállapotból $a \in \Sigma$ karakterre maximum egy δ állapotátmenet van definiálva. ▪

Definíció 18 (Nemdeterminisztikus véges automata). Nemdeterminisztikus véges automatának nevezzük azt az $\mathcal{M}$ véges automatát, amelyben van olyan $q \in Q$ kezdőállapot és $a \in \Sigma$ karakter, melyre több δ állapotátmenet is definiálva van ▪

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. Valósidejű automata

Az időzített automaták[2] közé tartozik a valósidejű automata[19][8], mely a valós idejű rendszerek leírásához használt formalizmus. Az automata élein nem csak a bemeneti események szerepelnek, hanem az azokhoz tartozó időkorlát is. Egy időkorlát azt határozza meg, hogy egy adott állapotban legalább és legfeljebb mennyi időt kell töltenie az automatának, hogy egy adott bemeneti eseményre végrehajtódjon egy állapotátmenet. Az valósidejű automata, röviden RTA (Real-Time Automaton) így tulajdonképpen a véges automata egy speciális típusa.

Definíció 19 (Valósidejű automata). A valósidejű automatát a $\mathcal{A} = \langle Q, \Sigma, D, q_0, F \rangle$ struktúra írja le, ahol

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

Az automata egy $d \in D$ átmeneti függvénye $\langle q, q', s, \phi \rangle$ négyesből áll, ahol $q, q' \in Q$ a kezdő és végállapotok, $s \in \Sigma$ egy karakter, ϕ pedig az időkorlát, ami egy intervallum $\mathbb{N}$ felett. ▪

Egy valósidejú automata működése a következőképpen írható le egy adott $\omega_t = (a_1, t_1)(a_2, t_2)...(a_n, t_n)$ időzített szón, ahol $a_1, a_2, ..., a_n \in \Sigma$ és $t_1, t_2, ..., t_n \in \mathbb{N}$:

Definíció 20. Az $r_0, r_1, r_2, ..., r_n (r_i \in Q)$ állapotsorozat az $(a_1, t_1)(a_2, t_2)...(a_n, t_n)$ időzített szóhoz tartozó számítás, ha $r_0 = q_0$ és $r_i = d(r_{i-1}, a_i, \phi_i)$, ahol $(t_i - t_{i-1}) \in \phi_i$ minden $i = 1, 2, ..., n$ esetén. ▪

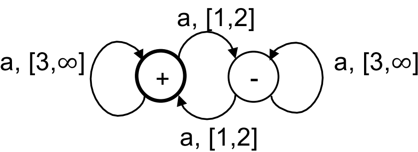
Tehát az automatát a $q_0 = r_0$ állapotból indítva rendre beolvassuk a karaktereket és a karakterek között eltelt időket $(t_i - t_{i-1})$, és a következő állapotba a beolvasott karakter, az eltelt idő és a jelenlegi állapot által meghatározott átmeneti függvénnyel lépünk.

Egy időzített szót elfogad az automata, ha a legutolsó beolvasott karaktere és időköze után az automata egy elfogadó állapotba tér. Ellenkező esetben az automata nem fogadja el, vagyis elutasítja azt az időzített szót. Egy időzített szó eleme a valósidejű automata által leírt időzített nyelvnek, ha az automata elfogadja azt. Pontosabban:

Definíció 21. Az $\mathcal{A}$ valósidejű automata elfogadja $\omega_t \in \Sigma_t^*$ szót, amennyiben $|\omega| = n$ és a ω hoz tartozó számítás végén r_n állapot egy elfogadó állapota az automatának. Vagyis $r_n \in F$. Egyébként A nem fogadja el, más néven elutasítja ω_t szót. ▪

Definíció 22 (Valósidejű automata nyelve). Az $\mathcal{A}$ valósidejű automata időzített nyelve azon ω_t szavak összessége, melyeket $\mathcal{A}$ elfogad. Jelölése: $L_t(\mathcal{A})$. A korábbiakból következik, hogy $L_t(\mathcal{A}) \subseteq \Sigma_t^*$. ▪

Példa 2. *Legyen* $\Sigma = \{a\}$*, vagyis az ábécé egyetlen karaktere legyen* a*. Legyen* $\mathcal{A}$ *egy olyan valósidejű automata, melynek* $L_t(\mathcal{A})$ *nyelvébe azok a szavak tartoznak, amelyekben páros számú a szerepel és a karakterek között legalább 1, legfeljebb 2 időegység telik el.*



2.5. ábra. Az 2. példában leírt valósidejű automata ábrázolása

2.3.1. Minimális valósidejű automata

Valósidejű automata esetén is értelmezett a minimális automata fogalma. Ebben az esetben a következők szerint módosul a definíció:

Definíció 23 (Minimális valósidejű automata). Egy L_t nyelvhez tartozó minimális valósidejű automata egy olyan $\mathcal{A}$ automata, melyre igaz, hogy $L_t(\mathcal{A}) = L_t$, és az ilyen automaták közül $\mathcal{A}$ állapottere a legkisebb. ▪

2.3.2. Teljes és hiányos valósidejű automata

Valósidejű automata esetében a teljesség fogalma annyiban módosul, hogy nem elég csupán minden állapotban minden karakterre definiálni állapotátmenetet, hanem minden időipillanatra is kell.

Definíció 24 (Teljes valósidejű automata). Teljes valósidejű automatának nevezzük azt az $\mathcal{A}$ valósidejű automatát, amelyben minden $q \in Q$ kezdőállapotból $a \in \Sigma$ karakterre és $t \in \phi$ időkorlátra d állapotátmenet definiálva van. ▪

Definíció 25 (Hiányos valósidejű automata). Hiányos valósidejű automatának nevezzük azt az $\mathcal{A}$ valósidejű automatát, amelyben nincs minden $q \in Q$ kezdőállapotból, $a \in \Sigma$ karakterre és $t \in \phi$ időkorlátra d állapotátmenet definiálva. ▪

2.3.3. Determinisztikus és nemdeterminisztikus valósidejű automata

Valósidejű automata esetében előfordulhat, hogy egy adott állapotból egy adott karakter olvasását követően a rendszer többféle állapotba kerülhet. Azonban nem megengedett, hogy ezeknek az átmeneteknek az időkorlátjai átlapolódjanak. Vagyis egy adott állapotban egy adott karakter hatására egy adott idő elteltével már nem megengedett, hogy többféle állapotba kerüljön a rendszer.

Definíció 26 (Determinisztikus valósidejű automata). Determinisztikus valósidejű automatának nevezzük azt az $\mathcal{A}$ valósidejű automatát, amelyben nincs olyan $q \in Q$ kezdőállapot és $a \in \Sigma$ karakter, melyre több d állapotátmenetnek t időkorlátjai rendelkeznének közös résszel. ▪

Definíció 27 (Nemdeterminisztikus valósidejű automata). Nemdeterminisztikus valósidejű automatának nevezzük azt az $\mathcal{A}$ valósidejű automatát, amelyben van olyan $q \in Q$ kezdőállapot és $a \in \Sigma$ karakter, melyre több d állapotátmenetnek t időkorlátjai rendelkeznek közös résszel. ▪

2.4. Automatatanulás

Az automatatanulás célja egy black box rendszer működésének feltérképezése az elérhető információk alapján. Black box rendszernek nevezzük azt a rendszert, melynek pontos belső működése nem ismert, csupán a külvilággal való kommunikációja. Az automatatanulás egy 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 (2.4.2. fejezet) esetén fordulhat elő, hogy az algoritmusnak kevés információ áll rendelkezésére.

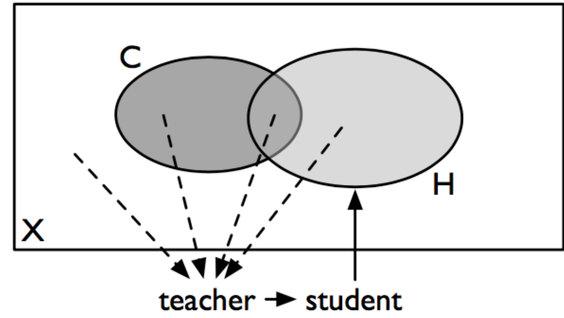
- **Minimális:** Szemléletesség miatt fontos, illetve kisebb memória elég hozzá. Általában nem tudja garantálni a minimális automata megtanulását egy algoritmus, de törekszik rá.

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

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

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

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 állít 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.



2.6. ábra. A tanulásban résztvevő komponensek

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 az arra adott válaszát is a rendszernek is (2.6. ábra).

Definíció 28 (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. ▪

Definíció 29 (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 pozitív példa C-ben. Vagyis ha H és C nyelve megegyezik. ▪

Definíció 30 (Tanár). C rendszer tanára egy olyan orákulum, mely megmondja egy $x \in X$ elemről, hogy C szerint az egy pozitív, vagy negatív példa. Vagyis visszatérési értéke egy címkézett (x, b), ahol b egy boolean típus, mely akkor $igaz$, ha $x \in C$, ellenkező esetben $hamis$. ▪

Definíció 31 (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 igaz, hogy $x \in H$ akkor és csak akkor igaz, ha $x \in C$ is igaz. ▪

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

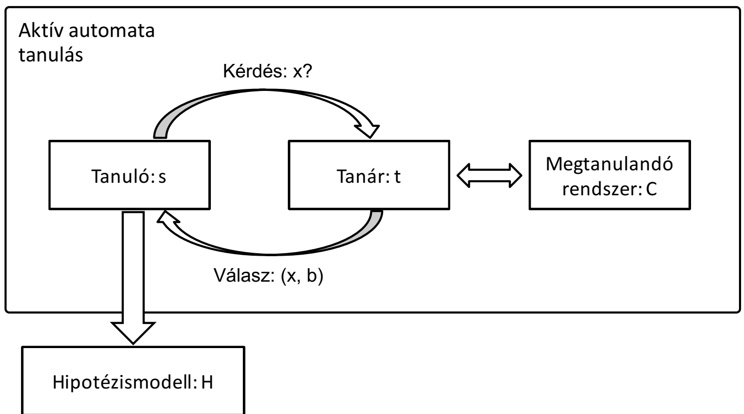
- **Aktív vagy passzív tanulás**
  Aktív tanulás során az algoritmus, ha szükségesnek érzi, feltehet további kérdéseket a rendszer működését illetően (pl. megkérdezheti egy lefutásról, hogy pozitív-e), ezzel bővíteni tudja a tudását. Ezzel ellentétben a passzív tanulás során az algoritmusnak egy előre 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 (x, b) 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 pl. 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.

Dolgozatunkban szeretnénk még jobban kifejteni az aktív és passzív tanulást. Mindkét típusú tanulást egy-egy egyszerű példán keresztül szemléltetnénk (3. és 4. példa).

2.4.1. Aktív automatatanulás

Az aktív tanulóalgoritmus eleinte semennyi információval nem rendelkezik a megtanulandó rendszerről. Ilyenkor a kezdeti hipotézismodellje egy nagyon általános automata (általában 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 (2.7. ábra).



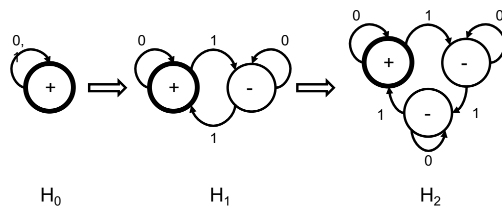
2.7. ábra. Aktív automatatanulás

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 3. *Legyen $\Sigma = \{0, 1\}$, vagyis az ábécé karakterei legyenek 0 és 1. X legyen továbbra is Σ^*, vagyis a karakterekből képezhető összes szó halmaza. Legyen a C által elfogadott nyelv azon x- ek halmaza, ahol $x \in X$ és amely x-ben az 1-esek száma hárommal osztható.*

Ha megfigyelnénk egy aktív tanuló algoritmust, mely a 3. példában definiált rendszert tanulja és a tanulás különböző fázisaiban lekérdeznénk az aktuális hipotézismodelljét, akkor a 2.8 ábrán láthatóhoz hasonló eredményt kapnánk.



2.8. 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[3] 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. A 1. pszeudokód írja le az algoritmus működését.

Algoritmus 1: Angluin féle aktív automatatanulás

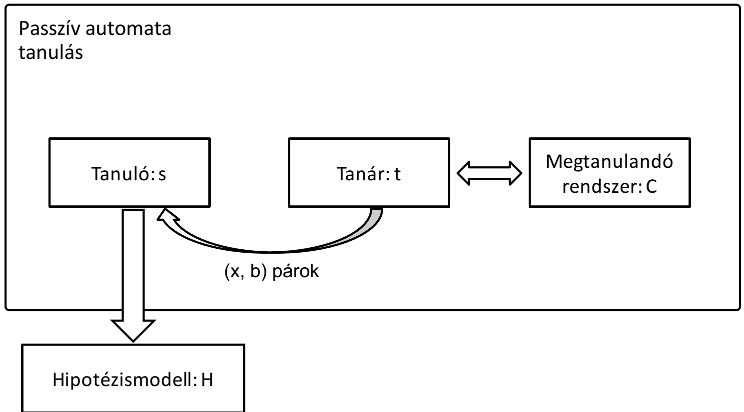
Input: C rendszer és t tanára, aki választ ad a mem() tagsági- és az eq() ekvivalencia kérdésekre
Output: H hipotézismodell C rendszer viselkedéséről

1 Kérdezze meg mem(ϵ) tagsági kérdést, hogy a kezdőállapotról kiderüljön, hogy elfogadó, vagy nem elfogadó
2 Állítsa elő egy kezdeti H hipotézismodellt, mely ezt az egy állapotot tartalmazza
3 **while** *eq(H) ellenpéldával tér vissza* **do**
4 | **while** *Van kérdés a rendszer felé* **do**
5 | | Kérdezze mem(*seq*)-t a tanártól, ahol *seq* a kérdéses szó
6 | **end**
7 | Vizsgálja meg az ekvivalenciát: eq(H)
8 **end**
9 **return** H

2.4.2. Passzív automatatanulás

A passzív tanulás főként abban különbözik az aktív tanulástól, hogy annak nincs lehetősége kommunikálni a megtanulandó rendszerrel (2.9. ábra). Egy előre megkapott információhalmazból kell dolgoznia. 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(32. definíció) épít, majd azt megpróbálja determinisztikusan összevonni, minimalizálni[18].



2.9. ábra. Passzív automatatanulás

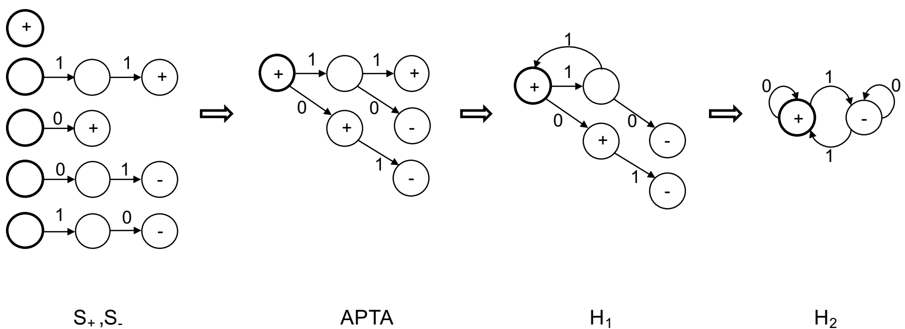
Definíció 32 (Lefutási fa). A lefutási fa, röviden APTA (Augmented Prefix Tree Acceptor) egy fa automata reprezentációja a tanulás során felhasznált pozitív és negatív elemeknek. Minden egyes elem, lefutás megtalálható a fában, annak egy útvonalaként, mely a fa gyökeréből indul. A pozitív és a negatív lefutásokhoz tartozó útvonalak végét jelző állapotok rendre elfogadóak vagy nem elfogadóak.
Az APTA így tulajdonképpen egy hiányos véges állapotgépnek felel meg, melynek nem minden állapotáról dönthető el, hogy elfogadó-e. ▪

Példa 4. *Legyen $\Sigma = \{0, 1\}$, vagyis az ábécé karakterei legyenek 0 és 1. X legyen továbbra is Σ^*, vagyis a karakterekből képezhető összes szó halmaza. Legyen a C által elfogadott nyelv azon x- ek halmaza, ahol $x \in X$ és amely x-ben az* 1*-esek száma kettővel osztható.*

A 4. példában leírt rendszernek a megtanulási folyamatát a 2.10. ábra szemlélteti. A tanulás első fázisában a tanuló megkapja a tanártól a pozitív és negatív lefutásokat. Ezek a pozitív és negatív lefutások most rendre legyenek 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 csökkentheti az automata méretét.



2.10. 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

1 $H = apta(S)$
2 **while** *van lehetőség összevonni* **do**
3 | Összevon két állapotot
4 **end**
5 **return** H

2.5. Tulajdonsággráf

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

A *tulajdonsággráfot* (*property graph*) a következő $G = (V, E, \mathsf{src_trg}, L_v, L_e, l_v, l_e, P_v, P_e)$ 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 rendeli hozzá. A csúcsok 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 *egyetlen típust* rendel.

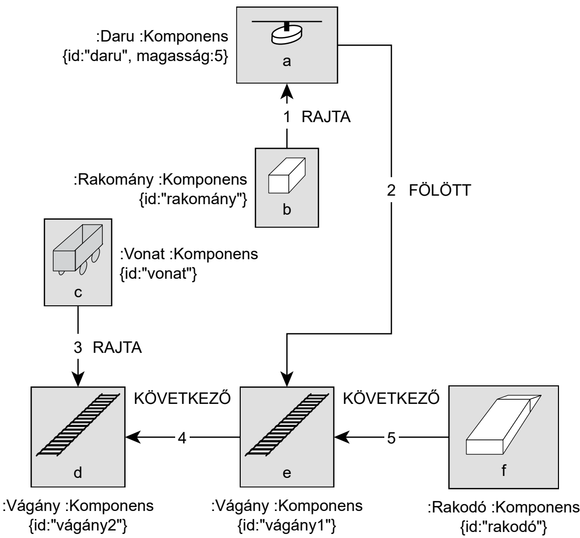
A tulajdonságok definiálásához $D = \cup_i D_i$ a D_i elemi tartományok uniója és NULL jelöli a NULL, üres értéket.

- 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 5. *A 2.11. 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 függőleges helyzetét a* magasság *tartalmazza (méterben). A gráf formálisan a 2.12. ábrán látható.*

2.6. Gráfminta

Gráfmintákat alkalmazunk, hogy a gráfban tárolt komplex információkból kigyűjtsük a vizsgálatunk szempontjából relevánsakat. A 2.13. ábrán látható egy ilyen gráfminta, amely egy adott $r paraméterhez megkeresi, hogy melyik Komponens tartalmazza. Így lehet meghatározni, hogy melyik vágányon vagy a darun van-e éppen az adott rakomány.

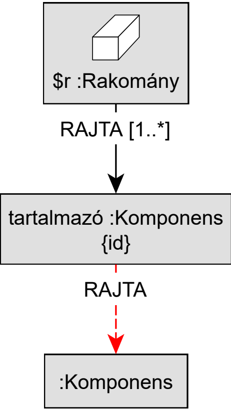


2.11. ábra. Rakodóállomás gráfmodellje

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

2.12. ábra. Rakodóállomás formális tulajdonséggráfja

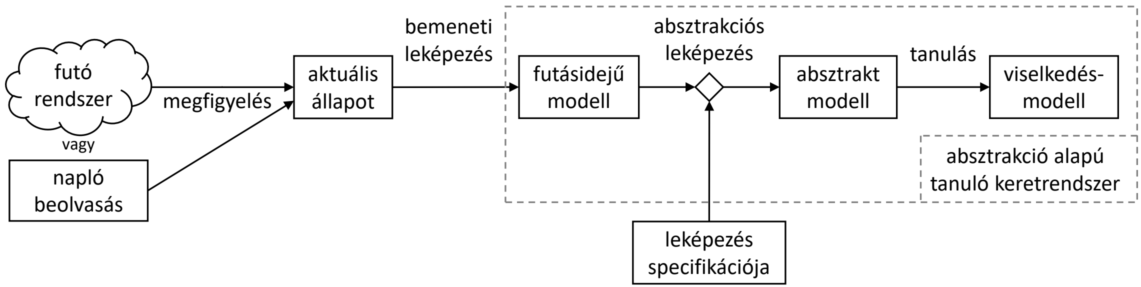
Keressük az olyan Komponens-eket, amelyek $r-ből egy vagy több RAJTA élen keresztül elérhetőek, de belőlük már nem megy ki RAJTA él Komponens címkéjű csúcsba (piros szaggatott él). A feltételeknek megfelelő csúcshoz tartozó id tulajdonságot szeretnénk megkapni.



2.13. ábra. Gráfminta példa

3. fejezet

A keretrendszer bemutatása



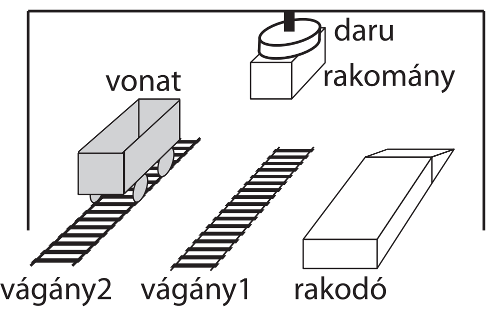
3.1. ábra. Absztrakció alapú tanuló keretrendszer

Az általunk megvalósított keretrendszer egy időzített rendszer működése során annak aktuális állapotából származtatott futásidejű modell alapján képes viselkedésmodellt alkotni (3.1. ábra). A bemenet egy gráf alapú futásidejű modell, amelyet célzott tanuláshoz használunk fel. A felhasználó gráfmintákkal specifikálhat egy absztrakciós leképezést. Az így képzett absztrakt modell felhasználható automatatanulásra, amely egy időzített viselkedésmodellt állít elő. Így lehetővé válik komplex, parametrikus rendszerek egy kiemelt részletének, az időfüggő viselkedést is figyelembe vevő vizsgálata.

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 gráf alapú tudásbázisban tartjuk nyilván. 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 különböző műveletek eltérő idők alatt hajtódnak végre, így a modellezés során az időbeliséget is figyelembe vesszük. 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 indított példa lefutás látható, ahol egy rakománnyal érkező vonatról egy másik, üres vonatra pakolja át a daru a rakományt, majd mindkettő elhagyja az állomást.

Időbélyeg	Változás		
8	vonat1	rakomány1-et be	A vonat1 a rakomány1-gyel bejött.
16	vonat2	be	
22	daru	rakodó→vágány1	A daru vízszintesen a rakodótól a vágány1 fölé ment.
26	daru	le	A daru függőlegesen a vonatig leereszkedett.
27	daru	felvesz: rakomány1	A daru felvette a rakományt.
31	daru	fel	
37	daru	vágány1→vágány2	
41	daru	le	
42	daru	lerak: rakomány1	A vágány2 fölött áll, így a vonat2-re rakta a rakományt.
46	daru	fel	
54	vonat2	rakomány1-et ki	
62	vonat1	ki	
Szabályos lefutás			Nem történt baleset, ezért szabályos a lefutás.

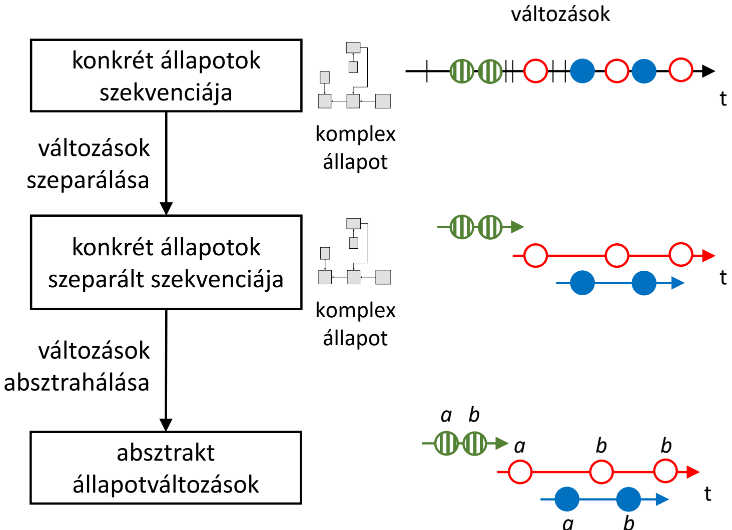
3.1. táblázat. Vasúti állomás egy lefutása

4. fejezet

Absztrakció

A rendszer megfigyelése során keletkező állapotszekvenciák önmagukban nem alkalmasak viselkedésmodell előállítására, mivel túl részletesek, ill. a vizsgálandó működés időben párhuzamosan több példányban is futhat. Továbbá egy komplex gráfmodell esetén az állapotok közti egyenlőség vizsgálata is nehézségekbe ütközik, azonban erre a tanulás során szükség van.

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

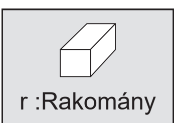


4.1. ábra. Absztrakció lépései

Megfigyelés Az állapot megfigyelése során a mintailleszkedések változásait szeretnénk észlelni. Ez úgy lehetséges, ha bizonyos időközönként lefuttatjuk az összes mintát. Amennyiben tudunk értesülni arról, hogy egy változás történt, akkor közvetlenül ez 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.

Lefutások szétválasztása A párhuzamosan futó folyamatok elkülönítésére egy 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 két vizsgálandó szakasz, a szeparálás segítségével két rövidebb szakaszra bontjuk, így ez a lépés a tanulás hatékonyságát is javítja.

Példa 6. *A vasúti esettanulmányban a rakományok lehetséges mozgását vizsgáljuk. Ahhoz, hogy az egy időben előforduló rakományokat megkülönböztessük, a lefutásokat rakomány szerint szétválasztjuk. A 4.2. ábrán látható minta illeszkedései a rendszerben aktuálisan előforduló rakományok. Egy illeszkedés megjelenésekor új szekvencia kezdődik, amely annak megszűnéséig tart.*



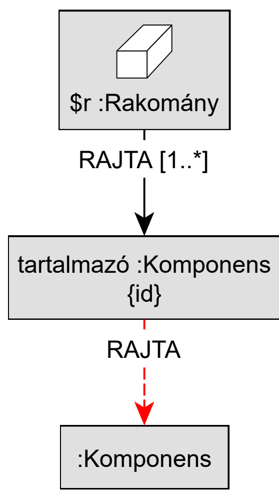
4.2. ábra. Szeparáló minta

Változások absztrahálása 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 már nem csúcsokat vagy éleket adnak vissza, hanem azok címkéit/típusát vagy bizonyos tulajdonságait. Ezek már elemi típusok (számok, logikai, *string*), így köztük egyszerűen elvégezhető az egyenlőségvizsgálat, amely a tanulás során szükséges. Ezt a lépést már a szeparált lefutásokon vizsgáljuk, így a minta felhasználhatja a szeparáló minta lekötéseit, így csak az ebben a lefutásban releváns részeket vizsgálja. Mivel a minták több vizsgált alkalommal is azonos eredményt adhatnak, ezért érdemes az aktuális illeszkedés helyett az illeszkedések változását figyelembe venni, és ezekből alkotni a tanulás során felhasználandó karaktereket.

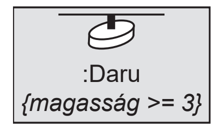
Példa 7. *Vizsgáljuk, hogy a rakomány a pálya mely elemén van. Felhasználhatjuk a szeparáló mintában az ehhez a lefutáshoz tartozó* r *rakományt paraméterként. A 4.3a. ábrán látható minta megállapítja, hogy az* $r *paraméterként kapott rakomány, milyen komponensen van. Azaz melyik az a* Komponens*, amely elérhető egy vagy több* RAJTA *élen keresztül, de belőle már nem megy ki* RAJTA *él egy* Komponens *címkéjű csúcsba. További absztrakciós lépésként megtehető, hogy az* tartalmazó.id *tulajdonság helyett a* l_v(tartalmazó) *címkehalmazt használjuk fel, ha a vizsgálat során elhanyagoljuk, hogy pontosan melyik vágányon van a rakomány, elég csak az, hogy valamelyik vágányon van.*

Emellett a daru függőleges helyzetét is szeretnénk vizsgálni. Ehhez felírtuk a 4.3b. ábrán szereplő mintát, amely csak akkor illeszkedik a darura, ha annak a $magasság *tulajdonsága 3 méternél nem kisebb. (A szűrő feltétel dőlt betűvel szerepel.)*

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 a lefutás, akkor egy gráfminta segítségével az osztályozás automatikusan elvégezhető. *Pozitív feltétel*



(a) Tartalmazó komponens keresése *(2.13. ábra)*

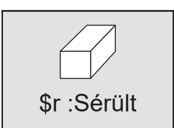


(b) Daru magasságának vizsgálata

4.3. ábra. Absztrakciós gráfminták

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

Példa 8. *A szeparált lefutásokat osztályozzuk aszerint, hogy történt-e baleset az adott rakománnyal. Az ilyen esetek a lefutásokban jelölve vannak. Negatív feltétel felírásával fogalmazható meg az osztályozás. A 4.4. ábrán látható minta, akkor illeszkedik a szeparált lefutás* $r *rakományára, ha azon van* Sérült *címke. Ekkor nem elfogadó a lefutás. Egyébként elfogadó.*



4.4. ábra. Osztályozó minta

Példa 9. *A 4.1. táblázatban láthatók az absztrakciós minták illeszkedései az esettanulmányban szereplő példán. Az egyes sorokban a gráfon történt változásokat is ábrázoltuk. Az első sorban megjelenik a* rakomány1*, ami illeszkedik a szeparáló mintára, így egy szekvencia kezdődik. Az absztrakciós minták illeszkedésének változásai láthatóak a két jobb oldali oszlopban. A szeparált szekvencia az 54. időegységig tart, itt eltűnik a rakomány1, így vége az adott szekvenciának. Mivel nem történt baleset a futás során, nem volt illeszkedése a negatív osztályozó mintának, így a szekvencia elfogadó.*

Idő-bélyeg	Változás		Változás a gráfban	**(a)** minta	**(b)** minta
8	vonat1	rakomány1-et be	{id:"rakomány1"} —RAJTA→ {id:"vonat1"} —RAJTA→ {id:"vágány1"}	→ vágány1	megjelenik
16	vonat2	be	{id:"vonat2"} —RAJTA→ {id:"vágány2"}		
22	daru	rakodó→vágány1	{id:"daru", magasság:5} —FÖLÖTT→ {id:"vágány1"}		
26	daru	le	{id:"daru", magasság:2}		eltűnik
27	daru	felvesz: rakomány1	{id:"rakomány1"} —RAJTA→ {id:"daru", magasság:2}	vágány1 → daru	
31	daru	fel	{id:"daru", magasság:5}		megjelenik
37	daru	vágány1→vágány2	{id:"daru", magasság:5} —FÖLÖTT→ {id:"vágány2"}		
41	daru	le	{id:"daru", magasság:2}		eltűnik
42	daru	lerak: rakomány1	{id:"rakomány1"} —RAJTA→ {id:"vonat2"}	daru → vágány2	
46	daru	fel	{id:"daru", magasság:5}		megjelenik
54	vonat2	rakomány1-et ki		szekvencia vége	
62	vonat1	ki			

4.1. táblázat. Absztrakciós minták illeszkedése a vasúti példán

5. fejezet

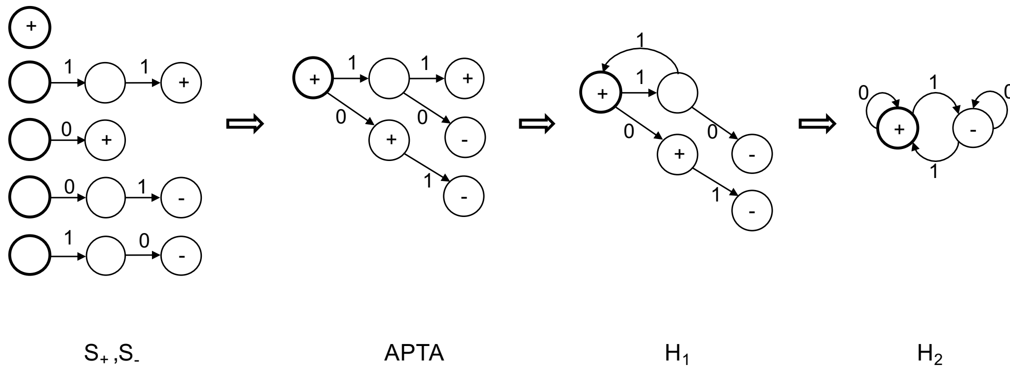
Automatatanuló algoritmus

Ebben a fejezetben bemutatjuk az általunk használt tanulóalgoritmus működését először az időzítés nélküli, majd az időzítéseket is figyelembe vevő esetre.

Munkánk során egy pozitív és negatív elemeket is felhasználó, passzív, offline tanulóalgoritmust használunk, amely képes előállítani valósidejű automatát, amelynek minden állapota vagy elfogadó vagy elutasító.

5.1. Algoritmus véges automatára

Munkánk során egy passzív tanuló keretrendszert valósítottunk meg. Ahogy arról már a 2.4.2. fejezetben szó volt, a passzív tanulás már meglévő lefutások halmazával dolgozik, azokból először egy APTA-t (32. definíció) készítve, majd azt összevonva azt egy véges állapotgéppé (9. definíció). A 5.1. ábrán látható egy áttekintő folyamatábra a 4. példában leírt rendszernek a megtanulásáról. Ebben a példában a cél az volt, hogy a páros számú 1-es karaktert tartalmazó szavakat fogadja el a megtanult automata.



5.1. ábra. Passzív tanulás folyamata

5.1.1. 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 $s_{i,j}i$ az i. lefutás j. állapotát jelöli, a az ábécé egy karakterét, q_i pedig az APTA i. 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 q0 = osszevon(∑_{i=0}^{k} s_{i,0})
 2 while ∃q_i, s_{a,j}, s_{b,j}, ..., s_{n,j} , amire 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 feltételezve azt, hogy a lefutásainkban nincs ellentmondás. Ezt követően töröljük a már összevont állapotokat.

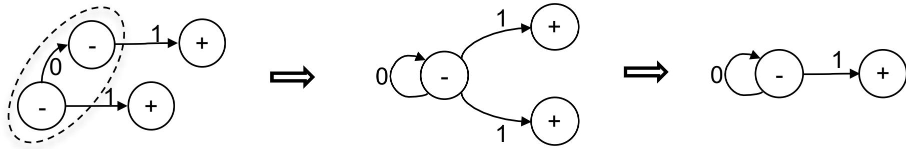
Algoritmus 4: Állapotok összevonása: *osszevon*

```
   Input: s1, s2, ...sn összevonandó állapotok
   Output: q állapot
 1 q állapot létrehozása
 2 for i ∈ s1, s2, ..., sn 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
```

5.1.2. Merge

A passzív tanuló algoritmus következő lépése, hogy az APTA-nak, ameddig csak tudja, csökkentse a méretét. Illetve, hogy 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* művelet

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

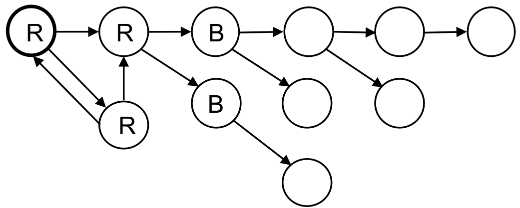
```
1  if q elfogadó és q' nem elfogadó, vagy fordítva then
2  |  return hamis
3  end
4  osszevon(q, q')
5  while H automatának van nemdeterminisztikus átmenete q_n és q'_n állapotokba do
6  |  boolean b = merge(q_n, q'_n)
7  |  if b hamis then
8  |  |  eddigi merge műveletek visszavonása
9  |  |  return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

5.1.3. Red-Blue eljárás

Annak érdekében, hogy a tanulás gyorsabb legyen a Red-Blue eljárást[18] 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



5.3. ábra. A Red-Blue algoritmus

a későbbiekben fog foglalkozni. Ebből az is következik, hogy az automatatanulás akkor fejeződik be, ha az automata minden állapota piros.

5.1.4. 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*.

5.1.5. Evidence driven state merging

Az automatatanuló algoritmustól azt is elvárjuk, hogy az általa megtanult automatának az állapottere minél kisebb legyen. Erre az evidence driven state merging[11][5] (EDSM) 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 $m = 0$ 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.

5.1.6. Az algoritmus

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

5.2. Algoritmus valósidejű automatára

A tanuló algoritmus valósidejű automatákra a véges automatákat megtanuló algoritmus kiegészítése. Különbség, hogy az algoritmus időzített lefutási fát használ, időzítetlen helyett. Ebben az időzített lefutási fában lehetnek inkonzisztens állapotok, vagyis olyan állapotok, melyek egyszerre elfogadóak és nem elfogadóak is. Ez akkor lehetséges, ha ugyan az a karaktersorozat más-más időzítéssel is szerepel az eredeti lefutásokban. Emiatt

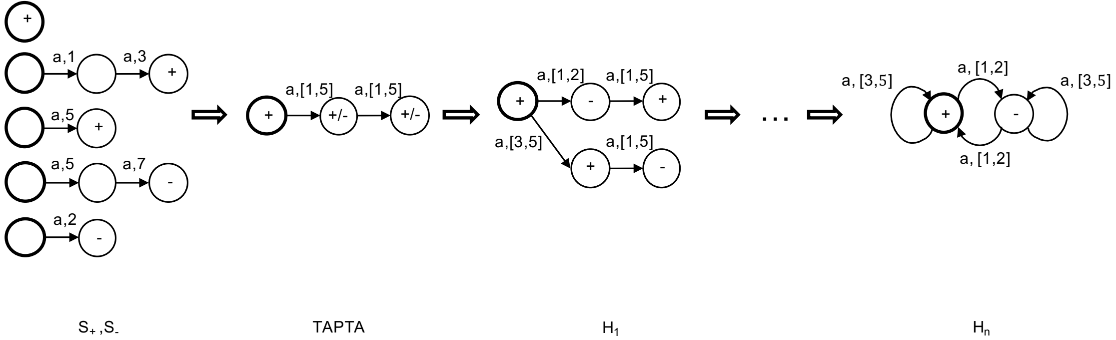
Algoritmus 6: DFA tanulóalgoritmus

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

```
1  H = apta(S)
2  q0, a kezdőállapot pirosra színezése
3  q0 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
```

az inkonzisztencia miatt időzített tanulás esetén szükséges bevezetni még egy műveletet. Ez az új művelet a *split*, amely fel tudja oldani az inkonzisztenciákat azáltal, hogy egy időzített élet felbont két újabb időzített élre (más időkorlátokkal). Az új művelet és az inkonzisztenciák miatt a metrikák számítása is módosul.

A 5.4. ábra szemléltet egy passzív valósidejű automatatanuló folyamatot. Az ábrában lévő algoritmus a 2. példában szereplő nyelvet tanulja. Ebbe a nyelvbe azok a szavak tartoznak, melyekben páros számú a karakter szerepel, legalább 1, legfeljebb 2 időközzel. A tanulás bemenete néhány pozitív és néhány negatív lefutás (S_+, S_-). Ezekből megépíti az időzített lefutási fát, majd *merge*, *split* és *color* műveleteket követően előállítja H_n végleges hipotézismodellt.



5.4. ábra. Valósidejű automata passzív tanulásának a folyamata

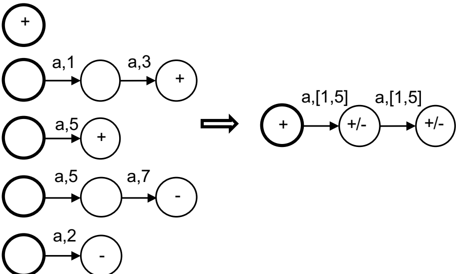
Ebben a fejezetben az eddig definiált *merge* és *color* műveletek felül lesznek definiálva az időzített megfelelőikre. Így innentől, mikor ezekre a műveletekre hivatkozunk, az időzített megfelelőjét értjük alatta.

5.2.1. TAPTA előállítása

Időzített tanulás esetén olyan lefutásaink vannak, melyek a bemeneti események idejéről is hordoznak információt. Éppen ezért ezekből a lefutásokból nem elég időzítetlen lefutási fát generálnunk, hiszen így információt veszítenénk. Emiatt az APTA egy módosított változatát, időzített lefutási fát generálunk.

Definíció 33 (Időzített lefutási fa). Az időzített lefutási fa, röviden TAPTA (Timed Augmented Prefix Tree Acceptor) egy időzített 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.
A fa állapotátmenetein az időkorlátok mind ugyan azok lesznek. Az időkorlát alsó határa a lefutásokban, azaz a $(a_1, t_1)(a_2, t_2)...(a_n, t_n)$ időzített szavakban előforduló legkisebb relatív időeltérés, a felső határa pedig a legnagyobb.
A TAPTA így tulajdonképpen egy hiányos valósidejű automatának felel meg, melynek nem minden állapotáról dönthető el, hogy elfogadó-e.

A 5.5. ábra az $S_+ = \{(\epsilon), (a, 1)(a, 2), (a, 5)\}, S_- = \{(a, 5)(a, 2), (a, 2)\}$ lefutásokból képzett TAPTA-t szemlélteti. Az időzített lefutási fa minden élén $[1, 5]$ időkorlát van, hiszen a lefutásokban a legkisebb előforduló relatív időeltérés 1, míg a legnagyobb 5 volt.



5.5. ábra. IIdőzített lefutási fa

A TAPTA tehát annyiban különbözik az APTA-tól, hogy állapotátmenetein időkorlátok is vannak. A TAPTA előállítását a 7. pszeudokód írja le.

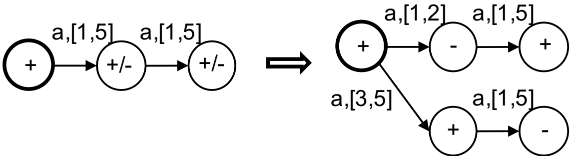
Algoritmus 7: TAPTA előállítása: *tapta*

Input: $S = \{S_+, S_-\}$ pozitív és negatív időzített lefutások halmaza (összesen k db)
Output: H_t időzített lefutási fa, amely tartalmazza S összes lefutását
1 $q_0 = osszevon(\sum_{i=0}^{k} s_{i,0})$
2 **while** $\exists q_i, s_{a,j}, s_{b,j}, ..., s_{n,j}$, *amire* $s_{a,j} = \delta(q_i, a),\ s_{b,j} = \delta(q_i, a),\ ...,\ s_{n,j} = \delta(q_i, a)$ **do**
3 | $osszevon(s_{a,j}, s_{b,j}, ..., s_{n,j})$
4 **end**
5 $T_{min} = S$ halmazban előforduló legkisebb időköz
6 $T_{max} = S$ halmazban előforduló legnagyobb időköz
7 $\forall d$ állapotátmenet időkorlátjának beállítása $[T_{min}, T_{max}]$-ra
8 **return** H_t

5.2.2. Split

A *split* művelet célja, hogy az automatát konzisztenssé tegye az inkonzisztens állapotokba vezető élek kettévágásával. A 5.6. ábrán látható egy példa a műveletre. A valósidejű automata eleinte két inkonzisztens állapotot is tartalmaz, majd a balról első élet $t = 2$ időpillanatban kettévágva megszűnik mindkét inkonzisztencia.

A 8. pszeudokód leírja a *split* művelet működését t időpillanatban. A művelet törli a kettévágandó állapotátmenetet és annak végállapotából kiinduló részfát. Majd két új



5.6. ábra. A *split* művelet

részfát hoz létre az eredeti lefutások felhasználásával. Végül pedig összeköti a megfelelő részfákat az eredeti él kiinduló állapotával. Ezeknek az összekötő éleknek az időkorlátai $[n, t]$ és $[t + 1, n']$ lesznek (amennyiben a kettévágott él időkorlátja $[n, n']$ volt).

Algoritmus 8: Split RTA tanulás során: *split*

Input: H_t RTA $d = \langle q, q', a, [n, n'] \rangle$ időzített állapotátmenete és $t \in [n, n']$ idő

1 d állapotátmenet törlése
2 q' állapotból induló részfa törlése
3 q_1 és q_2 új állapotok létrehozása
4 $d_1 \langle q, q_1, a, [n, t] \rangle$ állapotátmenet létrehozása
5 $d_2 \langle q, q_2, a, [t + 1, n'] \rangle$ állapotátmenet létrehozása
6 q_1 kezdőállapottal $tapta(S^{d_1})$
7 q_2 kezdőállapottal $tapta(S^{d_2})$

5.2.3. Merge

A *merge* művelet lényege ugyan az, mint időzítetlen esetben. Két állapot összevonására szolgál, illetve az ezáltal okozott nemdeterminizmusok megszüntetésére. Azonban a *split* művelet bevezetésével számos változtatást be kell vezetni a *merge*-ben.

- Mivel a *split* művelet lehetővé teszi az inkonzisztens állapotok konzisztensé alakítását, így *merge* esetén csak akkor kell feltétlen megtiltani az elfogadó és nem elfogadó állapotok összevonását, ha abból az egyik piros állapot (5.2.4. fejezet). Ez a változás az EDSM metrikaszámításánál fog majd szerepet játszani.
- A *split* művelet bevezetésével előfordulhat olyan eset, hogy *merge* során két, q és q' összevonandó állapot közül q-ból a karakterrel több állapotátmenet indul, különböző időkorlátokkal, míg q'-ből csupán egy. Ilyenkor a q'-ből a karakterrel induló állapotátmeneten is végre kell hajtani egy *split*-et.

Az 5.7. ábra szemléltet egy *merge* műveletet időzített esetben. A két összevonandó állapotból az időkorlátok szempontjából nem azonos állapotátmenetek indulnak. Így a *merge* végrehajtása előtt még végre kell hajtani egy *split* műveletet.

5.7. ábra. *merge* művelet során *split* végrehajtása

A 9. pszeudokód leírja a módosult *merge* működését. Az algoritmusban kikötjük, hogy a q' állapot nem piros, ez a kikötés a Red-Blue eljárás módosulásából következik (5.2.4. fejezet).

Algoritmus 9: Merge RTA tanulás során: *merge*

```
Input: H_t RTA két állapota: q, q'
Output: igaz, ha a merge lehetséges, hamis egyébként
1  if q vagy q' piros then
2  |  if q elfogadó és q' nem elfogadó, vagy fordítva then
3  |  |  return hamis
4  |  end
5  end
6  if ∃a, amire d1 = ⟨q, q1, a, [n, t]⟩ és d2 = ⟨q', q1, a, [n, n']⟩, ahol t < n' then
7  |  split(d2, t)
8  end
9  osszevon(q, q')
10 while H_t automatának van nemdeterminisztikus átmenete q_n és q'_n állapotokba do
11 |  boolean b = merge(q_n, q'_n)
12 |  if b hamis then
13 |  |  eddigi merge műveletek visszavonása
14 |  |  return hamis
15 |  end
16 end
17 return igaz
```

5.2.4. Red-Blue eljárás

A Red-Blue eljárásnak az lényege időzített esetben is szintén az algoritmus futásidejének csökkentése. Az eljárás garantálja azt, hogy a korábbi lépésekben csak a piros állapotok voltak módosítva, illetve hogy a már pirosra színezett állapotok nem fognak változni a tanulás további lépéseiben.

Éppen ezért nincs megengedve a *merge* művelet esetén a piros állapotok inkonzisztens állapotba hozása. Illetve az eljárást garantálja, hogy csak piros élekből kimenő éleken lehetett korábban *split* művelet végrehajtva. Így például mikor *merge* esetén vizsgáljuk, hogy szükséges-e *split* művelet, csak a piros állapotokból induló állapotátmeneteket kell figyelni.

5.2.5. Color

A *color* művelet megegyezik az időzítetlen változatával. Ha az értékelődik a legjobbnak, akkor egy kék állapotot pirosra színez az algoritmus.

5.2.6. Evidence driven state merging

A *split* művelet sok változást hoz a többi művelet metrikaszámításában is. Időzített esetben a következőképpen alakul az m metrika kiszámítása. Minden esetben m értékét 0-ra inicializálja az algoritmus, majd az alábbiaknak megfelelően módosítja azt.

- **Merge:** A *merge* művelet során összevonandó q_i, q'_i állapotpároktól függően

- ha q_i elfogadó és q_i' nem elfogadó, vagy fordítva és q_i vagy q_i' piros, akkor a metrika $m = -\infty$ lesz
- ha q_i elfogadó és q_i' nem elfogadó, vagy fordítva, akkor m értéki csökkenti 1-gyel
- 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.

- **Split:** A *split* művelet során kettébontott q_i, q_i' állapotpároktól függően
  - ha q_i elfogadó és q_i' nem elfogadó, vagy fordítva, akkor m értéki növeli 1-gyel
  - ha q_i és q_i' is elfogadó, vagy mindkettő nem elfogadó, akkor m értékét csökkenti 1-gyel
  - q_i vagy q_i' nem ismert, hogy elfogadó-e, akkor nem változtat m értékén
  - ha q_i vagy q inkonzisztens marad, akkor nem változtat m értékén.
- **Color:** A *color* művelet minden esetben $m = 0$ értékű.

5.2.7. Az algoritmus

Az általunk megvalósított időzített rendszerekre alkalmazható passzív automatatanuló algoritmust a 10. pszeudokód írja le.

Algoritmus 10: RTA tanulóalgoritmus

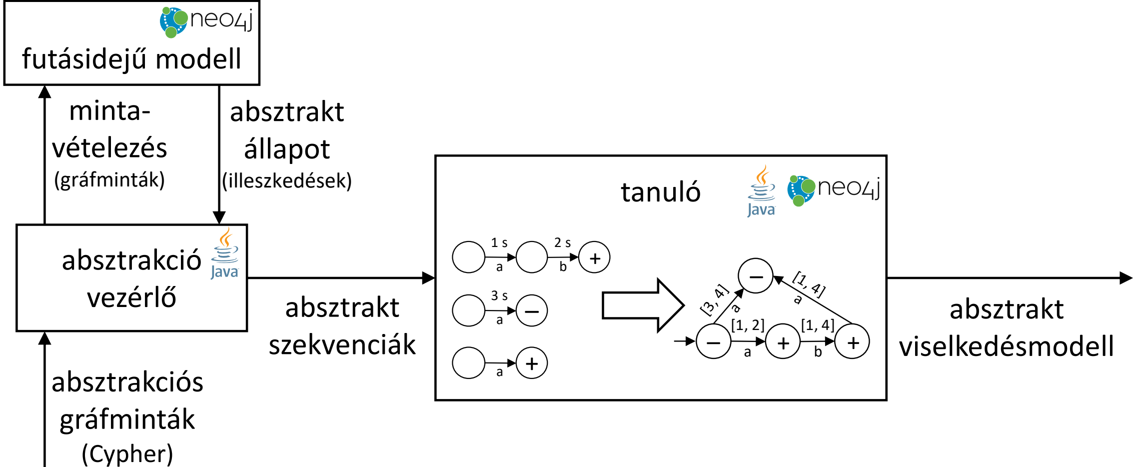
```
   Input: S = {S+, S−} pozitív és negatív időzített lefutások halmaza
   Output: Ht kis állapotterű RTA, melyre igazak S elemei
 1 Ht = tapta(S)
 2 q0, a kezdőállapot pirosra színezése
 3 q0 közvetlen gyerekeinek kékre színezése
 4 while Ht automatában van még nem piros állapot do
 5 |  Lehetséges merge műveletek metrikájának a kiszámítása
 6 |  Lehetséges split műveletek metrikájának a kiszámítása
 7 |  Lehetséges color műveletek metrikájának a kiszámítása
 8 |  if merge(r, b) rendelkezik a legmagasabb pozitív metrikával then
 9 |  |  merge(r, b)
10 |  end
11 |  if split(d, t) rendelkezik a legmagasabb pozitív metrikával then
12 |  |  split(d, t)
13 |  end
14 |  if color() rendelkezik a legmagasabb pozitív metrikával then
15 |  |  color()
16 |  end
17 |  Új piros állapotok gyerekeinek kékre színezése
18 end
19 return Ht
```

6. fejezet

Megvalósítás

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

6.1. Architektúra



6.1. ábra. A keretrendszer felépítése

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

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 gráfminta-illesztéssel 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 paramétereit 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.

A változások absztrahálása során az absztrakciós mintákat futtatjuk. Minden szeparált szekvencia esetén a szétválasztás során rögzített illeszkedést mint paraméter használhatjuk fel a mintában. A minták több frissítés során is illeszkedhetnek, így az illeszkedések megjelenését vagy eltűnését rögzítjük. A lekérdezésektől elvárt, hogy egy adott 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.

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.

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 során a Neo4j gráfadatbázison dolgozunk. Cypher lekérdezések segítségével hajtunk végre az adatbázison lekérdezéseket, illetve módosítjuk is

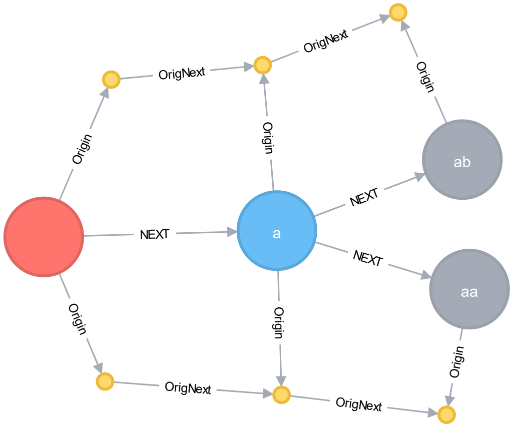
azt. Az algoritmus (10. pszeudokódnak megfelelő) logikája Java nyelven van megvalósítva. Azonban ennek egyes részletei gráflekérdezéseket takarnak. Így tehát a tanulóalgoritmus megvalósítása két nagyobb részre bontható.

- **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 *tapta* előállításához szükséges lekérdezések hajtódjanak végre. Majd a piros és kék állapotokat beszínező lekérdezések. Ezt követően pedig folyamatosan kérdezze le az egyes *merge*, *split* és *color* műveletek metrikáit, majd válassza ki ezek közül a legjobbnak bizonyulót és hajtsa is végre. Mely végrehajtások szintén módosító adatbázislekérdezéseket 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 Neo4j 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 újratöltésénél játszik szerepet.



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

Egy-egy művelet megvalósítása általában több lekérdezés hívásából épül fel. Egy *split* művelet végrehajtásához például először le kell kérdezni, hogy melyik piros állapotból melyik kék állapotba menő állapotátmeneten, milyen idővel kell azt végrehajtani. Ezt a lekérdezést a split metrikaszámító lekérdezése hajtja végre. Amely azt is eredményül

adja még, hogy a szekvenciákban szereplő állapotok közül melyikből melyik részfát kell majd generálni. Majd egy másik módosító lekérdezéssel törölni kell a kék állapotból induló részfát. Egy harmadikkal legenerálni az új részfákat, egy utolsóval pedig állapotátmeneteket generálni a piros állapotból, megfelelő időkorlátokkal, a részfákba.

Dolgozatunkban szeretnénk bemutatni egy összetettebb adatbázislekérdezést. Erre a *split* metrikájának a kiszámoló lekérdezését választottuk ki. Egy ilyen lekérdezés az algoritmusban akkor futhat le, mikor már van egy kezdetleges hipotézismodellünk. Ebben a hipotézismodellben egyaránt vannak piros és kék állapotok is. A java kódból ez a lekérdezés akkor hívódik, mikor szeretnénk kiszámolni, hogy melyik műveletet lenne a legérdemesebb végrehajtani, minek a legmagasabb a metrikája.

Split művelet metrikájának kiszámítása

A split metrikájának kiszámításához szükséges teljes lekérdezést a 6.1. lista szemlélteti.

```
// maxSplitScore
MATCH (red:Red)-[n:NEXT]->(blue:Blue)
WITH red, blue, n.Tmax as Tmax, n.Tmin as Tmin, n.symbol as symbol
UNWIND range(Tmin, Tmax-1) AS t
MATCH (blue)-[:NEXT*0..]->(s1:State)-[:Origin]->(so1:OrigState)<-[trace1:OrigNext*0..]-()<-[no1:
     OrigNext]-(:OrigState)<-[:Origin]-(red)
MATCH (blue)-[:NEXT*0..]->(s2:State)-[:Origin]->(so2:OrigState)<-[trace2:OrigNext*0..]-()<-[no2:
     OrigNext]-(:OrigState)<-[:Origin]-(red)
WITH t, r, b, s1, s2, so1, so2, no1.time>t as cat1, no2.time>t as cat2, Tmin, Tmax, symbol
WHERE s1 = s2
  AND cat1 <> cat2
  AND id(so1) < id(so2)
WITH r, b, t, s1, so1, so2, Tmin, Tmax, symbol,
  [so1.accepting, so1.rejecting] AS so1ar,
  [so2.accepting, so2.rejecting] AS so2ar
WITH
  r, b, t, so1, so2, Tmin, Tmax, symbol,
  CASE
    WHEN so1ar[0] <> so2ar[0] AND so1ar[1] <> so2ar[1] THEN 1
    WHEN so1ar =  so2ar AND (so1ar = [false, true] OR so1ar = [true, false]) THEN -1
    ELSE 0
  END AS score
WITH r, b, t, collect(so1) as so1s, collect(so2) as so2s, sum(score) AS metric, Tmin, Tmax, symbol
ORDER BY metric DESC
LIMIT 1
RETURN r, b, t, metric, so1s, so2s, Tmin, Tmax, symbol
```

6.1. lista. A legnagyobb *split* metrika kiszámítása.

Ez a lekérdezés az valósidejű automatában a legmagasabb értékű *split* művelet paramétereit adja vissza. Ezáltal a későbbiekben, ha ez a művelet bizonyul a legjobb metrikájúnak, akkor felhasználva a paramétereket végre lehet hajtani a *split* műveletet. Fontos megjegyezni, hogy a *split* metrikájának a kiszámításához végre is kell hajtani a műveletet, majd a metrika kiszámítását követően visszavonni. Az alábbiakban kifejtjük a lekérdezés egyes részeit.

A lekérdezés felső három sora (6.2. lista) arra szolgál, hogy kiválogassuk a lehetséges d állapotátmeneteket a $split(d, t)$ művelethez.

```
MATCH (red:Red)-[n:NEXT]->(blue:Blue)
WITH red, blue, n.Tmax as Tmax, n.Tmin as Tmin, n.symbol as symbol
UNWIND range(Tmin, Tmax-1) AS t
```

6.2. lista. d állapotátmenet és t idő kiválasztása.

Ezek az állapotátmenetek egy *red* piros állapotból egy *blue* kék állapotba kell, hogy mutassanak, ennek lekérdezésére szolgál az 1. sor. A továbbiakban szükségünk lesz

mindkét állapotra, illetve a közöttük lévő állapotátmenet minden tulajdonságára: a *symbol* karakterre, illetve az időkorlátra (2. sor).

Ha a lekérdezésünk csupán ez a két sor lenne és az állapotokkal, valamint az állapotátmenet tulajdonságaival rögtön visszatérne, akkor a 6.1. táblázathoz hasonló eredményt kapnánk (aktuális automatától függően).

red	blue	Tmax	Tmin	symbol
{id: n1, trace:}	{id: n2, trace: b}	1	5	b
{id: n3, trace: a}	{id: n5, trace: ab}	1	5	b
{id: n3, trace: a}	{id: n6, trace: aa}	1	5	a

6.1. táblázat. A Cypher lekérdezés eredményeként kapott reláció

A lekérdezés azonban folytatódik, végül meg kell vizsgálni a $split(d, t)$ művelet értékét adott d állapotátmenet esetén minden lehetséges t-re. Az alsó és a felső időkorlát közé eső t idők lehetségesek *split* szempontjából, ahogy az a 3. sorban látható.

```
MATCH (blue)-[:NEXT*0..]->(s1:State)-[:Origin]->(so1:OrigState)<-[trace1:OrigNext*0..]-()<-[no1:
    OrigNext]-(:OrigState)<-[:Origin]-(red)
MATCH (blue)-[:NEXT*0..]->(s2:State)-[:Origin]->(so2:OrigState)<-[trace2:OrigNext*0..]-()<-[no2:
    OrigNext]-(:OrigState)<-[:Origin]-(red)
```

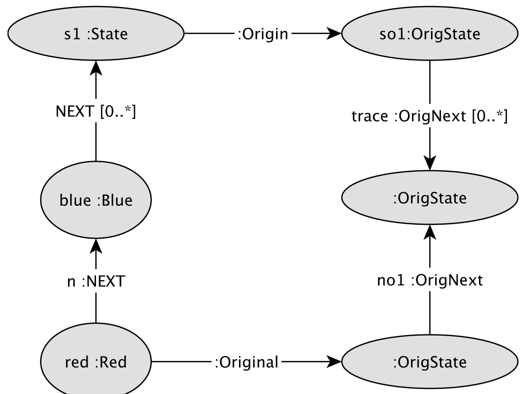
6.3. lista. Az RTA egy részének az eredeti S lefutásainak megkeresése.

A lekérdezés következő pár sora (6.3. lista) arra szolgál, hogy a kettévágott élet követő részfának az állapotaihoz megkeressük az eredeti S lefutásokban az ősét. Vagyis a lefutások azon részeit, amiből az adott részfa generálódott. Fontos megjegyezni itt, hogy egy-egy állapot a részfában több különböző lefutásnak is lehetett az állapota, melyek az APTA generálása során összevonódtak. A *split* műveletnek éppen az a lényege, hogy az ilyen állapotokat két külön kategóriára bontsa. Ezek közül az egyik kategória a *split*-et követően az egyik, míg a másik kategória a másik részfába fog kerülni.

A 6.3. ábra szemlélteti a gráfmintának ezt a két sorát. A *red* piros állapotból a *blue* kék állapotba mutató kettévágandó n állapotátmenet alapján keressük az illeszkedéseket olyan $s1$ állapotokra, melyek *blue* állapotból induló részfának az állapotai (*blue*-t is beleértve). Ezen $s1$ állapotok valamelyik (esetleg több) lefutás egy $so1$ állapotából generálódtak. Ezekbe az $so1$ állapotokba pedig az eredeti lefutásokban vezetett út a *red* piros állapot őséből, amely útnak az első állapotátmenete $no1$ volt. Vagyis az $so1$ állapotot tartalmazó lefutás egy korábbi állapotából *red* generálódott.

Az illeszkedés sora a 6.3. listában azért van megduplázva, hogy ha egy $s1$ állapothoz tartozóan több $so1$ is tartozna (vagyis $s1 = s2$ de $so1 \neq so2$), akkor azoknak a párba állításával ki lehet számítani a metrikát.

A 6.4. listában lévő sorok felelősek azért, hogy az állapotok ősét ($so1$ és $so2$) kategóriákba osszuk. A kategóriák lényege az, hogy a *split* során melyik keletkező részfa generálásában fog részt venni az adott állapot. A kategóriákat pedig az alapján számítjuk, hogy a korábban lekérdezett $no1$ és $no2$ állapotátmeneten szereplő *time* idő nagyobb-e a *split* művelet t időparaméterénél. Ez a *time* idő az előző állapotba lépéstől számított eltelt időt jelenti. A metrika kiszámításához olyan $no1$ és $no2$ párokra van szükségünk, amelyekből ugyanaz az állapot generálódtt, vagyis $s1 = s2$. Ezen kívül, $no1$ és $no2$ állapotoknak külön részfába kell kerülniük ahhoz, hogy a *split* metrikáját módosítsák, tehát más kategóriába kell tartozzanak. Az ezt követő sorban $no1$ és $no2$ egyedi azonosítóját (id) csak azért hasonlítjuk össze, hogy egy állapotpárt csak egyszer adjon eredményül a lekérdezés.



6.3. ábra. Eredeti S lefutások megkeresése

```
WITH t, r, b, s1, s2, so1, so2, no1.time>t as cat1, no2.time>t as cat2, Tmin, Tmax, symbol
WHERE s1 = s2
    AND cat1 <> cat2
    AND id(so1) < id(so2)
```

6.4. lista. Az eredeti lefutások t szerint kategóriákba rendezése.

A lekérdezés következő részlete (6.5. lista) végzi az állapotpárokra a metrika módosításának a kiszámítását. A 5.2.6. fejezetben leírt elvek alapján.

```
WITH r, b, t, s1, so1, so2, Tmin, Tmax, symbol,
    [so1.accepting, so1.rejecting] AS so1ar,
    [so2.accepting, so2.rejecting] AS so2ar
WITH
    r, b, t, so1, so2, Tmin, Tmax, symbol,
    CASE
        WHEN so1ar[0] <> so2ar[0] AND so1ar[1] <> so2ar[1] THEN 1
        WHEN so1ar =  so2ar AND (so1ar = [false, true] OR so1ar = [true, false]) THEN -1
        ELSE 0
    END AS score
```

6.5. lista. A *split* során szétszedett q és q' állapotokhoz tartozó metrika változtatásának a kiszámítása.

Végül pedig a lekérdezés utolsó pár sora rendezi a lehetséges *split* műveleteket a metrikájuk alapján. Rendezés után az egyetlen legmagasabb értékű *split* művelet paramétereivel visszatér. Így a későbbiekben a paraméterek felhasználásával meg tud történni a *split* végrehajtása.

```
WITH r, b, t, collect(so1) as so1s, collect(so2) as so2s, sum(score) AS metric, Tmin, Tmax, symbol
ORDER BY metric DESC
LIMIT 1
RETURN r, b, t, metric, so1s, so2s, Tmin, Tmax, symbol
```

6.6. lista. m metrikák kiszámítása és a legnagyobb kiválasztása.

7. fejezet

Kapcsolódó munkák

Ebben a fejezetben áttekintjük az irodalomban ismert kapcsolódó munkákat. Előző évi TDK munkánk [9] is hasonló témát érintett, így az első kettő kapcsolódó munka abban is megtalálható.

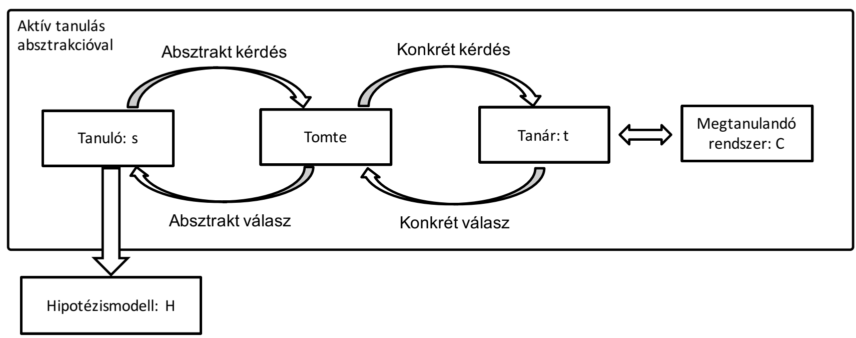
7.1. LearnLib keretrendszer aktív automatatanulásra

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. Illetve az időzített rendszerek kezelésére sem tartozik a felhasználási körébe.

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 LearnLib. 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.

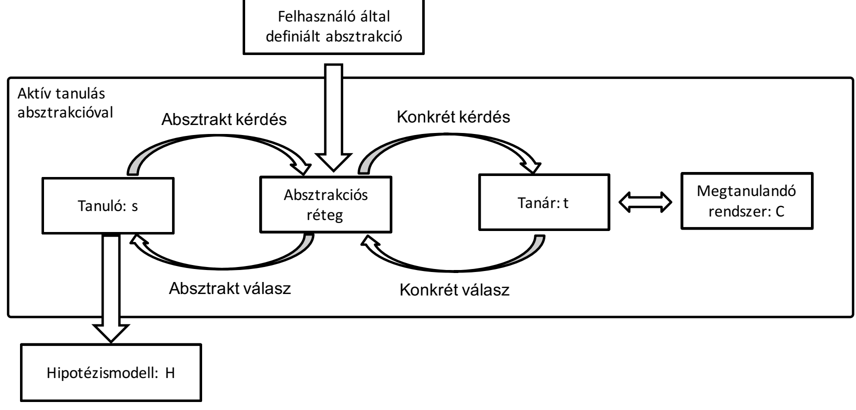
Az eszköz először egy általános absztrakciót definiál, majd ha a tanulás során nemdeterminizmust tapasztalna, akkor a nemdeterminizmust okozó ellenpélda mentén automatikusan finomítja az absztrakciót [6]. A folyamat végén a megtanult automata kellően pontosan fogja tükrözni a valós rendszer működését.

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 TDK munkánkban [9] 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.



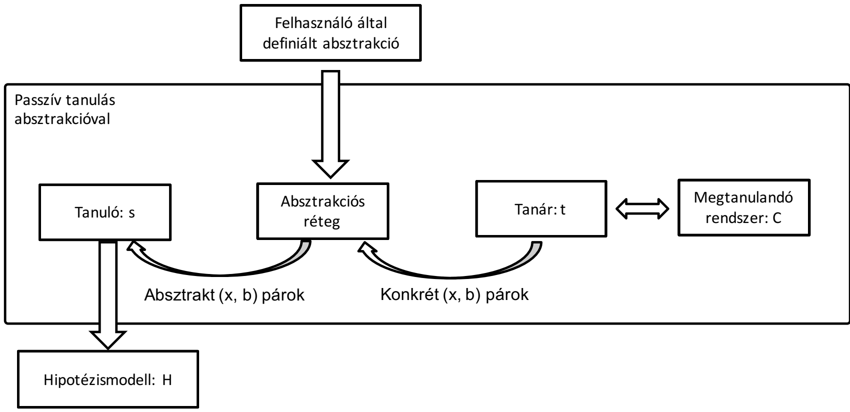
7.2. ábra. Aktív automatatanulás testreszabott absztrakcióval

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 hiányossága még, hogy időzített rendszereket sem tud megfelelően kezelni.

7.4. Modellalapú automatatanulás formális modellek szintéziséhez

Idei TDK munkánk 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.

8. fejezet

Összefoglaló

Munkánk célja komplex rendszerek formális specifikációjának előállítása volt időzített automata formájában. Munkánk során egy olyan módszert fejlesztettünk, amely képes kezelni kritikus kiberfizikai rendszerek adatfüggő, gráf jellegű és időzített viselkedéseit is.

Az általunk 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 időzített rendszerek tanulását lehetővé tevő, időzített automata szintézis algoritmust illesztettünk.
Elméleti eredményeink az alábbiak:

- Kidolgoztunk egy módszert, amely az adat és gráf jellegű viselkedések absztrahálásán keresztül támogatja kiberfizikai rendszerek tanulását.
- Gráfminta alapú nyelvet javasoltunk az automatatanulás támogatására, amely nyelven egyrészt a bemeneti adatok specifikálhatóak, továbbá az absztrakció és a paraméterek kezelése definiálható, ezáltal támogatva a tanulás fókuszálását.
- Időzített automata tanuló algoritmussal kombináltuk a gráfmintaillesztő rendszert az időzített automata modellek szintézisének támogatására.

Gyakorlati eredményeink az alábbiak:

- Megvizsgáltuk kritikus kiberfizikai rendszerek automata tanuláson alapuló analízisének lehetőségét.
- Implementáltuk az irodalomban található, ám elérhető implementációval nem rendelkező időzített automata tanuló algoritmust. Az algoritmust kiegészítettük több helyen is saját ötleteinkkel.
- Implementáltuk a keretrendszer prototípusát.

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

Munkánk során példák segítségével vizsgáltuk megközelítésünk gyakorlati működését és használhatóságát.

8.1. Jövőbeli munka

A jövőben dolgozni fogunk, hogy a rendszer skálázódását és korlátait is megvizsgáljuk. Ehhez tervezzük komplexebb ipari esettanulmányok felhasználását. Emellett a meglévő automata tanuló algoritmuson kívül egyéb tanuló algoritmusokat is tervezünk a keretrendszerbe integrálni, így várhatóan a rendszerek tágabb körét tudjuk majd vizsgálni.

Érdekes elméleti továbbfejlesztési lehetőség a nem megfelelő absztrakciók automatikus finomítási lehetőségének megvizsgálása, amellyel egy CEGAR [7] jellegű tanuló algoritmust készíthetnénk, 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 támogassuk, hogy inkrementális algoritmusokat használnánk 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 [17].

Köszönetnyilvánítás

Emberi Erőforrások Minisztériuma

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 Vaandrager: Automata learning through counterexample guided abstraction refinement. In *International Symposium on Formal Methods* (konferenciaanyag). 2012, Springer, 10–27. p.

[2] Rajeev Alur – David L Dill: A theory of timed automata. *Theoretical computer science*, 126. évf. (1994) 2. sz., 183–235. p.

[3] Dana Angluin: Learning regular sets from queries and counterexamples. *Information and computation*, 75. évf. (1987) 2. sz., 87–106. p.

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

[5] Miguel Bugalho – Arlindo L Oliveira: Inference of regular languages using state merging algorithms with search. *Pattern Recognition*, 38. évf. (2005) 9. sz., 1457–1467. p.

[6] Edmund Clarke – Orna Grumberg – Somesh Jha – Yuan Lu – Helmut Veith: Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification* (konferenciaanyag). 2000, Springer, 154–169. p.

[7] Edmund Clarke – Orna Grumberg – Somesh Jha – Yuan Lu – Helmut Veith: Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification* (konferenciaanyag). 2000, Springer, 154–169. p.

[8] Catalin Dima: Real-time automata. *Journal of Automata, Languages and Combinatorics*, 6. évf. (2001) 1. sz., 3–24. p.

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

[10] John E Hopcroft – Rajeev Motwani – Jeffrey D Ullman: Automata theory, languages, and computation. *International Edition*, 24. évf. (2006).

[11] KJ Lang – BA Pearlmutter – RA Price: Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. lnai, 1433: 1–12, 1998. In *4th International Colloquium, ICGI-98* (konferenciaanyag).

[12] Alexander Maier: Online passive learning of timed automata for cyber-physical production systems. In *Industrial Informatics (INDIN), 2014 12th IEEE International Conference on* (konferenciaanyag). 2014, IEEE, 60–66. p.

[13] József Marton – Gábor Szárnyas – Dániel Varró: Formalising openCypher graph queries in relational algebra. In *ADBIS* (konferenciaanyag). 2017, 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/`, 2016.

[15] Neo Technology: Neo4j. `http://neo4j.org/`, 2016.

[16] Harald Raffelt – Bernhard Steffen – Therese Berg: Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems* (konferenciaanyag). 2005, ACM, 62–71. p.

[17] 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 & Systems Modeling*, 15. évf. (2016. Jul) 3. sz., 609–629. p. ISSN 1619-1374.
URL `https://doi.org/10.1007/s10270-016-0530-4`.

[18] 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.

[19] Sicco Ewout Verwer: *Efficient identification of timed automata: Theory and practice.* PhD értekezés (TU Delft, Delft University of Technology). 2010.