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

Absztrakcióval támogatott tanulás regressziós tesztelés támogatására

TDK-dolgozat

Készítette:

Gujgiczer Anna
Elekes Márton Farkas

Konzulens:

Vörös András
Semeráth Oszkár

2016

Tartalomjegyzék

Kivonat

Napjaink komplex, elosztott és kritikus rendszerei új kihívások elé állítják a rendszermérnököket. Ez különösen igaz az utóbbi időben, amikor a gyorsan változó igények miatt a rendszerek újrakonfigurálása, áttervezése is a napi rutin részévé vált. Ez a gyors fejlődés azonban problémákat vet fel, fontos kérdéssé vált a rendszerek új állapotának az ellenőrzése: vajon a változtatások elrontották-e a rendszer viselkedését. Ilyenkor nyújt megoldást a regressziós tesztelés, ahol azt vizsgáljuk, hogy a rendszer változásai okoznak-e az eddigi viselkedéstől való nem kívánt eltérést.

A regressziós tesztelés támogatása fontos, de nehéz probléma. Sokat segít a regressziós tesztelésben, ha a rendszernek elkészül egy modellje, amely az elvárt viselkedéseket tartalmazza. Ezen modell elkészítésére azonban a fejlesztőknek gyakran nincs ideje, utólag pedig csak jóval költségesebben, a forráskódok és a konfigurációs fájlok visszafejtésével lehet előállítani.

Munkánk célja egy olyan módszer kidolgozása, amely képes a regressziós tesztelés támogatására specifikációs modellek automatikus szintézisével. A megközelítés során az egyes komponenseket külön-külön próbáljuk megtanulni és a komponensmodelleket előállítani. A tradicionális automatatanuló algoritmusok erre nem bizonyultak alkalmasnak, mivel nem kezelik a rendszer adatfüggő viselkedéseit, ezért kidolgoztunk egy absztrakciós keretrendszert, amely lehetővé teszi a komponens-bemenetek konfigurálható absztrakcióját és a kimeneti adatok szintézisét, ezáltal támogatva a rendszer komponenseinek tanulását. Az így előállított modellek alapján teszt szekvenciákat generálunk, amelyek a fejlesztés későbbi fázisaiban alkalmazhatóak regressziós tesztelésre. Amennyiben a regressziós tesztelés sikeres, a specifikációs automatát az új rendszerből megtanulva a két automata ekvivalenciáját az általunk fejlesztett keretrendszer ellenőrzi.

Dolgozatunkban bemutatjuk az általunk kidolgozott keretrendszert, amely nyelvi támogatást ad konfigurálható módon absztrakciók definiálására, majd ennek megfelelően megvalósítja a komponensek bemeneteinek absztrakcióját és a kimenetek konkretizációját. Emellett támogatja a megtanult modell alapján különböző feltételeknek megfelelő teszt szekvenciák generálását. A keretrendszer képes ellenőrizni a rendszer különböző fejlesztési fázisokban megtanult modelljeit ekvivalencia-ellenőrzéssel, így támogatva a különböző szoftververziók összehasonlítását és a változtatások ellenőrzését.

A módszer segítségével hatékonyan támogatjuk a regressziós tesztelést. Munkánk által lehetővé válik olyan szoftverkomponensek regressziós tesztelése, amelyekhez eddig nem állt rendelkezésre a működést leíró modell.

Abstract

Nowadays, the development of complex, distributed and critical systems yield a huge challenge to systems engineers. Ensure the correct behaviour is especially difficult in evolving environments where the frequent changes in demands leads to the frequent reconfiguration and redesign of the systems. This rapid evolution raises additional problems. It has become important to verify the new state of the system, whether it satisfies the original specification at least as much as the previous version. Regression testing is a solution, as it searches for unintended divergence in the behaviour caused by the changes in the system.

Supporting regression testing is an important though difficult problem. Creating a model for the desired behaviour of the system contributes to regression testing. However developers usually do not have time to construct the specification model and it is costly to reconstruct afterwards from the source and configuration files.

The goal of our work is to develop a methodology to support regression testing with automatic synthesis of specification models. Our approach learns each component separately and produces behaviour models from them. Traditional automata learning algorithms proved to be insufficient for this task, as they do not deal with the data-dependent behaviour of the system. Therefore we developed a language framework to facilitate configurable abstractions of the inputs and the synthesis of the outputs for each component: our solution supports the learning of the model of even complex components. Based on these models we generate test sequences to be applied for regression testing in a later phase of development. In case the regression test succeeds, our framework could perform an equivalence test between the specification automata learnt from the new and the former system implementation.

In this report we introduce our framework which gives us a language to define and configure abstractions. Accordingly, it performs the abstraction of the inputs and concretization of the outputs in the components. Moreover it supports the generation of test sequences based on the learnt model and additional criteria. Also it is able to verify the learnt models from each development phase of the system with equivalence check by comparing the different versions of the software model and verifying the changes.

Our method provides an efficient support for regression testing of software components without the manual construction of specification models describing their behaviour.

1. fejezet

Bevezető

Szoftverek minőségének biztosítása komplex feladat, amely során komoly szerep hárul a tesztelésre. A tesztelésnek sok fajtája létezik, melyek közül egy szoftverkomponens tesztelésére az egységtesztelést (unit test) használjuk. Ilyenkor a komponens számára bemeneteket adunk és a válaszok/kimenetek megfigyelésével vizsgáljuk, hogy helyesen működik-e a szoftver. Munkánk során szoftverkomponensek tesztelésével foglalkozunk.

Egy szoftver soha sincs kész, folyamatosan fejlődik, változik. A hibákat ilyen esetekben is meg kell találni. Regressziós tesztelést (Regression testing) alkalmazunk olyan esetekben, mikor egy meglévő rendszer komponenseiben változtatások történtek. Ilyenkor érdemes megbizonyosodni arról, hogy a rendszer változatlan funkciói továbbra is működőképesek maradtak, az eredeti specifikációnak továbbra is megfelel. Módosítás lehet a kisebb optimalizálástól kezdve egészen odáig, hogy egy másik programnyelven készítünk el a komponenst. Emellett ha egy új funkciót teszünk hozzá a szoftverünkhöz, a korábbi funkcióknak változatlanul működnie kell. Ilyen esetben tud segíteni a regressziós tesztelés, amely során ellenőrizzük, hogy a módosítások, fejlesztések nem vittek-e be hibát, akaratlan módosítást a szoftver funkcióiba.

A regressziós tesztelés során azt vizsgáljuk, hogy a szoftver módosítása és fejlesztése során akaratlanul nem változtattunk-e meg funkciót. Továbbá a módosítások új hibák bekerülését okozhatják, ezek kiszűrése is fontos feladat. Regressziós tesztelés támogatásához szükségünk van arra, hogy hatékonyan tudjuk megfogalmazni a rendszerünkkel szemben azokat a funkciókat és viselkedési aspektusokat, amelyek megváltozása hibát eredményez. Nehezíti a regressziós tesztelést, hogy (i) a szoftverek specifikációja gyakran hiányos, (ii) a tervezőnek és a fejlesztőnek sincs elég ideje a precíz (formális) specifikáció elkészítésére, vagy (iii) a rendszer komplexitása miatt nem is lehet teljes specifikációt elkészíteni. Ez oda vezet, hogy a regressziós tesztelést ad-hoc jelleggel végezzük el, így a rendszerben megbúvó hibák könnyen elkerülhetik a figyelmünket, benne maradhatnak a szoftverben. Tehát a regressziós tesztelést nagyban megkönnyíti, ha ismerjük a megváltozott komponens működését. A tesztelést továbbá az is nehezíti, hogy sok olyan külső könyvtárat és egyéb komponents használunk, amelynek a pontos specifikációja nem ismert számunkra, hiszen napjainkban gyakori más gyártók szoftverkomponenseinek feketedoboz jellegű használata. Ezen komponensek viselkedésének közvetlen felderítése nehéz, vagy teljes komplexitásukban lehetetlen lehet.

Kutatásunk célja egy olyan eszköz megvalósítása, amely képes egy komponens működését megtanulni anélkül, hogy belelátna a belső működésébe, továbbá a tanulás alapján képes viselkedési modellt készíteni. Amennyiben a rendszer túl összetettnek bizonyul a viselkedés feltérképezéséhez, absztrakciók segítségével automatikusan fókuszálunk a tanulás szempontjából releváns funkciókra. Ezáltal definiálni lehet a tanulás, tesztelés és ellenőrzés szempontjából releváns funkciókat. Amennyiben a tanulás sikeres, a feltérképezett

viselkedéshez magas fedettséggel rendelkező tesztkészletet generálunk adott funkciók ellenőrzésére, amelyet regressziós tesztelés során fel tudunk használni, a további módosítások ellenőrizhetővé válnak.

Dolgozatunkban bemutatunk egy nyelvet, amely támogatja egy komponens bemeneteinek feature model [10] alapú absztrakcióját és a kimenetek konkretizációját. Az így elkészített algoritmus segítségével bonyolultabb rendszerek ellenőrzése is lehetővé válik. Emellett bemutatjuk, hogy automata tanulás segítségével miként támogatható a regressziós tesztelés. Az automatából különböző fedési kritériumoknak megfelelő teszteket generálunk, amelyek célzottan felhasználhatóak a komponens megtanult funkcióinak regressziós tesztelésére. Emellett ha a módosításokat elvégezte a felhasználó, akkor a módosított rendszer a kiindulási absztrakciót felhasználva megtanulható, és a korábbi rendszeren megtanult automatával ekvivalencia szempontjából összehasonlítható, így ellenőrizve, hogy a fejlesztés során nem került-e be szándékolatlanul módosítás vagy hiba.

Munkánk eredményeként az elkészített keretrendszer támogatja szoftver komponensek funkcionális regressziós tesztelését. Fontos előnye a megközelítésünknek, hogy lehetővé teszi ismeretlen (például harmadik féltől származó) szoftverkomponensek olyan viselkedéseinek megtanulását és ellenőrzését is, amelyekre az eddigi megközelítések nem nyújtottak támogatást.

Dolgozatunkat a 2. fejezetben az előismeretek áttekintésével kezdjük. Bemutatjuk általánosan a regressziós tesztelés folyamatát, definiáljuk az általunk használt Mealy-automata fogalmát, bemutatjuk az automatatanulás néhány lehetséges algoritmusát. Ezek után szót ejtünk a feature modellezésről is, hiszen az általunk fejlesztett eszköz bemeneteinek és kimeneteinek definiálására ez jelenti az alapot. A 3. fejezetben bemutatunk pár, a kutatásunkhoz kapcsolódó munkát. Ezekbe beleértve az általunk felhasznált harmadik fél által tervezett komponenseit a rendszerünknek, illetve a kutatási témánkhoz szorosan kapcsolódó hasonló eszközöket. Utóbbinál kitérünk arra, hogy az általunk fejlesztett keretrendszer milyen fő pontokban tér el ezen munkáktól. Ezt követően a megvalósítás elméleti (4. fejezet) és gyakorlati (7. fejezet) oldalát mutatjuk be egy esettanulmányon keresztül, ezzel szemléletesebbé téve a keretrendszer egyes funkcióinak a leírását. Ebben a részben az általunk hozzátett lényegesebb újításokat - az absztrakciós és konkretizációs algoritmusokat (5. fejezet), illetve az automatatanuláson alapuló regressziós tesztelést (6. fejezet) - részletesebben is kifejtjük. Az eszköz egyes komponenseinek működéséről méréseket végeztünk, hogy megtudjuk hogyan skálázódnak, ezekről is a 7. fejezetben írunk. Végezetül a 8. fejezetben egy rövid összefoglalást követve a jövőbeli fejlesztési terveinkkel zárjuk a dokumentációt.

2. fejezet

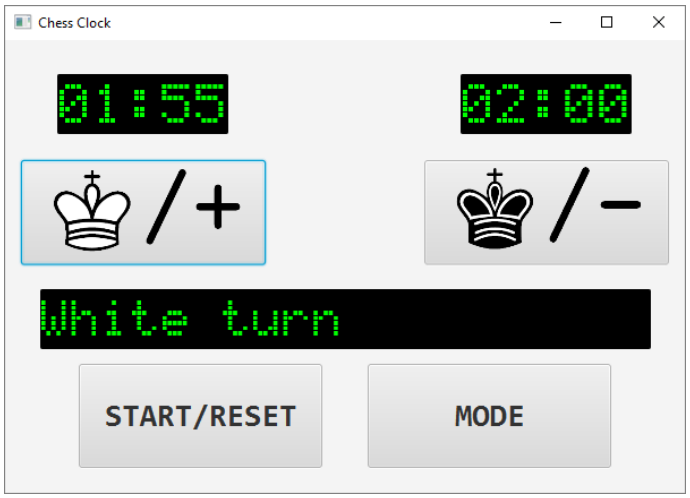
Előismeretek

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

2.1. Esettanulmány: Sakkóra

Dolgozatunkban az általunk implementált keretrendszer működését egy esettanulmányon keresztül fogjuk bemutatni. Erre a célra egy sakkóra YAKINDU [20] fejlesztőeszköz segítségével készített modelljét fogunk alkalmazni, mely az egyetemen oktatott Rendszermodellezés tantárgy [5] házifeladatához készült.

A sakkóra alapvető feladata, hogy a sakkjáték résztvevőinek gondolkodási idejét megszabja. Mindkét játékos (Világos ill. Sötét, angolul White ill. Black) rendelkezik meghatározott mennyiségű gondolkodási idővel, amelyet a sakkóra kijelez. Az aktuálisan soron lévő játékos gondolkodási ideje folyamatosan fogy; ha elfogyott, akkor veszít, és a gép sípszóval jelzi a játék időtúllépés miatti befejeződését. Ha az aktuális játékos az ideje lejárta előtt megteszi a lépését, akkor a sakkóra megfelelő gombját meg kell nyomnia. A gombnyomás hatására a lépését befejező játékos gondolkodási idejének csökkenése megáll, és (a játék beállításaitól függően) az idő értéke egy rögzített mennyiségű jutalom idővel nő; ezek után azonnal a másik játékosra kerül a sor. Amikor újra sorra kerül a játékos, az előző lépése után maradt (és jutalommal növelt) gondolkodási időt használhatja fel a lépéséhez.



2.1. ábra. Sakkóra alkalmazás képernyőképe

A sakkóránk kezelőfelülete a következő elemekből áll:

- Fő szöveges kijelző (**Display text**), ahol a sakkóra általános tájékoztatást ad. A 2.1. ábrán látható képernyőképen épp a "White turn" (fehér játékos lép) menüpontban való tartózkodásról tájékoztat.
- A Világos játékos idejét mutató órakijelző (**White display**), mely az alkalmazás bal felső sarkában helyezkedik el.
- A Sötét játékos idejét mutató órakijelző (**Black display**), mely az alkalmazás jobb felső sarkában helyezkedik el.
- Sípjelző, amely az idő leteltekor és egyéb esetekben jelezhet (**Beeper**).
- A **buttonMode** és **buttonStart** gombok a játékmód beállítására és a játék indítására ill. leállítására
- A Világos játékos által minden lépése után lenyomott **buttonWhite** gomb, mellyel a menü egyes beállításainál az értékeket növelni lehet.
- A Sötét játékos által minden lépése után lenyomott **buttonBlack** gomb, mellyel a menü egyes beállításainál az értékeket csökkenteni lehet.

A játékosoknak a játék kezdete előtt lehetőségük van a gondolkodási- és a jutalomidők beállítására. Ezt a menüben tudják megtenni, ahol a menüpontok között az erre megfelelő gombbal lehet lépkedni. A beállítható értékeket a sakkóra beállításaiban szereplő határok között, meghatározott időközökkel, tudják állítani. Ezek mellet lehetőségük van még a kezdőjátékos beállítására is. Ezt követően elindíthatják a játékot, amely a fent leírtaknak megfelelően zajlik.

2.2. Regressziós tesztelés

A regressziós tesztelés egy szoftver két különböző verziójának az összehasonlítására használatos eljárás. Ennek segítségével állapítható meg, hogy vajon az új verzió újdonságai okoztak-e változást a rendszer eddigi funkcióit tekintve, okozott-e a fejlesztés regressziós hibákat. Regressziós hibának nevezzük azt, ha a módosítás után a rendszer egy funkciója hibásan, vagy egyáltalán nem működik. Ennek három fő típusát szoktuk elkülöníteni: lokális, leleplező és távoli. Lokális regressziós hiba esetén a változtatások új hibák megjelenését eredményezik, leleplező hiba esetén egy már meglévő, korábban nem jelentkező hiba válik detektálhatóvá. Távoli regressziós hibáról több komponens esetén lehet beszélni, ilyenkor az egyik részben végzett módosítás a rendszer egy teljesen másik részében okoz hibás működést. A regressziós hibák kiszűrését nagyban megkönnyíti a változtatások részleteinek az ismerete, ezek azonban nem mindig elérhetőek. Erre jó példa lehet akár, ha az új verzió másfajta programnyelven lett implementálva, vagy ha a megváltoztatott szoftverkomponens forráskódja nem ismert.

2.3. Mealy automata

Rendszer viselkedésének matematikaliag precíz specifikációjára gyakran alkalmaznak formális modelleket. A Mealy Machine, vagy Mealy automata [11] egy véges állapotgép, melynek tranzícióin megjelennek bemeneti szimbólumok (kiváltó események) és kimeneti szimbólumok (akciók, viselkedés) is.

Definíció 1 (Mealy-automata). A Mealy-automatát a következő $\mathcal{M} = \langle I, O, Q, q^0, \rightarrow \rangle$ struktúra írja le [9], ahol

- I a bemeneti ábécé véges halmaza,
- O a kimeneti ábécé véges halmaza,
- Q az állapotok véges halmaza,
- $q^0 \in Q$ egy kezdőállapot,
- $\rightarrow \subseteq Q \times I \times O \times Q$ átmeneti függvény. ▪

Jelölje $q \xrightarrow{i/o} q'$ azt, ha $(q, i, o, q') \in \rightarrow$.
Egy rendszer viselkedését akkor írja le egy $\mathcal{M}$ automata, ha

- Teljesség: minden $i_1, \ldots, i_n$ bemeneti és $o_1, \ldots, o_n$ kimeneti sorozatra, ahol $\forall_{k \in [1;n]}$: $i_k \in I \wedge o_k \in O$ igaz az, hogy a rendszer a $i_1, \ldots, i_n$ bemeneti szekvenciára adhatja válaszul az $o_1, \ldots, o_n$ kimeneti szekvenciát, akkor létezik $\mathcal{M}$-ben egy útvonal, ahol $q^0 \xrightarrow{i_1/o_1} q^1 \xrightarrow{i_2/o_2} \ldots \xrightarrow{i_n/o_n}$ teljesül
- Helyesség: minden $\mathcal{M}$-beli $q^0 \xrightarrow{i_1/o_1} q^1 \xrightarrow{i_2/o_2} \ldots \xrightarrow{i_n/o_n}$ útvonalra létezik a rendszerben, hogy a $i_1, \ldots, i_n$ bemeneti szekvenciára adhatja válaszul az $o_1, \ldots, o_n$ kimeneti szekvenciát

Feltesszük, hogy $\mathcal{M}$ teljes, tehát minden q állapotra és i bemenetre létezik $q \xrightarrow{i/o} q'$ átmenet valamely o, q'-re.

Definíció 2 (Determinisztikus Mealy-automata). $\mathcal{M}$ Mealy-automata determinisztikus, ha minden q állapotra és i bemenetre létezik pontosan egy o és pontosan egy q', hogy $q \xrightarrow{i/o} q'$. ▪

2.3.1. Automatatanulás

Az automatatanulás régóta tanulmányozott terület egyre több gyakorlati alkalmazással. A mi dolgozatunk célja is, hogy ezt a szép elméleti területet praktikusan, problémák megoldására is használjuk. Az automatatanulás alapja, hogy a rendszer lehetséges lefutásait, azaz bemenetekre adott válaszait feldolgozva egy véges állapotgép alapú modellt építsünk a megfigyeléseinkből. Többféle algoritmus is létezik változatos formalizmusokat és rendszereket támogatva, ezek közül mi a Mealy-automata-tanuló algoritmusokat használtuk munkánk alapjaként. Az Angluin által megalkotott algoritmus [2] úgynevezett tagsági kérdéseket, azaz membership query-ket fogalmaz meg a tanulandó rendszer számára (SUL, System Under Learning), és a kapott válaszok alapján bemenet-kimeneti párokat építve tanulja meg a viselkedéseket. Az elkészült automata rendszerrel való ekvivalenciájának ellenőrzésével fejeződik be a tanulás. Mealy automaták tanulására is készítettek tanuló algoritmusokat[16], mi a munkánk során ezekre a munkákra építettünk. Rendelkezésre álló tanuló keretrendszerek közül a népszerű LearnLib [14] szoftvert egészítettük ki a saját céljainknak megfelelően. A LearnLib keretrendszer épít Angluin L* algoritmusára [8, 15].

Sajnos a keretrendszer használatának vannak korlátai, többek között nem képes adatstruktúrákat kezelni, és nem képes nemdeterminisztikus rendszereket megtanulni, továbbá végtelen állapotterű rendszerek tanulására sem alkalmas. Munkánk során ezeket a hiányosságokat igyekszünk kiküszöbölni részben vagy egészben.

2.4. Automatikus logikai következtetők

Munkánk során SAT alapú logikai következőtetőket használtunk fel az absztrakció és a konkretizáció során. A SAT megoldó bemenete egy logikai formula (úgynevezett SAT formula), amely bináris változókból és azok logikai kapcsolatait leíró operátorokból áll. A SAT megoldó kiértékeli a formulát és eldönti, hogy kielégíthető-e. Amennyiben talál változók egy olyan lekötését, amire a formula igaz értékűre értékelődik ki, azt a lekötést vissza is adja a kimenetén. A SAT probléma eldöntése általános esetben NP nehéz probléma, azonban napjaink hatékony megoldó szoftverei kifejezetten nagy SAT problémákat is meg tudnak oldani már.

2.5. Feature model

Úgynevezett feature modelleket, azaz "tulajdonság" modelleket régóta használnak szoftverek, rendszerek lehetséges konfigurációinak reprezentálására. Munkánk során a bemenetek és kimenetek tulajdonságainak reprezentálására mi is ezt a megközelítést választottuk, ugyanis ez alapján nyelvi támogatást készíthettünk az absztrakció és konkretizációs algoritmusok működéséhez. Ily módon megteremtettük a lehetőségét, hogy a felhasználó megadja a tanulás és a tesztelés fókuszát, célját. A feature model feature-ök egy hierarchikus elrendezése. Első felhasználása a szoftverfejlesztésben, a szoftverkomponensek konfigurációjának kompakt ábrázolása volt [10]. Azóta felhasználásának sok fajtája elterjedt, az általunk használt pontos definíció alább olvasható.

Definíció 3 (Feature model). Az $F = \{feature_1, \ldots, feature_n\}$ feature halmaznak *fm* feature modelje egy T(*fm*) feature fából és Constr(*fm*) keresztirányú megkötésekből áll. A T(*fm*) feature fa egy $\langle F, Children \rangle$ irányított fa, melynek csúcsait F featurehalmaz, éleit *Children* halmaz alkotja. Jelölje $p \to ch$, hogy $\langle p, ch \rangle \in Children$ (p a szülő, *ch* a gyerek), és $p \not\to ch$, hogy $\langle p, ch \rangle \notin Children$. Ch($p$) jelöli p összes lehetséges *ch* gyerekének halmazát. p(*ch*) = p jelölje *ch* szülőjét.

Pontosan egy olyan $r \in F$ létezik, melyre $\nexists p \in F : p \to r$. Ez a T(*fm*) fa gyökere. Jelben r(*fm*).

Minden $p \in$ F(*fm*) feature esetén, melyre Ch(p) $\neq \emptyset$ (p-nek van gyereke) p gyerekeire az alábbiak közül pontosan egy igaz:

1. *tartalmazás* kapcsolatban vannak: ekkor *child* $\in$ Ch(p) gyerek
   (a) vagy *kötelező eleme* p-nek,
   (b) vagy *opcionális eleme* p-nek;
2. *vagy* kapcsolatban vannak;
3. *alternatíva* kapcsolatban vannak. ▪

Definíció 4 (Keresztirányú megkötések). A Constr(*fm*) keresztirányú megkötések a F(*fm*) feature-ökből, negálás, és, vagy, implikáció operátorokból összeállítható kifejezések.▪

Feature modelen ábrázoljuk a vizsgált rendszer be- és kimenetét, így az üzenet egyes részeinek meglétét, igaz-hamis értékeket, felsorolt típusokat és véges számintervallumokat is leírhatunk igaz-hamis értékekként [4].

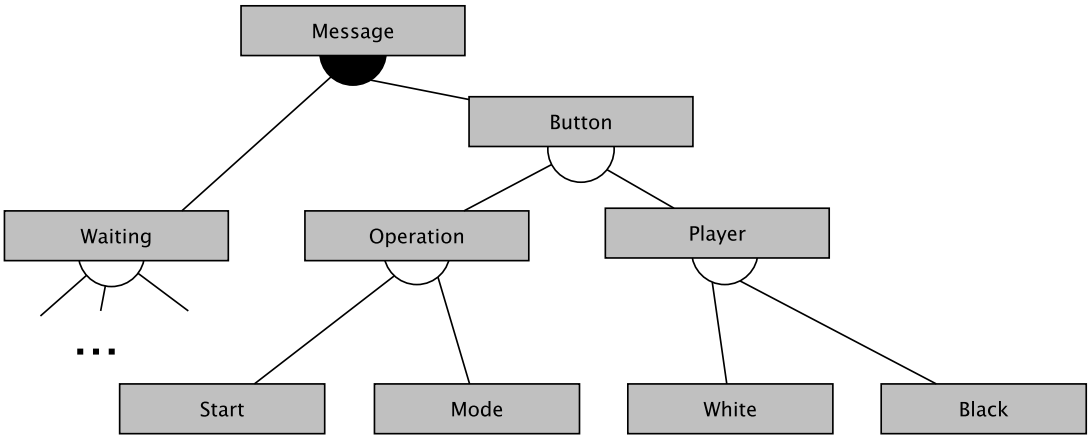
A feature modelek példányai a konfigurációk, amelyben az egyes feature-ökhöz igaz-hamis értékek tartoznak a következők szerint:

Definíció 5 (Konfiguráció). *fm* feature model konfigurációja egy $c : \mathrm{F}(fm) \to \{igaz, hamis\}$ függvény, melyre az alábbi kifejezések igazak [12]:

1. $\mathrm{c}(\mathrm{r}(fm))$, vagyis a gyökér mindig igaz;
2. $\forall child \in \mathrm{F}(fm) \setminus \mathrm{r}(fm) : \mathrm{c}(child) \implies \mathrm{c}(\mathrm{p}(child))$, vagyis ha egy feature igaz, akkor a szülője is;
3. $\forall mand \in \mathrm{F}(fm)$, ahol *mand kötelező eleme* $\mathrm{p}(mand)$-nak : $\mathrm{c}(\mathrm{p}(mand)) \implies \mathrm{c}(mand)$, vagyis ha egy feature igaz, akkor a *kötelező elemei* is igazak;
4. $\forall p \in \mathrm{F}(fm), \text{ahol } \mathrm{Children}(p) = \{ch_1, ch_2, \ldots, ch_n\}$ gyerekek *vagy* kapcsolatban vannak : $\mathrm{c}(p) \implies (\mathrm{c}(ch_1) \vee \mathrm{c}(ch_2) \vee \ldots \vee \mathrm{c}(ch_n))$, vagyis ha egy feature igaz, akkor a *vagy* kapcsolatban lévő gyerekei közül legalább 1 igaz;
5. $\forall p \in \mathrm{F}(fm), \text{ahol } \mathrm{Children}(p) = \{ch_1, ch_2, \ldots, ch_n\}$ gyerekek *alternatív* kapcsolatban vannak : $\mathrm{c}(p) \implies \bigvee_{i=1}^{n} \left(\mathrm{c}(ch_i) \wedge \bigwedge_{\substack{j=1 \\ i \neq j}}^{n} \neg\, \mathrm{c}(ch_j) \right)$, vagyis ha egy feature igaz, akkor a *alternatív* kapcsolatban lévő gyerekei közül pontosan 1 igaz.
6. $\forall constr \in \mathrm{Constr}(fm)$: a *constr*-ban szereplő összes $f \in \mathrm{F}(fm)$ feature-t $\mathrm{c}(f)$ értékkel helyettesítve kapott kifejezés, vagyis a *keresztirányú megkötések* igazak.

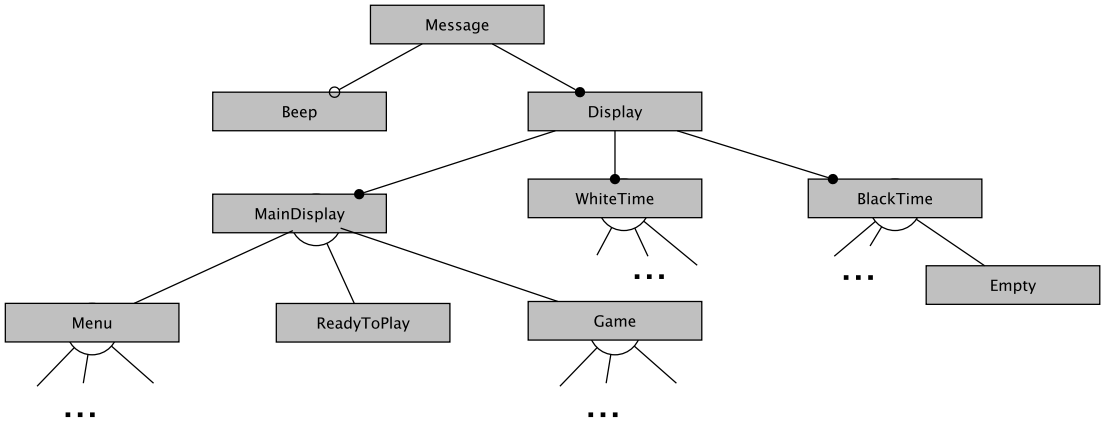
$\mathfrak{C}(fm)$ jelölje a lehetséges c konfigurációk halmazát. ▪

A feature model tipikus ábrázolása a feature diagram. Erre példa a 2.2 és 2.3 ábra. A feature-öket egyszerű fa szerkezetben ábrázolja, a szülők alá rendezve a gyerekeit. Az egyes tartalmazási kapcsolatokra más-más jelölést alkalmazva.



2.2. ábra. Bemeneti feature model

Példa 1. *A sakkóra bemenetére egy üzenet megy (Message), amelynek a gyerekei* vagy *kapcsolatban vannak: tehát vagy egy gombnyomás (Button), vagy egy időtartam telik el (Waiting), vagy mindkettő, az időtartam eltelte után gombnyomás következik. A várakozásnál a bemeneten előforduló számokat alternatív kapcsolatban felsoroltuk, így pontosan egy igaz belőlük, ha a van várakozás. A gombok csoportosítva szerepelnek az ábrán. Szintén alternatív kapcsolatban.*



2.3. ábra. Kimeneti feature model

Példa 2. *A sakkóra kimenetén* opcionális elemként *megjelenhet egy sípolás (Beep). A kijelzők (Display), a fő (MainDisplay) és számkijelzők (WhiteTime, BlackTime)* kötelező elemek*. A számkijelzők a számok mellett üresek (Empty) is lehetnek. A főkijelzőn előforduló szövegek hierarchikusan csoportosítva szerepelnek a MainDisplay feature alatt.*

Példa 3. *A sakkóra kimenetén előírhatjuk, hogy a játékra kész állapotban (ReadyToPlay) a számkijelzők üresek. Ezt a következő keresztirányú megkötés írja le:*

$$ReadyToPlay \implies Empty.$$

3. fejezet

Kapcsolódó munkák

Ebben a fejezetben áttekintjük az irodalomban ismert kapcsolódó munkákat.

3.1. Regressziós tesztelés

A szoftver tesztelésnek, és azon belül is a regressziós tesztelésnek nagy irodalma van, hiszen egy olyan régóta létező diszciplina, amely alapvető módon járul hozzá a szoftver és egyéb informatikai rendszerek működésének helyességéhez. Egy jó áttekintő munka [21] alapján elmondhatjuk, hogy a megfelelő tesztek kiválasztása, minimális tesztkészlet összeállítása, továbbá az ezekhez kapcsolódó priorizálási és egyéb feladatok komoly kihvást jelentenek napjaink teszt mérnökei számára is.

3.2. LearnLib keretrendszer automatatanulásra

A LearnLib egy automatatanulásra készített keretrendszer [14]. Többféle tanulóalgoritmus implementálva van benne, ezek segítségével lehet véges automatákat és Mealy automatákat készíteni a tanulás alatt álló rendszerről. Több ipari alkalmazása is van, kommunikációs protokoll implementációkban találtak hibákat a segítségével [9]. A LearnLib keretrendszernek többféle kiegészítése is van, többek között tettek lépéseket adat jellegű viselkedések megtanulására is regiszterautomaták segítségével. Ilyenkor azonban a tanulás komplexitása nagyon megnő, gyakorlati esetekre nehezen alkalmazható. A LearnLib hátránya emellett még az is, hogy nem képes nemdeterminisztikus rendszerek kezelésére, továbbá a nagy állapottér méreteket sem kezeli hatékonyan.

3.3. Tomte keretrendszer absztrakciós automatatanulásra

A Tomte keretrendszer [1] automatikus absztrakciódefiniálást biztosít automaták megtanulásához. Míg általában a tanuló algoritmusok 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, annak az ellenpélda mentén automatikusan finomítja azt [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 annak csak egyes funkcióira szűrni. Az általunk fejlesztett keretrendszer ebben fog különbözni a Tomte eszköztől. A felhasználó a kommunikációra megfogalmazott megkötéseivel tudja az általa kívánt szemszögből vizsgálni a rendszert, megtanulni annak absztrakt működését. A regressziós tesztelés is ennek a funkcionalitásnak a megőrzését vizsgálva zajlik.

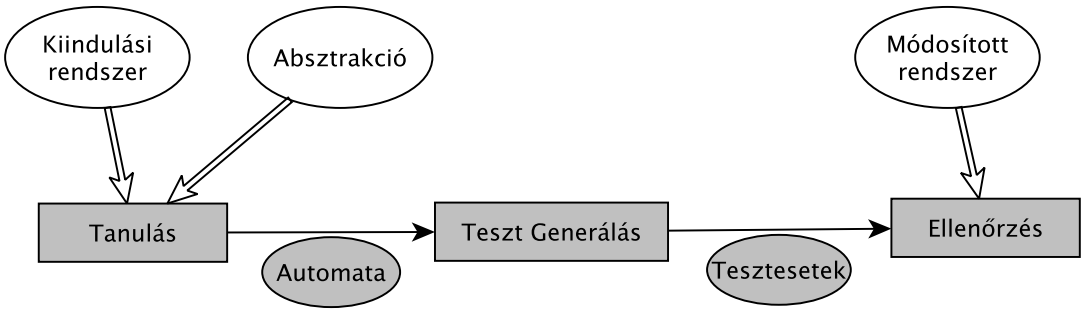
4. fejezet

Automatatanulás alapú regressziós tesztelés

Rendszerek regressziós tesztjeinek előllításához a rendelkezésre álló megvalósítás viselkedését vesszük alapul. Automatatanulással a rendszer viselkedését feltéképezzük, hogy az új verzió ellenőrzését hatékonyan el tudjuk végezni.

4.1. Megközelítés

A regressziós tesztek kézi elkészítése időigényes folyamat. A rendszer komplexitása, állapotainak számossága nagy mennyiségű teszteset előállítását igényli a kimerítő teszteléshez. Továbbá egyes változtatások szándékosak, ezeknél nem szükséges ellenőrizni a funkciók eltérését. Így egy megfelelő absztrakció bevezetésével elegendő lenne csak bizonyos szempontból vizsgálni a rendszer funkcióinak változatlanságát, és csak ezeket a szempontokat vizsgáló teszteseteket generálni. A fent vázolt problémák megoldására egy automatizálható módszert dolgoztunk ki. Ennek során a rendszer viselkedését a kívánt absztrakción keresztül megtanuljuk. Az így előállt automatán generálunk teszteket, majd a módosított rendszert ezekkel a tesztesetekkel ellenőrizzük. A megközelítés átfogó működését a 4.1. ábra mutatja be.



4.1. ábra. Automatatanulás alapú regressziós tesztelés

A folyamat első lépése a tanulás. Ez, ahogy az már fentebb említve volt egy absztrakción keresztül történik, melyet a felhasználó maga szabhat meg ezáltal konfigurálva a tanulás folyamatát, a megtanulandó funkcionalitást. A tanulás eredményeként előáll egy automata, ami a vizsgált rendszer működését írja le az absztrakciós szempontokat figyelembe véve. Az automata felhasználásával válik lehetővé a következő lépésben a regressziós teszteléshez szükségek tesztesetek generálása, amelyekkel a rendszer funkcióinak a megfe-

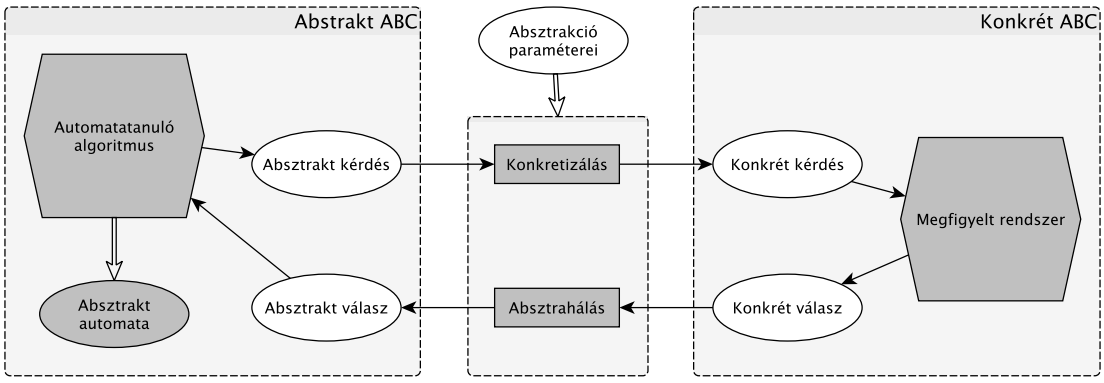
lelő működését kívánjuk leellenőrizni. Az így kapott tesztesetekkel elvégezhető a vizsgált rendszer új verziójának az ellenőrzése. A folyamatok egyes részletei a következő alfejezetekben kerülnek bővebb kifejtésre.

4.2. Konfigurálható absztrakcióval támogatott tanulás

Munkánk célja a tanuló algoritmusok kiegészítése konfigurálható absztrakcióval, hogy akár olyan szoftver komponensek megtanulását is támogassuk, amelyek viselkedése adatfüggő. A megfelelő absztrakció kiválasztása a felhasználó feladata, aki ennek segítségével tudja definiálni, hogy a szoftver/komponens/rendszer mely aspektusai, funkciói relevánsak a regressziós tesztelés szempontjából. Emellett az absztrakció lehetőséget ad arra is, hogy ha sikertelen a tanulás, akkor még több viselkedést kiegyszerüsítsen a felhsaználó a tanulás folyamatából. Több dolog is vezethet a sikertelen tanuláshoz, például az automatának lehet túl sok vagy akár végtelen állapota, ha a tanulandó szoftver túl bonyolult. Ezen problémák kezelésében tud segíteni a konfigurálható absztrakció,

A rendszer viselkedésének absztrakciója az üzenetváltások absztrakciójával valósul meg. A megfigyelt rendszert az aktív tanuló algoritmus nem közvetlenül hajtja meg a vizsgálni kívánt bemenetekkel, hanem absztrakt bemeneteken keresztül tanul, amelyekre absztrakt kimeneteket kap válaszul, ahogy az a 4.2. ábrán látható.

Az absztrakció segítségével több különböző kimenet és bemenet sorolható ekvivalencia osztályokba. Például amennyiben egy termosztát viselkedését szeretnénk megtanulni, amely vizet tárol, akkor a viselkedése nem a hőmérséklet pontos értékétől függ, hanem hogy elért-e egy kritikus hőmérsékletet vagy nem. A felhasználó ezt tudja meghatározni az absztrakció definiálása során.



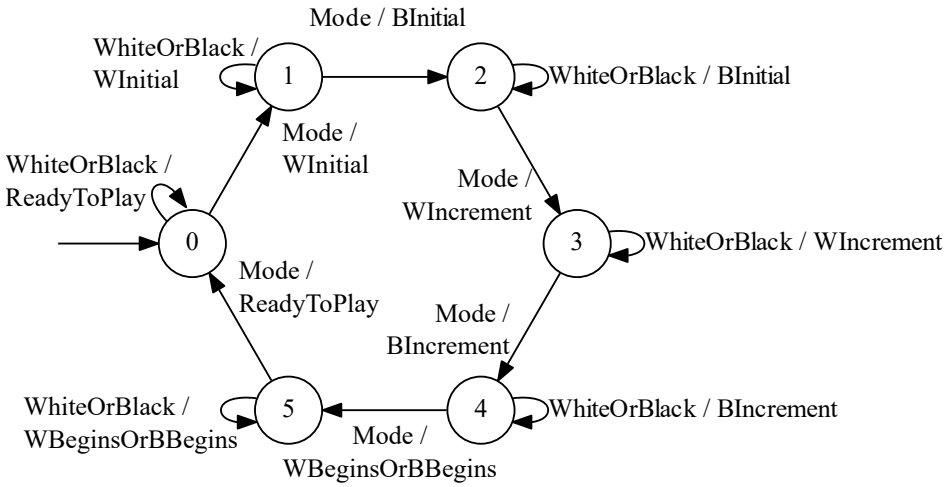
4.2. ábra. Absztrakció a tanulásban

A tanuló algoritmus egy olyan komponenst reprezentál, amely egy be- és egy kimeneti ábécén keresztül kommunikálni tud a megfigyelt rendszerrel. Ahogy azt az előismeretekben is áttekintettük, a különböző lekérdezések segítségével ez a komponens képes egy a rendszer működését tükröző automatát készíteni. Mivel a komponens által használt ábécék (kérdései és a válaszok) absztraktak, így az általa előállított automata is absztrakt lesz. A későbbiekben az automata ábécéjét Σ szimbólummal is fogjuk jelölni. Az ábra túl oldalán szereplő megfigyelt rendszer azonban csak konkrét, az általa definiált interfésznek megfelelő kéréseket képes fogadni. Illetve az ezekre adott válaszai is konkrétak lesznek. Ezért szükséges mind az absztrakció, mind a konkretizáció megvalósítása a kommunikáció nyelvére.

Az ehhez szükséges konkretizáló és absztraháló szabályok beállítását a felhasználóra bízzuk, így biztosítva azt, hogy bármely általa fontosnak tartott funkcionalitás szem-

pontjából tudja vizsgálni a rendszert. A sakkóra viselkedésének egy lehetséges absztrakt megtanulását a 4. példában mutatjuk be.

Példa 4. *Vegyük azt az esetet, hogy a sakkóra menürendszerének a működését szeretnénk megfigyelni és ezen belül is csupán a beállítási opciók érdekelnek minket, ezek konkrét értékei nem. Ebben az esetben a rendszernek küldött üzenetekre megfogalmazhatjuk azt a megkötést, hogy a Start gomb ne kerüljön megnyomásra, így biztosítva, hogy csak a menü viselkedését vizsgáljuk. A rendszer kimenetei közül csak a fő szöveges kijelző (Display text) értékét vizsgálva pedig azt tudjuk elérni, hogy a beállítási opciók értékei ne növeljék meg feleslegesen a megtanult automata állapotterét. Ezt az absztrakciót alkalmazva a megtanult rendszer automatája a 4.3. ábrának megfelően fog kinézni.*

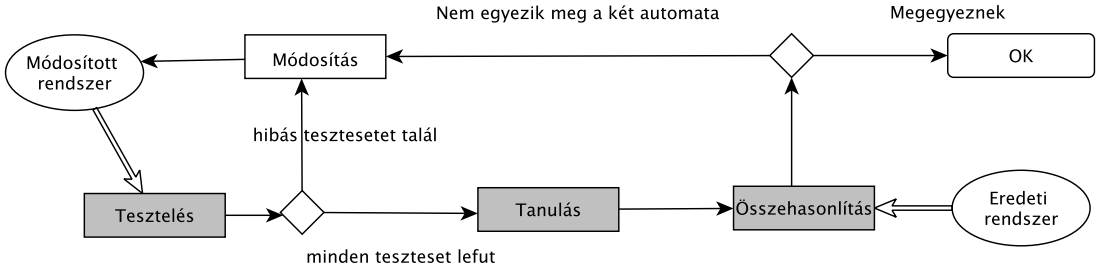


4.3. ábra. Menürendszer automata reprezentációja a tanulás után

Munkánk során feature model alapú nyelvet választottunk a komponens környezettel való interakcióinak ábrázolására, mivel erre egyszerűen meg lehet fogalmazni az absztrakciós és konfigurációs szabályokat is, ezekről a 5. fejezetben bővebben írunk.

4.3. Ellenőrzési módszerek

A módosított rendszeren regressziós tesztelést hajtunk végre. Ennek két lépése van, melyet a 4.4. ábra szemléltet. Bármelyik lépésben is eltérést tapasztalunk, azt mondjuk, hogy a rendszer funkcióinak a változatlansága nincsen biztosítva.

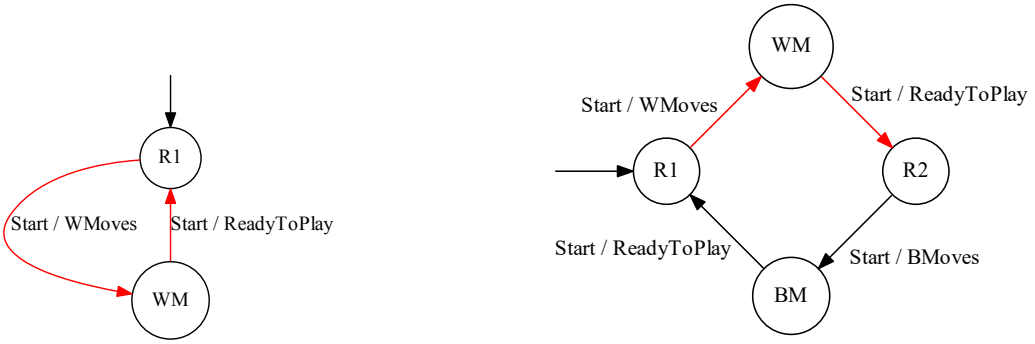


4.4. ábra. Ellenőrzés folyamata

Az ellenőrzés első lépésében a korábban megtanult automata alapján generált teszteseteket futtatjuk a megváltozott rendszeren. Ezen tesztesetek az absztrakt kérdéseknek megfelelő konkrét kérdés és elvárt válasz párokat tartalmazzák. A hibás teszteset szekvenciák így könnyen detektálhatóak, a javítások megkönnyítése érdekében a fejlesztő számára könnyen elérhetőek.

Abban az esetben sem lehetünk biztosak benne, hogy a módosított rendszer funkcionálisan megegyezik az előző verzióval, ha az előző lépésben generált összes tesztesetre megfelelően működik. Előfordulhat például, hogy az új verzió komplexebb, működése csak a teszteseket által le nem fedett utakon tér el. Ennek szemléltetésére szolgál a 5. példa.

Példa 5. *Tételezzünk fel egy olyan módosítást, hogy két sakkjátszma között a program automatikusan megváltoztatja a kezdő játékost a sportszerűség jegyében. Absztrakciónk során pedig csak a Start gomb megnyomásának eseményét, illetve a fő kijelző (Display text) kijelző értékét vizsgáljuk. Ebben az esetben az eredeti rendszer és az új sportszerű verzió absztrakt automata modellje a 4.5. ábrán láthatóan fog kinézni. Az eredeti automatán generált - átmenetfedő - tesztesetek azonban le fognak futni rajta, hiszen a Start gomb kétszeri megnyomására még az elvárt szöveget írja ki a fő szöveges kijelzőre (Display text), csak a harmadik input beérkezésekor tapasztalnánk eltérést, a harmadik input tudja csak megkülönböztetni a két automatát.*



(a) Régi modell **(b)** Új modell

4.5. ábra. Tesztesetekkel nem detektálható funkcionális hiba

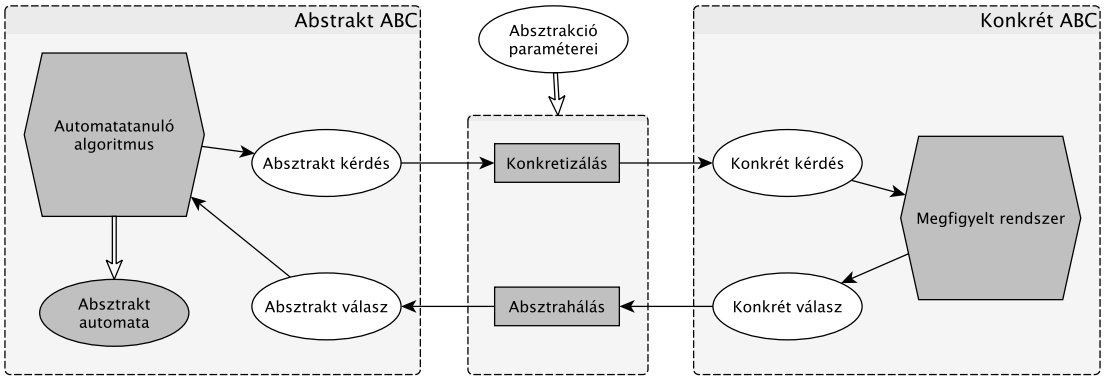
Az ilyen esetek kiszűrése érdekében a sikeresen lefutott tesztesetek kiértékelése után még az új verzió működését leíró automatát is megtanuljuk. Majd összehasonlítást végzünk a rendszer verzióinak megtanult automatáin. Csak akkor mondhatjuk, hogy a módosítás nem okozott meghibásodást a funkcionalitás szempontjából, ha az automata összehasonlítás is pozitív eredményt ad.

5. fejezet

Absztrakciós és konkretizációs algoritmusok a tanulásban

Ahogyan azt korábban is írtuk, a rendszer absztrakt viselkedésének a megtanulása a kommunikáció során, azaz a komponens bemeneteinek és kimeneteinek absztrakciójával valósul meg. Ezt a folyamatot a 5.1. ábra szemlélteti. Ehhez szükség van a bemenetek és kimenetek definiálására, továbbá egy nyelvre, amelyen a felhasználó megfogalmazza az absztrakciót, azaz a bemenetek és kimenetek ekvivalencia osztályokba sorolását. Az absztrakció és konkretizációt automatikusan számoló algoritmusok pedig támogatják a regressziós tesztelés folyamatát.

A 5.1 alfejezetben az absztrakció paramétereinek a megadásával fogunk foglalkozni. A 5.2 az absztrakció folyamatát, a 5.3 pedig a konkretizáció folyamatát fogja bemutatni.



5.1. ábra. Lekérdezések feature modelje

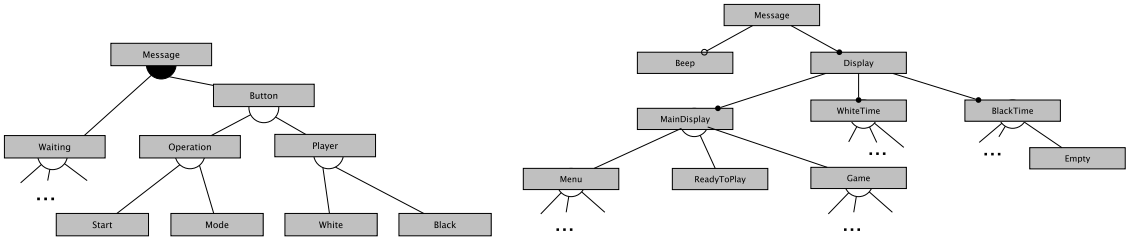
Dolgozatunk további részeiben használjuk majd az absztrakciókra vonatkozóan a determinisztikusság és a stabilitás fogalmát, ezért ezeket is ebben a fejezetben definiáljuk.

Definíció 6 (Determinisztikus absztrakció). Egy komponens működésére vonatkozó absztrakciót determinisztikusnak nevezünk, ha az absztrakción keresztül és a lehetséges konkretizálások esetére megtanulva a komponens működését a kapott Mealy automata is determinisztikus lesz. ▪

Definíció 7 (Stabil absztrakció). Egy komponens működésére vonatkozó absztrakciót stabilnak nevezünk, ha az absztrakción keresztül, bármely lehetséges konkretizálások estén megtanulva a komponens működését, a kapott Mealy automaták ekvivalensek lesznek. ▪

5.1. Feature model kezelése

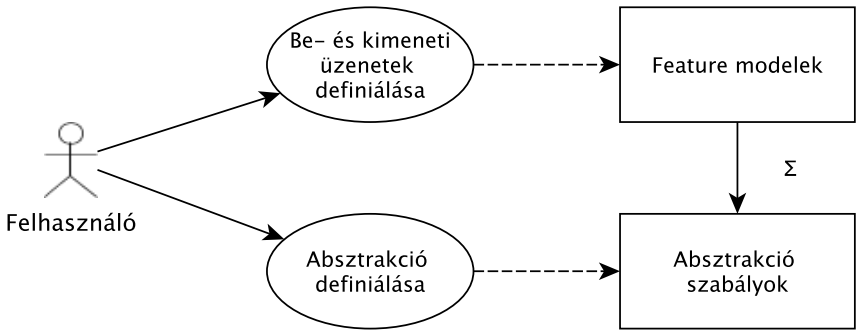
A tanuló algoritmus és a megfigyelt rendszer közötti interakció definiálására a feature modell alapú leírást választottuk. Ennek előnye, hogy absztrakciós és konkretizációs szabályok könnyen megfogalmazhatóak rá, ezen kívül szemléletesen lehet velük ábrázolni az üzenetek tartalmát. A esettanulmányunkhoz tartozó kérdés- és válasz üzenetek modelljét a 5.2. ábrának megfelelően valósítottuk meg.



(a) Bemenet feature modelje **(b)** Kimenet feature modelje

5.2. ábra. Feature model példák

Az absztrakciós (és a konkretizációs szabályok egy részének) a definiálását a felhasználóra bízzuk, hiszen ezeken a megkötéseken keresztül tudja ő leírni, hogy milyen szemszögből szeretné megfigyelni a megtanulandó rendszert. Ezen kívül a felhasználó feladata a vizsgált rendszer be- és kimenetének definiálása a feature model definiálásán keresztül. Ezeknek a bemente-kimenet (kérdés és válasz) feature modelleknek a konfigurációi meghatározzák a konkrét ábécét. Ennek a megadott szabályok szerinti absztrakciójával fogjuk majd megkapni az automatatanuláshoz szükséges absztrakt ábécét. Ezt a folyamatot a 5.3 ábra szemlélteti.



5.3. ábra. Absztrakciós paraméterek megadása

Az absztrakciós szabályok megadása mellett még lehetőséget biztosítunk a felhasználónak konkretizációkra vonatkozó megkötések tételére is. Ez akkor fontos, ha a rendszer egyes bemenetekre adott válaszai nem érdekesek az automata tanulás során.

5.2. Absztrakció

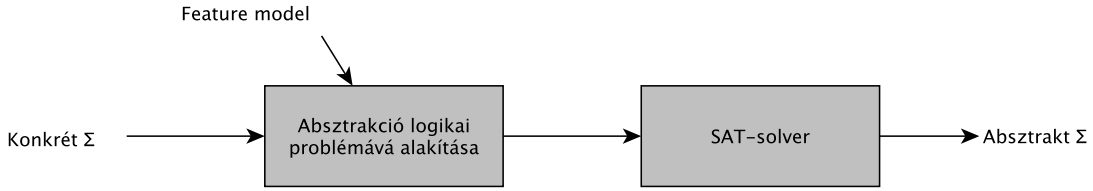
Az absztrakcióra sokféle módszert ki lehet dolgozni, mi a feature model alapú kommunikáció során a feature model üzenetek transzformációjával oldottuk meg a problémát. Kétféle absztrakciós lehetőséget tettünk lehetővé a felhasználó számára:

- Feature törlése: Ilyen absztakciós műveleteket a felhasználó akkor alkalmazhat, ha a rendszer egy-egy kimeneti, vagy bemeneti paramétere az absztrakció szempontjából teljesen lényegtelen. Törlésnél figyelembe kell venni, hogy a törölt feature a szülőjéhez képest milyen viszonyban áll.
  - *alternatíva* vagy *vagy* kapcsolat esetén a törlés műveletét nem tesszük lehetővé, hiszen így előfordulhatnak olyan konkrét feature modelek, amiket nem lehetne absztrahálni.
  - *kötelező elem* esetén az absztrahálást lehetővé tesszük, bár jelentősége csak akkor van, ha a feature a feature fában nem levél, hiszen konkretizáció esetén a *kötelező elem* mindenféleképpen visszakerül a modellbe. A gyerek elemeiben hordozott információt azonban elabsztrahálhatjuk ily módon.
  - *opcionális elem*, akkor nem lesz megkötés a az absztrakt példányból a konkrét példány értékére.
- Feature-ök összevonása: Ilyen műveleteket akkor alkalmazhat a felhasználó, mikor a feature-ök funkciója a megvalósítandó absztrakció szintjén azonos. Első kikötésünk egy ilyen művelettel szemben, hogy csak egy szinten lévő, közös szülőtől származó feature-öket vonhassunk össze. Ilyen absztrakciók alkalmazása során ügyelnünk kell arra, hogy mikor az absztrakt üzenetet konkretizáljuk figyelembe vegyük, hogy az összevont feature elemek milyen feature-öknek feleltek meg az összevonás előtt. Összevonás esetén az összevont elemek összes gyerekét töröljük. Az előző pontban vázolt *alternatíva* és *vagy* esetekben a törlés tiltva van, de ha egy szülőnek az összes ilyen kapcsolatban lévő gyerekét töröljük, akkor nem jutunk ellentmondásra. Feature-ök összevonását a szülőjükhöz viszonyított kapcsolatuk szerint különbözőképpen valósítjuk meg.
  - *alternatíva* kapcsolat esetén az elemeket problémamentesen össze lehet vonni. Az absztrakció konfigurációjában ennek az összevont feature-nek az igaz értéke azt fogja jelenteni, hogy az összevont elemek közül pontosan egy volt igaz.
  - *vagy* kapcsolat az előzőhoz hasonlóan problémamentesen összevonható. Az absztrakció konfigurációjában ennek az összevont feature-nek az igaz értéke azt fogja jelenteni, hogy az összevont elemek közül legalább egy volt igaz.
  - *kötelező elem* esetén nincs értelme az összevonásnak, hiszen az absztrakt példány konretizációja során biztosan visszakapnánk ezt az elemet. Éppen ezért programunk a felhasználónak ezt a műveletet nem engedi, helyette inkább a törlést javasolja.
  - *opcionális elem* esetén szintén egy opcionálisba vonjuk össze az elemeket. Így az összevont feature igaz értéke szintén azt fogja jelenteni, hogy az összevont elemek közül legalább egy igaz.

Az absztrakció megvalósítása a gyakorlatban úgy működik, hogy a felhasználó által definiált szabályok által meghatározott absztrakt feature model leírását logikai problémává alakítjuk, majd ezt egy SAT-megoldó (SAT-solver) segítségével megoldjuk. Így a konkrét ábécé elemeit az absztrakt üzenetek formátumának megfelelő absztrakt ábécé eleire tudjuk leképezni. Ezt a 5.4 ábra szemlélteti.

5.3. Konkretizáció

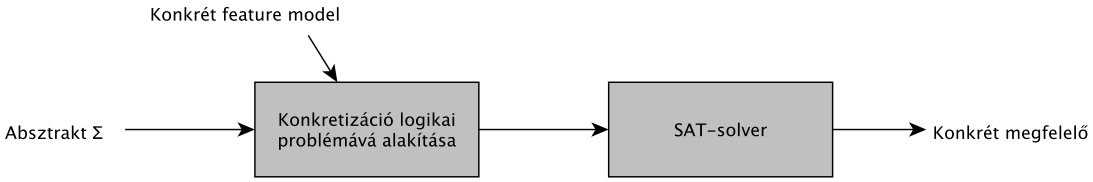
Konkretizáció során az absztrakt featuremodelekből állítunk elő konkrét featuremodeleket, mind a metamodell, mind konfiguráció szintjén. Konkretizálás során figyelembe kell venni



5.4. ábra. Absztrakció végrehajtásának vázlata

alapvető, a feature modellekre jellemző megkötéseket, a felhasználó által külön definiált és az absztrakciós műveletek során keletkezett egyéb megkötéseket is.

Az absztrakció megvalósításához hasonlóan, a konkretizáció is két lépésben történik. Először megfogalmazzuk a konkrét feature modeltől elvárt megkötéseinket a logikai problémák nyelvén, majd ezt a következő lépésben egy SAT-megoldó segítségével megoldjuk. Így az absztrakt ábécé elemeihez hozzá tudunk rendelni konkrét megfeleltetéseket. Ezt a folyamatot a 5.5 ábra szemlélteti.



5.5. ábra. Konkretizáció végrehajtásának vázlata

Az automatatanulás során az automata egyszeri megtanulása után az absztrakt üzeneteket mindig ugyanúgy konkretizáljuk. Ez determinisztikus és stabil absztrakció során elegendő is a komponens működésének megtanulásásra. Azonban arról nehéz meggyőződni, hogy az absztrakcióra ezek az állítások igazak-e. Ezért a tanulás folyamatát többször végrehajtjuk, az absztrakt ábécé elemeit - beállítható valószínűséggel - különbözőképpen konkretizálva. Ha az így megtanult automaták mind stabilak és egymással ekvivalensek, akkor jónak tekintjük az absztrakciót és a megtanult automatára a továbbiakban, mint a rendszer absztrakt működését leíró automatára tudunk hivatkozni. Az automatával szemben a következő elvárásokat tudjuk megfogalmazni a komponens működésének leírására:

1. Az így kapott Mealy-automatának helyesnek kell lennie abból a szempontból, hogy ha a megtanult komponens egy absztrakt bemenet szekvenciára produkálhat egy adott absztrakt kimenetszekvenciát, akkor egy ezt a viselkedést leíró útvonal kell, hogy legyen az automatában. Konkrét be- és kimenetsorozatokra is megfogalmazhatjuk ezt az elvárásunkat. Ha a megtanult komponens egy adott konkrét bemenetszekvenciára produkálhat egy adott konkrét kimenetszekvenciát, akkor az automatának a bemeneti üzenetek absztrakciójával kapott bemenetszekvenciára és a kimenetek absztrakciójának szekvenciájára is helyesnek kell lennie.

2. A megtanult automata teljes is a komponens működését tekintve, tehát az automatán végigjárható bármely útvonalra léteznie kell a komponensnek absztrakt be- és kimeneti szekvenciái és az ezt megvalósító konkrét szekvenciái is.

6. fejezet

Ellenőrzési módszerek

A rendszer absztrakt működésének a megtanulásán túl eszközünk lényeges funkciója még a rendszer új verziójának a funkcionális regressziós tesztelése, azaz a helyes rendszerfunkcionalitás megtartásának támogatása. Az ellenőrzés során többféle hibaesetet tudunk kiszűrni. Leghamarabb úgy derülhet fény egy nemkívánt módosításra, ha az új verzió által adott kimenetek az üzenetprotokollnak ellentmondanak (kivételek, vagy rosszul formázott adatok). Az általunk kidolgozott megközelítés ezt a feature model alapú, a modellre felírt megkötésekkel könnyen tudja vizsgálni. A megváltozott rendszer által adott kimenetek helyességénekvizsgálatán túl, fontos a funkciók helyességéről is meggyőződni. Ez tulajdonképpen az úgynevezett black box tesztelés (Black-box testing), avagy funkcionális tesztelés megvalósítását jelenti, minden új verzió esetén. Ennek sokféle megvalósítása lehetséges [3]. Mi ezek közül az állapotátmenet fedést biztosító tesztelést (State transition testing) választottuk. A rendszerről már előzetesen elkészített automata alapján lehetséges a tesztesetek előállítása. Az így kapott tesztesetekkel pedig ellenőrizhető az új verzió működése. A módosítás során előfordulhat az is, hogy a eszköz működése nemdeterminisztikussá válik vagy nem válik stabillá az absztrakcióra nézve. Mivel az absztrakció lényege a rendszer működésének csak bizonyos szempontokból vett megfigyelésének a lehetővé tétele, így elvárjuk, hogy a rendszer ilyen absztrakciós szempontok alapján determinusztikus és stabil legyen, tehát ilyenkor a felhasználónak más absztrakciót kell választania.

6.1. Tesztgenerálás

A tesztesetek generálásához a megtanult automatát használjuk fel. Ennek segítségével könnyedén lehet különböző tesztfedettséget biztosító bejárásokkal teszt szekvenciákat készíteni. Ilyen bejárások közül mi az átmenet- és az állapotfedő bejárásokat valósítottuk meg. A teljes átmenetfedő bejárás a rendszer funkcióit jobban lefedi, ráadásul a teljes állapotfedő bejárás teszteseteit is magába foglalja, ezért főként ezt használjuk.

Mindkét opciónál a szükséges tesztszekvenciákat az automata mélységi bejárásával állapítottuk meg. A tesztszekvenciákat mindig a kezdőállapotból indítottuk. Mivel az automata a rendszer viselkedését valamilyen absztrakción át mutatja be, így a generált teszteseteink is absztraktak lesznek. Sajnos előfordulhat, hogy a rendszer módosítása a meglévő absztrakciónk stabilitását elrontja. Ennek kiszűrése érdekében a tesztek bemeneteit és az azokra elvárt kimeneteket konkretizálva tároljuk el, a megváltozott új rendszeren is, így futtatva és ellenőrizve később. Annak érdekében, hogy tesztelés még jobban ki tudja szűrni a funkcionalitásban okozott hibákat, illetve, hogy könnyebben tudja detektálni az absztakció stablitásának megsértését, ezek a teszteset konkretizálások egy-egy absztrakciónak megfelelően többfélék is lehetnek. Esettanulmányunk során ez a konkretizálás az absztakt feature model üzenetek különböző konkretizálásainak a konfigurációival fog megvalósulni.

Az automatán az átmeneteket meg tudjuk feleltetni absztrakt feature model üzenet konfigurációknak. Ezeket konkretizálva kapunk konkrét feature model konfigurációkat, amik már könnyen átlakíthatóak a sakkóra be- és kimeneteire. A tesztesetek elmentése során a kérdéseket és az elvárt válaszokat is konkrét konfigurációkként tároljuk el.

6.2. Automata-összehasonlítás

Mint azt már korábban is bemutattuk, a tesztesetek sikeres lefutása sajnos még nem biztosítja a rendszer funkcionalitásának sértetlenségét. Erre példát is láthattunk korábban az 4.3 fejezet 5. példájában. Éppen ezért a tesztelést még kibővítjük egy újabb automata tanulással is, hogy a régi és új verzió automatáit is össze tudjuk még hasonlítani. Ahhoz is szükségünk van az automataekvivalencia ellenőrző algoritmusra, hogy a rendszer egyes funkcióinak működését meg tudjuk tanulni. Ezzel az algoritmussal tudjuk kiszűrni ugyanis, ha a felhasználó által megfogalmazott absztrakció nem lenne stabil. Nem stabil absztrakció esetén különböző konkretizációkon keresztül tanult automaták nem lesznek ekvivalensek, ilyenkor jelezni kell a felhasználó számára, hogy finomítsa az absztrakciót.

Ilyen evivalenciatesztelésre sokféle algoritmus létezik/megvalósítható. Mi a Mealy-automaták minimalizását [19] követően egy gráfizomorfia-vizsgálattal ellenőrizzük az egyezőséget. A minimális, determinisztikus automata egy nyelvre egyedi, kanonikus, így ha az őket reprezentáló címkézett gráfok izomorfak, az automaták által reprezentál nyelvek is evivalensek lesznek.

A LearnLib [14] keretrendszer L* algoritmussal [8] való tanulás során minimális automatát tanul meg, mi ezt az előnyét kihasználtuk az eszköznek, így elegendő volt csak a gráfizomorfia vizsgálatot megvalósítanunk. A gráfizomorfia vizsgálat megvalósítását az alábbi vázlatos leírás alapján készítettük el:

- Mindkét gráfnak a kezdőcsúcsából indítjuk az algoritmust, a kezdőcsúcsokat megfeleltettük egymásnak, ezt a megfeleltetést eltárolva.
- Következő lépésben a két automatán egymásnak megfeleltetett éleken (Mealy-automata esetén egyező inputok) továbbléptünk egy következő csúcsba. Ezeket a csúcsokat is egymáshoz rendelve.
- Az előző lépést egészen addig ismételjük, míg olyan állapotokba nem érünk, amiknek már szerepelniük kellett a bejárás során .
- Ha ezek az állapotok nincsenek egymáshoz rendelve, akkor a két automatánk nem ekvivalens, az algoritmus véget ér. Ha meg vannak feleltetve egymásnak, akkor tovább folytatjuk az algoritmust a még be nem járt éleken.
- Az algoritmus véget ér és ekvivalenciát állapít meg, ha már nem találunk több bejáratlan élt és eddigi bejárásaink során minden csúcs-párt meg tudtunk feleltetni egymásnak.

7. fejezet

Megvalósítás

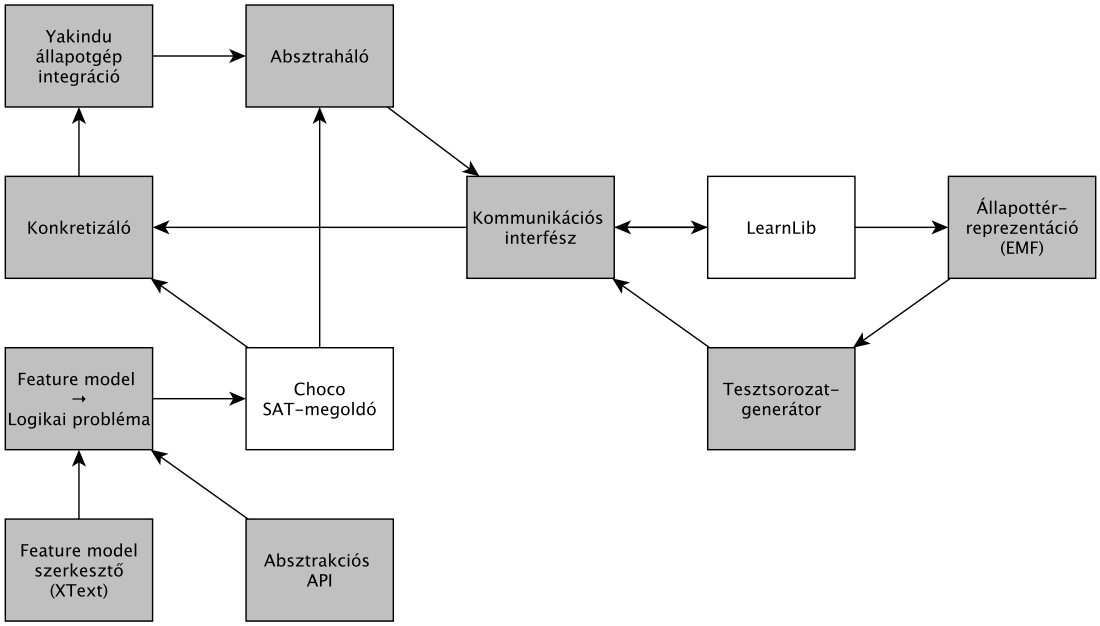
Az alábbi fejezetben bemutatjuk az általunk fejlesztett prototípus tanuló eszköz megvalósításával kapcsolatos részleteket és a különböző komponensek teljesítményének a hatékonyságát is szemléltetjük.

7.1. Tanuló keretrendszer architektúrája

A 7.1. ábrán látható az általunk elkészített alkalmazás architektúra ábrája és legfőbb komponensei. Az általunk fejlesztett komponenseket szürkével jelöljük, míg a fehérrel jelzett részek kész komponensek integrációját jelölik. A komponenseket közvetlenül integráltuk Eclipse fejlesztőrendszerbe, ami a legelterjedtebb modellező eszköz, így a fejlesztők közvetlenül a tervezőeszközben használhatják a tanuló és tesztelő algoritmusunlat.

Az algoritmusunk megvalósításának az alábbi fő komponensei vannak:

- **Feature model szerkesztő**: Feature modelek leírására készített szerkesztőfelület XText-ben [18] megvalósítva. A szerkesztőfelületen a tervezőmérnök specifikálhatja az általa vizsgált komponens ki és bemeneteit (7.2. ábra) és lehetőséget biztosít további keresztirányú megkötések megfogalmazására is (A 7.3).
- **Absztrakciós API**: Ennek a komponensnek a segítségével tudja a felhasználó az feature modelleken értelmezett absztrakciót definiálni.
- **Feature model → Logikai probléma**: Az absztrakcióra, konkretizációra és konfigurációra megfogalmazott elvárások logikai problémára fordításáért felelős komponens. Bemenete egy (esetlegesen absztrakciókkal elláttott) feature model probléma, kimenete pedig egy logikai probléma. Ezen kívül logikai probléma megoldását képes feature modell tulajdonságokként interpretálni.
- **Choco SAT-megoldó**: Logikai problémákat fejlett SAT-megoldókkal meg lehet oldani, mi a Choco-t [13] választottuk munkánk során.
- **Konkretizáló** és **Absztraháló**: Ezek a komponensek valósítják meg a SAT-megoldó komponens segítségével a lekérdezések és válaszüzenetek absztrakcióját és konkretizációját.
- **Yakindu állapotgép integráció**: Esettanulmányunk során a megtanult rendszert egy YAKINDU [20] állapotgép definiálta, az ezzel való kommunikáció interfészének biztosítása volt a komponens feladata.
- **Kommunikációs interfész**: A komponens felelőssége a generált tesztesetek konkrét lekérdezés-válasz konfigurációi párjainak az elmentése volt, a regressziós teszteléshez segítséget nyújtva. A tanuló algoritmus ezen a komponensen keresztül éri el a

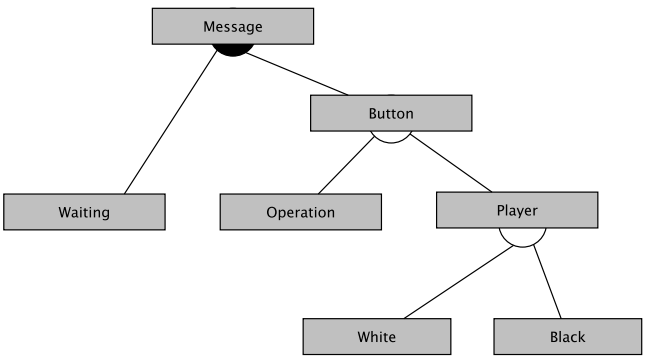


7.1. ábra. Keretrendszer felépítése

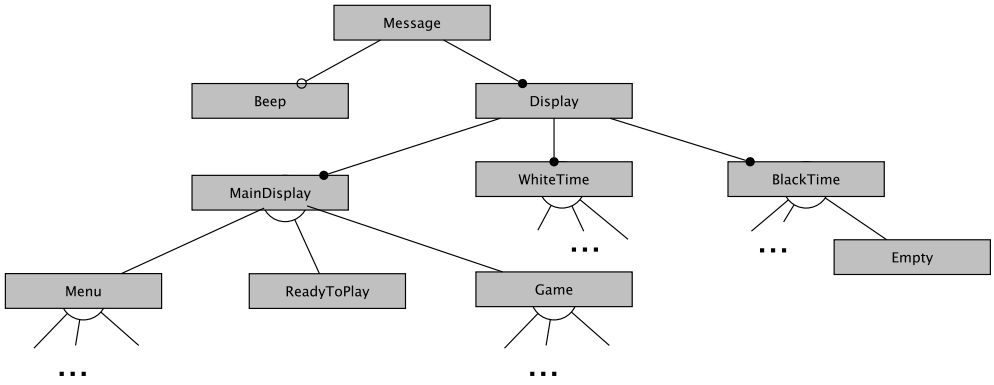
konkrét rendszert, illetve az absztrakció és a konkretizáció is ezen a kommunikációs interfészen keresztül valósul meg.

- **LearnLib**:A LearnLib [14] az automatatanulást megvalósító beintegrált komponens.
- **Állapottérreprezentáció**: Általunk megvalósított Eclipse Modeling Frameworkben [17] készített Mealy-automata model. Tesztgenerálásainkat, automataösszehasonlításainkat ennek a modellnek a segítségével végeztük.
- **Tesztsorozatgenerátor**: A komponens a megtanult automatán végzett tesztsorozatgenerálásért felelős. A tesztsorozatot képes absztrakt és konkrét állapottér felett is létrehozni, absztrakt állapottér esetén az absztrakt tesztbemenetek a kommunikációs és konkretizáló interfészen keresztül valósítható meg.

```
fmodel QuestionAbstract {
 root Message cont {
  Waiting
  Button xor {
   Operation
   Player xor {
    White
    Black
   }
  }
 }
}
```



7.2. ábra. Absztrakt kérdés feature modelje.




```
implies(ref ReadyToPlay, not(ref BEmpty))
implies(ref ReadyToPlay, not(ref WEmpty))
```

7.3. ábra. Absztrakt válasz feature modelje.

7.2. Az elkészült rendszer helyességének ellenőrzése

Ahogy a tesztesetekkel lehet tesztelni a vizsgált rendszert, úgy a tesztgenerátor helyességét is ellenőrizhetjük. A fejlesztett eszköz által megvalósított tesztek könnyen áttekinthetőek, a tesztek végrehajtása során egyértelműen kiderül, hogy a tesztgenerátorunk működésében, vagy a vizsgált rendszerben van-e a hiba.

Az általunk megvalósított rendszer komponenseit külön-külön több (a mérési esetekhez hasonlító) feladatra kipróbáltuk, illetve a rendszer egészét egy komplex esettanulmányon sikeresen ellenőriztük.

7.3. Mérések

Hogy megvizsgáljuk a prototípus megvalósításunkon teljesítményét, néhány kezdeti mérésekt végeztünk. Megfigyeléseink szerint az algoritmusnak futásidő szempontjából három kritikus lépése van: állapotgéptanulás, absztrakció, tesztgenerálás. Ezért ezt a hármat különböző méréssekkel vizsgáljuk, hogy hogyan skálázódik, milyen a teljesítménye.

7.3.1. Automatatanulás

Az automatatanulás teljesítményvizsgálata során az alábbi kérdés megválaszolást célozzuk:

K1 Mennyi időt vesz igénybe az automatanulás különböző méretű állapottérrel rendelkező rendszerekre?

A kérdés megválaszolására egy olyan tanulandó rendszert készítettünk amelynek az állapottere skálázhatóan növelhető. A mérés során azt vizsgáltuk, hogy mennyi idő alatt tudja megtanulni az általunk külső komponensként alkalmazott LearnLib a rendszer viselkedését. Mérési esetként a sakkórában előforduló különböző számparaméterek beállítását mértük. Egy 4 számlálóból álló automata volt a megtanulandó rendszer. Kiszámítottuk, hogy a rendszer hány állapotból áll, és mértük a futásidőt több alkalammal. Ezen mérések mediánját tüntettük fel.

Látható, hogy a példában vett viszonylag kevés állítási lehetőség – kezdőidő és jutalomidő állítása a két felhasználónak külön – is nagy állapotteret ad, amelynek absztrakció nélküli megtanulására nincs lehetőség. Ezek alapján a válaszunk a **K1** kérdésre:

Számlálók maximuma	Állapotok száma	Futásidő (s)
5	625	0,398
10	10000	3,734
15	50625	20,178
17	83521	36,088
20	160000	> 600,000

7.1. táblázat. 4 számlálós rendszer tanulása

V1 Bár a tanulóalgoritmus az állapotok számával jól léptékeződik, gyakorlati alkalmazásoknál egyszerű rendszerek is könnyen nagy állapottérrel rendelkezhetnek, ezért a korszerű tanuló algoritmusok is csak egyszerű automaták megtanulására képesek.

Mérések eredménye is azt mutatja, hogy összetett rendszerek tanulásához szükséges absztrakció alkalmazása, hiszen a konkrét rendszerek ennél jóval nagyobb állapottérrel rendelkezhetnek.

7.3.2. A lehetséges konfigurációk felsorolása

Második kérdésünk éppen ezért az absztrakciós komponens teljesítményét vizsgálja:

K2 Hogyan skálázódik a rendszer bemeneti ábécéjének felsorolása?

A kérdés megválaszolásához több különböző méretű feature modellt állítottunk elő, és megvizsgáljuk hogy képes-e az általunk használt logikai következtető a lehetséges konfigurációk előállítására. Ezért a rendszerünkben használt bemeneti feature modelt többszörözve készítettünk különböző méretű modelleket. Számításokat végeztünk, hogy az így kapott modelleknek hány konfigurációja létezik, majd mértük az ezek felsorolásához szükséges időt.

Feature-ök száma	Lehetséges konfigurációk száma	Futásidő (s)
13	12	0,002557517
25	168	0,006000343
37	2196	0,039701065
49	28560	0,645220063
61	371292	kevés memória

7.2. táblázat. Konfigurációk felsorolása

A mérési eredményeket a 7.2 táblázat tartalmazza. A kiindulási feature modelnél néhányszor nagyobb modelleknél már nagyságrendekkel változik a lehetséges konfigurációk száma. A korlátot itt nem a futásidő, hanem az ábécé eltárolásához szükséges tármennyiség szabja meg. Innen látható, hogy már az ábécé eltárolása is sok erőforrást igényel, azoban ennyi féle szimbólummal cimkézett állapotgépeknél már a tanulás okozna teljesítményproblémákat.

V2 Futásidő szempontjából az egész absztrakt abc generálási ideje másodperc alatti, és másfél nagyságrenddel gyorsabb mint egy hasonló mérettel rendelkező rendszer tanulási ideje.

Ezek alapján érdemes absztrahálni a tanult rendszerünket, már csak azt kell megvizsgálni, hogy tanulási folyamatot nem hátráltatja-e.

7.3.3. A konkretizáláshoz szükséges kisszámú konfiguráció felsorolása

A konfigurációk felsorolására nemcsak az ábécé előállításához van szükség, hanem a tanulás közbeni konkretizációs leképezés során is.

K3 Mennyi idő alatt lehet a konkretizáláshoz szükséges konfigurációkat előállítani?

A konfigurációk előállítása során szükséges, hogy az egyes absztrakt konfigurációknak megfelelő konkrét konfigurációk közül néhányat generáljunk. Ehhez az előző feature modeleknek megfelelő konfigurációk közül adott számút állítottunk elő és mértük az ehhez szükséges időt.

Konfigurációk száma / Feature-ök száma	1	5	10
	Futásidő (s)		
13	0,000623	0,000878	0,000703
25	0,000376	0,000743	0,000509
37	0,000406	0,000854	0,000761
49	0,000674	0,000975	0,000650
61	0,000679	0,000711	0,000984
73	0,000816	0,000988	0,001000
85	0,000876	0,000986	0,001006
97	0,004968	0,005252	0,006018
109	0,001860	0,001791	0,001989
121	0,001984	0,001317	0,001607
133	0,002104	0,001460	0,001897
145	0,001694	0,001572	0,001575
157	0,001998	0,003000	0,002829
169	0,001967	0,002979	0,003029
181	0,001959	0,003281	0,002564
193	0,001989	0,002265	0,002644
205	> 600		

7.3. táblázat. Első k db. konfiguráció felsorolása

Kisszámú konfiguráció generálásához szükséges idő jóval kevesebb volt, mint ha az összes megoldásra van szükség, és nagyobb modellekre is lefutott. Így a konkretizáláshoz szükséges konfigurációk előállítására jól használható.

V3 A konkretizációk előállítása minimális extra munkával jár, ezért a technika jól alkalmazható a tanulás során.

Mérési eredményeinket összefoglalva: **V1** tapasztalatai alapján komplex rendszerek megtanulása szükségessé teszi absztrakciók alkalmazását, amire (**V2** és **V3** alapján) a feature modellezés egy hatékony módszernek bizonyult.

8. fejezet

Összefoglalás

Dolgozatunkban két érdekes terület kombinációját vizsgáltuk, azaz hogy hogyan lehetne automatatanulás segítségével támogatni a tesztelés folyamatát, azon belül is módszerünket a regressziós teszteléshez dolgoztuk ki. Munkánk során megvizsgáltuk, hogy szoftver komponensek specifikációját hogyan lehetne hatékonyan kinyerni az implementációból, és ez alapján kidolgoztunk egy absztrakciós módszert, amely támogatja a tanulás során a felhasználó által definiált ekvivalencia osztályok alapján az adatvezérelt viselkedések absztrakcióját. Ezáltal egy lépést tettünk valódi rendszerek szoftver komponenseinek megtanulása felé. A tanulási algoritmust illesztettük a regressziós tesztelési folyamatba, amely során:

- Támogatjuk akár harmadik féltől származó szoftver komponensek alapján is a szoftver automata modelljének automatikus származtatását.
- Az elkészült automata modell alapján teszteket generálunk, amely felhasználható fókuszáltan a regressziós tesztelés támogatására.
- A fejlesztési fázis lezárása utána lehtőséget adunk a szoftver kiindulási és a módosított verziójának összevetésére a belőlük megtanult automatákon keresztül.

Elméleti eredményeink az alábbiak:

- Kidolgoztunk egy módszert, amely az adat jellegű viselkedések absztrahálásán keresztül támogatja a szoftver komponensek megtanulását.
- Nyelvet fejlesztettünk az automatatanulás támogatására, amely nyelven egyrészt a bemeneti és kimeneti adatok specifikálhatóak, továbbá az absztrakció és a konkretizáció definiálható, ezáltal támogatva a tanulás fókuszálását a szoftver komponens releváns aspektusaira.
- Algoritmusokat fejlesztettünk az absztrakció és konkretizáció problémájának automatikus végrehajtására.
- Kidolgoztunk egy módszert, amely támogatja a regressziós tesztelés automatizálását automatatanulás és ellenőrzési algoritmusok segítségével.

Gyakorlati eredményként elkészítettünk egy keretrendszer prototípusát, amely támogatja az absztrakció definiálását, az automatatanulást és a tesztek származtatását és automatellenőrzések végrehajtását. A keretrendszer egy népszerű automatatanulási könyvtáron alapul, amelyet kiegészítettünk azokkal a szolgáltatásokkal, amelyekkel beilleszthettük nem csak a keretrendszerünkbe, de a regressziós tesztelési folyamatba. Munkánk során példák segítségével vizsgáltuk megközelítésünk gyakorlati működését és használhatóságát.

8.1. Jövőbeli tervek

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 az absztrakciós nyelv adta konfigurálhatóságot az absztrakció finomítás vezérlésében. Gyakorlati továbbfejlesztési lehetőség a feature modellekben a numerikus értékek leírása és absztrakciója, melyre intervallum logikát tudnánk használni. Legvégül szeretnénk az általunk készített rendszert az oktatásban is felhasználni, a hallgatók házifeladatának az ellenőrzését tudnánk vele segíteni.

Köszönetnyilvánítás

A dolgozat az MTA-BME Lendület Kiberfizikai Rendszerek Kutatócsoport szakmai támogatásával, továbbá

Az Emberi Erőforrások Minisztériuma ÚNKP-16-1-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] Dana Angluin: Learning regular sets from queries and counterexamples. *Information and computation*, 75. évf. (1987) 2. sz., 87–106. p.

[3] Boris Beizer: *Black-box testing: techniques for functional testing of software and systems*. 1995, John Wiley & Sons, Inc.

[4] David Benavides – Pablo Trinidad – Antonio Ruiz-Cortés: *Automated Reasoning on Feature Models*. Berlin, Heidelberg, 2005, Springer Berlin Heidelberg, 491–503. p. ISBN 978-3-540-32127-9. URL `http://dx.doi.org/10.1007/11431855_34`.

[5] Budapesti Műszaki és Gazdaságtudományi Egyetem: *Rendszermodellezés tantárgy (VIMIAA00)*. `https://inf.mit.bme.hu/edu/courses/remo`.

[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] Maximilian X Czerny: *Learning-based software testing: Evaluation of Angluin's L* algorithm and adaptations in practice*. PhD értekezés (National Research Center). 2014.

[9] Paul Fiterău-Broştean – Ramon Janssen – Frits Vaandrager: Combining model learning and model checking to analyze tcp implementations. In *International Conference on Computer Aided Verification* (konferenciaanyag). 2016, Springer, 454–471. p.

[10] Kyo C Kang – Sholom G Cohen – James A Hess – William E Novak – A Spencer Peterson: Feature-oriented domain analysis (foda) feasibility study. Jelentés, 1990, DTIC Document.

[11] George H Mealy: A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34. évf. (1955) 5. sz., 1045–1079. p.

[12] Marcilio Mendonca – Andrzej Wąsowski – Krzysztof Czarnecki: Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference* (konferenciaanyag). 2009, Carnegie Mellon University, 231–240. p.

[13] Charles Prud'homme – Jean-Guillaume Fages – Xavier Lorca: *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. URL `http://www.choco-solver.org`.

[14] 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.

[15] Muzammil Shahbaz – Roland Groz: Inferring mealy machines. In *International Symposium on Formal Methods* (konferenciaanyag). 2009, Springer, 207–222. p.

[16] Bernhard Steffen – Falk Howar – Malte Isberner és mások: Active automata learning: From dfas to interface programs and beyond. In *ICGI* (konferenciaanyag), 21. köt. 2012, 195–209. p.

[17] The Eclipse Project: *Eclipse Modeling Framework*. `//www.eclipse.org/emf`.

[18] The Eclipse Project: *Xtext*. `http://www.eclipse.org/Xtext/`.

[19] Bruce W Watson: A taxonomy of finite automata minimization algorithms. 1993.

[20] Yakindu Statechart Tools: *Yakindu*. `http://statecharts.org/`.

[21] Shin Yoo – Mark Harman: Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22. évf. (2012) 2. sz., 67–120. p.