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

Automatizált ellenőrzési módszerek kiértékelése és kidolgozása kritikus szoftverrendszerekhez

Habilitációs tézisfüzet

Micskei Zoltán

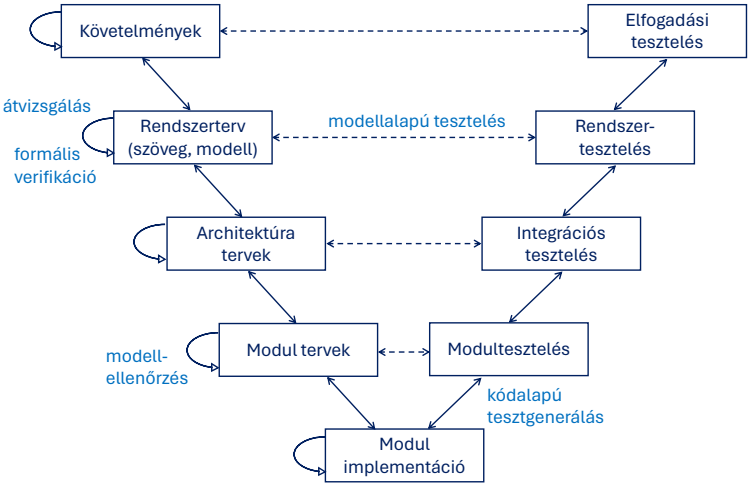
egyetemi docens, PhD

Budapest, 2024

1. Bevezetés

A szoftverintenzív rendszerek az élet egyre több területén meghatározóvá váltak. Nemcsak a virtuális, hanem a fizikai világban működő rendszerek funkcióit is egyre inkább a bennük lévő szoftver határozza meg. Ezen szoftverek *helyes*, *megbízható működése* így kiemelt fontosságú. Különösen igaz ez kritikus rendszerek esetén, például az autó-, repülő- vagy vasútiparban. Ilyen rendszerek esetén egy-egy hibának komoly anyagi vagy emberéletet veszélyeztető következménye lehet.

A rendszertervezési- és fejlesztési folyamatokban éppen ezért nagyon sokféle ellenőrzési módszert és eszközt alkalmaznak. Ezek egy része *statikus*, például a készülő specifikáció vagy rendszermodellek átvizsgálása vagy a forráskód futtatás nélküli statikus analízise. A másik részük *dinamikus*, azaz végrehajtja a vizsgált munkaterméket, például szimuláció vagy tesztelés keretében [IEE10].



1. ábra. Fejlesztési tevékenységek és ellenőrzési módszerek illusztrálása

Az 1. ábra bemutatja, hogy tipikusan milyen lépések és termékek találhatók egy rendszerfejlesztési folyamatban, valamint ehhez milyen ellenőrzési módszerek kapcsolódhatnak. Az ábra a V-modellnek[1] megfelelően rendezi el az egyes tevékenységeket, azonban hasonló lépések a legtöbb szoftverfejlesztési módszertanban megtalálhatók (esetleg más névvel és fókusszal).

A V-modell bal oldalán haladva a követelményekből kiindulva a fejlesztés során egyre részletesebben megismerjük, megtervezzük és implementáljuk a készítendő rendszert. Az egyes tevékenységeknél szereplő önhurok jelzi, hogy már az adott lépésnél sokféle ellenőrzést végezhetünk (például az összegyűjtött követelményeket

[1] A V-modellnek nagyon sok változata létezik, majd minden iparág és szabvány kidolgozta a saját változatát. A jelen ábra pusztán illusztráció, nem ad teljes képet egyik fejlesztési módszertanról sem.

átvizsgálhatjuk és javíthatjuk). Továbbá, ahogy a szintek közötti oda-vissza nyíl jelzi, ez sokszor egy iteratív folyamat, ahol az eredmények alapján visszatérünk egy-egy korábbi termékhez (például a részletes specifikáció elkészítése során rájövünk, hogy egy követelményt pontosítani kell). A V-modell jobb oldalán az egyre összetettebb tesztelési lépések szerepelnek. Azonban ezekkel sem kell megvárni a teljes rendszer elkészítését. A szaggatott nyilak jelzik, hogy adott szintű tesztek megtervezése jóval korábban elkezdődhet; már a tesztek megtervezése során is rengeteg hibára fény derülhet a kivételek és hibalehetőségek végiggondolásával.

Az egyre komplexebb és magasabb minőségi elvárásokkal rendelkező rendszerek fejlesztését támogatandó sokféle módszer és technika jelent meg, amiknek célja vagy a fejlesztési idő és munkaigény csökkentése, vagy a készülő rendszer minőségének javítása, tipikusan a hibalehetőségek korai azonosításával. A tézisfüzetben bemutatott eredmények szempontjából ezen módszerek két családja releváns.

- *Modellalapú fejlesztés* [BCW12]: A korábbi, döntően dokumentumalapú fejlesztés helyett egyre több iparágban a különböző modellezési nyelveken készített rendszermodelleket helyezik a középpontba. Ezek a például UML [OMG17] vagy SysML [OMG19a] nyelven készített modellek strukturált, feldolgozható és analizálható formában gyűjtik össze a rendszerrel kapcsolatos elvárásokat, tervezői döntéseket, és fokozatosan finomítva lehetőségeket adnak a rendszer felépítésének és viselkedésének megadására.

- *Automatizált ellenőrzés* [Ana+13]: Az úgynevezett verifikáció & validáció (V&V) körébe tartozó tevékenységek a fejlesztési folyamat kulcsfontosságú, de rendkívül idő- és erőforrásigényes feladatai. Az irodalomban javasolt megoldások többféle módon is igyekeznek csökkenteni az ellenőrzés költségét. Egyrészt bizonyos feladatok jól algoritmizálhatók, így azokra automatikus verifikációs módszereket dolgoztak ki. Másrészt, a V&V tevékenységhez szükséges bemenetek és termékek egy része generálható is (például a rendszermodellből a rendszertesztek egy része származtatható).

A fenti területeken belül az alábbi módszerekre helyeztem a hangsúlyt. A rendszerek diszkrét eseményekre való válaszainak specifikálására egy elterjedt *modellezési nyelv* az *állapotgépek* nyelve. Az állapotgépek a véges automaták formalizmusát többek között hierarchia és párhuzamosság megadásával bővítik ki. Az UML szabvány a nyelv elemkészletét és szintaktikáját megadja metamodellek és kényszerek segítségével [BKP20], azonban egy összetett modellezési nyelvnél legalább ilyen fontos a *szemantika* precíz megadása. A modellezési nyelv szemantikája adja meg, hogy egy modell példány „mit jelent”, tipikusan valamilyen *szemantikai tartományra* (semantic domain) való leképezéssel [HR04]. A modellek helyes felhasználása és ellenőrzése szempontjából elengedhetetlen, hogy a nyelv szemantikáját ugyanúgy értelmezzék a modelleket használó mérnökök és eszközök.

Az ellenőrzési módszereken belül pedig több technikát is vizsgáltam.

- A *formális verifikáció* matematikai módszerek segítségével igyekszik egy rendszer vagy modell helyességét belátni. Ennek egyik speciális módszere a *modellellenőrzés* (model checking) [Cla+18], amely során a vizsgált modell állapotterét szisztematikusan bejárjuk, és azon valamilyen tulajdonság meglétét igazoljuk vagy pedig egy ellenpéldát szolgáltatunk.

- A *tesztelés* ezzel szemben olyan technikák csoportja, ahol a rendszert vagy modellt bizonyos kritériumok alapján kiválasztott, de véges sok bemenettel és helyzetben végrehajtjuk, és a megfigyelt kimenetek és viselkedés alapján vonunk le következtetéseket. Tesztelés során kiemelt kérdés, hogy mi alapján döntjük el, hogy a megfigyelt viselkedés elvárt-e; ebben segítenek a *teszt orákulumok* (test oracle) [Bar+15] heurisztikái. A teszteseteket vagy azok bemeneteit lehet modellekből származtatni (*modellalapú tesztelés*) vagy akár a forráskódból generálni (*kódalapú tesztgenerálás*).

1.1. Célkitűzés és kutatási kérdések

Az ismertetett modellezési és ellenőrzési technikák sok évtizedes múltra tekintenek vissza. Egyes területeken komoly sikereket értek el, például mikrokernel formális verifikációja [Kle+14] vagy protokollok dokumentációjának modellalapú tesztelése [Gri+11] terén. Azonban a mérnöki gyakorlatban még nem terjedt el széleskörűen a formális verifikáció vagy a tesztgenerálás használata. Az elmúlt 15 évben kutatási érdeklődésem középpontjában az állt, hogy miért korlátozott ezen ellenőrzési módszerek használata, és hogyan lehetne az elterjedésüket segíteni.

Célkitűzés Fejlett modellalapú és automatizált ellenőrzési módszerek használhatóságának növelése a mérnöki gyakorlatban.

Természetesen ezzel a kérdéssel már korábban is sokan foglalkoztak, és a terület egyik fő kihívásának tartják. Parnas provokatív című cikkében [Par98] arra hívja fel a figyelmet, hogy ezen módszerek praktikusságát kellene növelni, valamint beemelni az oktatásba a szoftverfejlesztés szerves részeként. Az alkalmazható formális módszerekről szóló cikk [GPW23] kiemeli, hogy ezen módszereknek skálázhatónak és könnyen használhatónak kell lenniük. A modellalapú fejlesztés nagy kihívásait vizsgáló összefoglaló [Buc+20] is kulcsfontosságúnak tartja az eszközök képességeinek fejlesztését, továbbá a modellezés emberi tényezőit.

A fenti általános célkitűzést három nézőpontból vizsgáltam, amikkel kapcsolatos eredmények segíthetnek a módszerek elterjedésében.

1. kérdés. Konzisztensen interpretálják a mérnökök és az eszközök készítői a rendszermodellek vagy a generált tesztek szemantikáját?

Bár nagyon sok cikk a modellezési és ellenőrzési eszközök algoritmikus jellemzőire fókuszál, tapasztalatom szerint legalább olyan fontos, hogy ezeket az eszközöket végső soron emberek használják a rendszerek fejlesztése során. Kutatásaim egy részében azt vizsgáltam, hogy a létrehozott rendszermodelleket vagy az eszközök által generált teszteket mennyire konzisztensen értelmezik a mérnökök, és ezen megértésbeli különbségek milyen problémákat okozhatnak.

2. kérdés. Milyen hiányosságai vannak a jelenlegi automatizált ellenőrzési eszközöknek, amik akadályozzák ezen módszerek szélesebb körű elterjedését?

Az irodalomban javasolt automatizált ellenőrzési módszereket szoftver eszközökben valósítják meg, és a mérnökök ezeket az eszközöket tudják használni a

munkájuk során. Így egy-egy adott módszer elméleti korlátai mellett fontos tényező az is, hogy maguknak az eszközöknek milyen korlátai és hiányosságai vannak. Például a vizsgált modellezési vagy programozási nyelv mely részhalmazát képesek kezelni, vagy melyik az a jellemző, ami különösen kihívást jelent egy tesztgeneráló vagy verifikációs eszköznek.

3. kérdés. Milyen új ellenőrzési módszerekkel és eszközökkel lehet segíteni az azonosított részterületeken felmerült kihívásokon?

Az első két kérdés vizsgálata során azonosított hiányosságok és kihívások leküzdéséhez új módszereket és eszközöket kellett kidolgozni. Kutatásaim során olyan új módszereket javasoltam, ami egy-egy specifikus probléma megoldásában segít, például a SysML nyelv egy bizonyos részhalmazának ellenőrzésében vagy szimbolikus végrehajtás alapú tesztgenerálás hatékonyságának javításában.

A három nézőponthoz kapcsolódó kérdések és kutatások iteratív módon, folyamatosan egymásra hatva jelentek meg kutatási munkámban[2]. Azaz a kísérletek során azonosított hiányosságok új módszerek szükségességére világítottak rá, az új eszközök kidolgozása és kiértékelése rámutatott, hogy nemcsak az eszközök algoritmikus korlátai, hanem a modellekhez vagy tesztekhez tartozó szemantika megértésének nehézsége is a problémák oka. A három különböző nézőpont miatt többféle kutatási módszert is kellett alkalmaznom vizsgálataim során.

1.2. Kutatási módszerek

A „software engineering”[3] kutatási módszertani szempontból is érdekes határterület [SF18]. A terület számítástudományi gyökerei miatt az algoritmusok helyességének bizonyítása és a formális logikára való támaszkodás hangsúlyos a formális módszerek kutatásában. Ugyanakkor a klasszikus mérnöki területek statisztikai alapú kiértékelési módszerei is elterjedtek, nemcsak szimulációs környezetben, hanem a szoftvereket különböző környezetben futtató kiértékelések (benchmark) során. Végezetül legalább ilyen hangsúlyossá vált az utóbbi egy-két évtizedben a szoftverfejlesztés emberi tényezőinek vizsgálata, így a társadalomtudományokban kidolgozott kutatási módszereket is egyre gyakrabban adaptálják.

Kutatásaim során ennek megfelelően többféle kutatási módszert alkalmaztam.

- A modellezési nyelvek vizsgálata során a denotációs vagy operációs szemantikáikat *analizáltam*, azok helyességét és konzisztenciáját ellenőriztem.
- Az eszközök kiértékelése különböző *empirikus módszerek* használatával történt. Megterveztem többféle kontrollált kísérletet [Woh+12].
- Külön kiemelendő, hogy a kísérletek egy része *emberi résztvevőket* alkalmazott [KLB13], ahol a résztvevőknek egy kísérleti környezetben alkalmazni kellett egy tesztgeneráló eszközt. Az emberi résztvevőket alkalmazó kísérleteknek az eredményei legtöbbször a résztvevők véges száma miatt korlátozottan általánosíthatók, de a belőlük származó tanulságok kritikusak az adott eszköz vagy módszer valós problémáinak megértéséhez.
- Az eszközök létrehozása a „*design science*” [Ral+21] elvei mentén történtek.

[2] A tézisfüzet azonban ezeket egy adott sorrendbe rendezve tudja csak bemutatni.
[3] Talán a szoftvermérnökség lenne a legjobb fordítása, de ez kevéssé terjedt el.

2. Új tudományos eredmények

2.1. Modellek és tesztek szemantikájának megértése

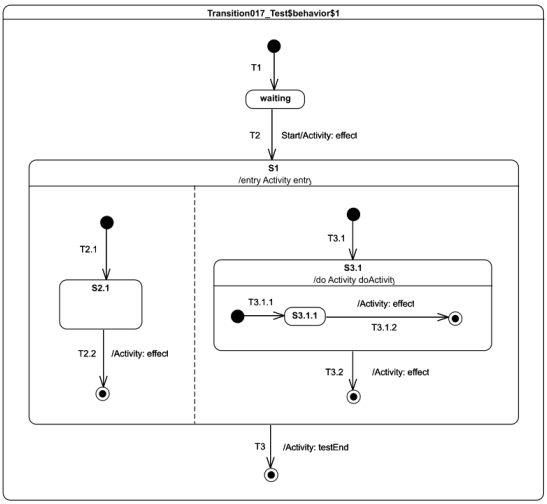
A kapcsolódó kutatásokban azt szokták gyakran elhanyagolni, hogy a modellezési és ellenőrzési módszereket és eszközöket végső soron emberek használják majd. Így például hiába fogalmazza meg egy rendszermodell precízen a megvalósítandó viselkedést, ha annak szemantikai részleteit máshogy értelmezi az implementációt végző mérnök és a szimulációt végrehajtó eszköz, akkor inkonzisztens vagy hibás működést vihet be a rendszerbe. Hasonló módon, hiába képes egy tesztgeneráló eszköz egy olyan tesztet generálni, ami a vizsgált programban lévő hibát érinti, ha az eszközt futtató mérnök a teszt kimenete alapján nem veszi észre, hogy hiba van, vagy nem megfelelően értelmezi a kimenetet, akkor azt a hibát nem fogják javítani. A szemantika megértésének ilyen kihívásait két területen vizsgáltam.

Modellezési nyelvek A modellezési nyelvek szemantikájának definiálására sokféle módszert és formalizmust dolgoztak ki [VP03]. Kutatásaim során az UML állapotgépek példáján keresztül vizsgáltam a problémát, mert az állapotgépek használata elterjedt a kritikus és beágyazott rendszerek tervezése során [AGD18]. Az UML specifikáció [OMG17] informális, természetes nyelvű szöveggel írja le a szemantika fontosabb elveit (például, hogy az állapotgéphez beérkező események egy üzenetsorba kerülnek, és innen egyesével az úgynevezett „run-to-completion" elv mentén dolgozza fel az állapotgép). Ez a leírás elégséges, ha az állapotgépeket illusztrációként és informális leírásként használjunk. Azonban, ha az állapotgépeket végre szeretnénk hajtani [CMS19] szimuláció, tesztgenerálás vagy formális verifikáció [LMM99; Var04] céljából, akkor egy precízebb leírásra van szükség. Az elmúlt évtizedekben rengeteg cikk javasolt különböző szintű formális szemantikát az állapotgépekhez. Ezekből 61 cikket elemez egy friss áttekintő cikk [And+23], amelyek többek között absztrakt állapotgépekre, hierarchikus automatákra, Petri-hálókra vagy különböző modellellenőrző és tételbizonyító eszközök bemenetére képezik le az UML állapotgépek nyelvét.

Felismerve az informális leírás hiányosságait, az UML specifikációt kidolgozó OMG szervezet is elkészített egy új, precíz szemantikát a *Precise Semantics of UML State Machines (PSSM)* [OMG19b] specifikáció keretében. A PSSM-ben definiált szemantika operációs jellegű, ami egy absztrakt végrehajtási modellt definiál az állapotgépekhez. A PSSM az UML metamodellt kiterjeszti a szemantikai fogalmak explicit reprezentációjával (például az állapotkonfiguráció modellezésével), definiálja a végrehajtási modell állapotát megváltoztató egyes operációkat, valamint megad egy tesztkészletet, ami egyrészt bemutatja a szemantika működését, másrészt a PSSM szemantikát megvalósító eszközök konformancia teszteléséhez használható. A 2. ábra egy ilyen PSSM tesztesethez tartozó állapotgépet ábrázol, amivel bemutathatók az állapotgépek szemantikájának kihívásai. Az állapotgép tartalmaz *ortogonális régiókat* (az S1 állapoton belül a szaggatott nyíllal elválasztott részek, amik konkurens viselkedést modelleznek), úgynevezett *do aktivitásokat* (amik elindulásuk után az állapotgéptől függetlenül, akár hosszú ideig is futhatnak és megszakíthatók), valamint úgynevezett „completion" átmeneteket (például a T3.2, amihez nem tartozik trigger esemény, és a forrás állapotához tartozó viselkedések befejezése utána tüzelhet). A teszteset megad egy eseményszekven-

ciát, aminek hatását meg kell figyelni, és az egyes átmenetek tüzelése és viselkedések végrehajtása során naplózott *lefutásokat* (trace) kell összehasonlítani a specifikációban megadottakkal. Az ortogonális régiók és a do aktivitás okozta *nemdeterminisztikus választások* miatt azonban többféle érvényes lefutás is tartozik a megadott eseményszekvenciához[4]. Például az alábbi mindkettő érvényes lefutás:

```
T2(effect)::S1(entry)::T2.2(effect)::T3.1.2(effect)::S3.1(do)::T3.2(effect)
T2(effect)::S1(entry)::S3.1(do)::T3.1.2(effect)::T2.2(effect)::T3.2(effect)
```

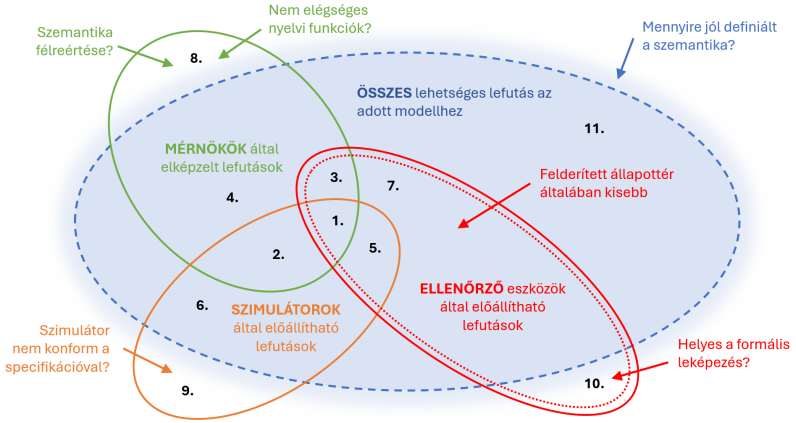


2. ábra. PSSM teszteset az állapotgépek szemantikájának illusztrálására [OMG19b]

A szemantika értelmezésénél az egyik fő kérdése az, hogy ennek a lehetséges lefutáshalmaznak *mekkora részhalmazát* társítja az adott modellhez az értelmezés során egy mérnök vagy egy eszköz. A specifikáció pusztán annyit vár el egy konform eszköztől, hogy az eszköz által előállított lefutások halmaza részhalmaza legyen a lehetséges lefutásoknak. Így tehát például hiába definiálja explicit módon a PSSM, hogy az ortogonális régiók bármilyen sorrendben végrehajthatók (sőt, akár ténylegesen párhuzamosan is futhatnak megfelelő hardver platform esetén), egy szimulátor vagy verifikációs eszköz konform lehet akkor is a specifikációval, ha mindig a bal oldali régiót hajtja először végre (és ezzel a lehetséges viselkedés egy jelentős részét sosem vizsgálja!). Az adott eszköz vagy mérnök szempontjából a szemantika ilyen értelmezése lehet teljesen racionális és indokolt döntés. Komoly problémát az okoz, ha az egyes szereplők értelmezése *inkonzisztens* egymással.

[4] A tesztesethez a specifikációban megadott lefutások egyébként hibásak, ezt a vizsgálatok során jelentettem is az OMG felületén (Issue PSSM11-9). Összesen 8 érvényes lefutása van a tesztesetnek.

Kutatásom során megmutattam, hogy még a formálisnak tűnő szemantika értelmezése sem egységes, és az egyes szereplők és eszközök által elképzelt lehetséges lefutások halmaza *csak részben* egyezik meg. A 3. ábra illusztrálja a lehetséges *értelmezésbeli eltéréseket* és ezek hatását, amelyek közül néhányat ki is emelek. A modelleket szerkesztő vagy azokat olvasó mérnökök sokszor nincsenek tisztában a formális szemantika minden részletével vagy olyan viselkedéseket is elképzelnek, amit az adott modell nem tud mutatni (8. metszet az ábrán). A szimulátor eszközök tipikusan egy-egy végrehajtást mutatnak meg, hiába van több érvényes lefutás egy konkurenciát tartalmazó modellben. Ezen lefutások egy részét egy formális verifikációs eszköz meg tudja találni (7. metszet), mivel az megpróbálja bejárni a modell teljes állapotterét. Azonban a skálázódási problémák miatt a verifikációs eszközök is tipikusan csak egy részhalmazát tudják felderíteni a modell lehetséges viselkedésének, emiatt további lehetséges lefutások rejtve maradhatnak (11. metszet). Minden ilyen felderítetlen vagy inkonzisztens módon azonosított lefutás egy-egy nem azonosított hiba vagy későbbi hibaok lehet (pl., ha az állapotgép kézi vagy generált implementációja során nem kezeli a kimaradt esetet).



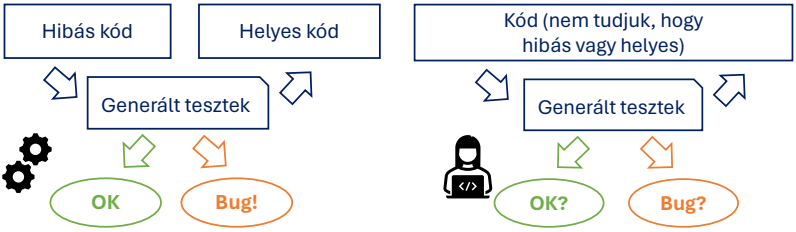
3. ábra. Azonosított értelmezésbeli eltérések a lehetséges lefutások halmazán [j1]

Cikkünkben [j1] a PSSM specifikáció és a kapcsolódó eszközök vizsgálatával támasztottuk alá ezeknek az inkonzisztens értelmezéseknek a meglétét. Többek között a szabvány szövegének átvizsgálásával és a formális szemantika analizálásával rámutattunk olyan esetekre, amikor maga a szabvány sem konzisztens (például, hogy mi számít atomi lépésnek). A PSSM referencia implementációjaként fejlesztett Eclipse Papyrus Moka [Gue+15] felhasználásával demonstráltuk, hogy létezik olyan szimulátor, ami nem képes előállítani az összes lehetséges lefutást. A PSSM tesztesetek átvizsgálása és futtatása során pedig azonosítottuk, hogy maguk a tesztesetek is néha ellentmondanak egymásnak, és a megadott elvárt lefutások halmaza is sokszor pontatlan vagy hiányos (a szabvány készítői ezeket kézzel, a saját szemantika értelmezésük alapján állították elő). Ezen hiányosságok kezelésére

javaslatokat tettünk, amik alkalmazásával a jövőbeli modellezési nyelvek specifikációja egyértelműbbé válhat.

Generált tesztek A kódalapú tesztgenerálás esetén egy már meglévő forráskód alapján generálunk automatikusan teszt(bemenet)eket. A generálás sokféle módszer alapján történhet [Ana+13], például irányított véletlen keresés vagy *szimbolikus végrehajtás* (symbolic execution) [Kin76] alapján. A generálás célja általában az, hogy a forráskód struktúrájának (utasítások, döntések) minél nagyobb részét lefedje, vagy olyan bemeneteket válasszon ki, amire futásidejű kivétellel vagy egyéb hibával válaszol a rendszer. A hibák egy része könnyen azonosítható (pl., ha összeomlik a program). Egyéb esetben a legtöbb eszköz pusztán csak rögzíti a tesztben, hogy mi volt a *megfigyelt kimenet* (más orákulum híján jobbat nem tud tenni, hiszen tipikusan nem érhető el az elvárt viselkedés részletes specifikációja).

A fundamentális probléma ezzel a megközelítéssel, hogy nem veszi figyelembe, hogy az így generált tesztek jelentését nem triviális értelmezni. Ha például a specifikáció szerint elvileg 200-as értéket kellene visszaadni a kiválasztott tesztbemenetekre, de az implementáció 300-as értéket ad vissza, akkor a generált tesztek ezt a hibás értéket fogják tartalmazni, és mindaddig sikeresen futnak, amíg ez a hibás érték a válasz. A hibát csak akkor tudjuk ténylegesen *észrevenni*, ha valaki részletesen *átvizsgálja* a generált teszteket, végiggondolja, hogy milyen választ kéne adni a bemenetre, majd összehasonlítja azt a generált tesztben rögzített kimenettel (a 4. ábra jobb oldala mutatja az *emberi kiértékelésen* alapuló esetet).

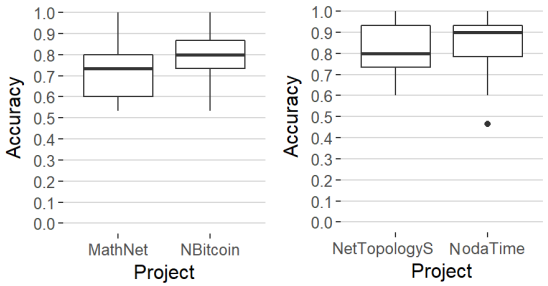


(a) Összehasonlító kiértékelés a cikkekben (b) Emberi kiértékelés a valóságban

4. ábra. Generált tesztek értelmezése és a hibák detektálása

Az új tesztgeneráló módszereket javasló vagy az eszközöket kiértékelő cikkek általában nem szembesülnek ezzel a problémával. Ugyanis a cikkekben legtöbbször nem emberi kiértékelést, hanem automatizált, a helyes és egy mesterségesen előállított hibás verzión való futtatást *összehasonlító kiértékelést* használnak. A teszteket a hibás verzióból generálják, majd a helyes változaton futtatják. Ha a kettő futtatás eredménye között eltérés van, akkor azt úgy értelmezik, hogy a generált teszt detektálni tudta az adott hibát (a 4. ábra bal oldala). Holott a valóságban a hibát csak akkor detektálnánk ténylegesen, ha a generált tesztet megvizsgáló mérnök észreveszi, hogy egy nem várt kivételt vagy váratlan értéket rögzít a generált teszt. A kapcsolódó kísérletek nagyon kis része alkalmaz a kiértékeléshez emberi résztvevőket (például Fraser és társai munkája [Fra+15]), de ezekben is a kiértékelések a hibás és helyes változaton való futtatás összehasonlítása vagy áttételes metrikák (kódfedettség, mutációs pont) alapján történtek.

Kutatásom során megmutattam, hogy az emberi résztvevők nem képesek tökéletesen értelmezni a generált tesztek jelentését, és ezért az általuk ténylegesen észrevett hibák száma kevesebb lehet, mint ahogyan azt az összehasonlításon alapuló módszerek és áttételes metrikák alapján számítanák. Ráadásul az emberi résztvevők többféle módon is hibáznak. Ha meg kell állapítaniuk, hogy egy forráskódból generált teszt megfelel-e a specifikációjának, akkor i) az elvárt viselkedést rögzítő teszteseteket is képesek helytelennek osztályozni, valamint ii) a helytelen viselkedést rögzítő tesztesetek elemzése során sem veszik észre a hibát.



5. ábra. Találati arány (accuracy) a generált tesztek értékelésénél

Cikkünkben [j2] bemutattunk egy olyan kísérletsorozatot, amiben 106 ember vett részt[5]. Az eredmények alapján a résztvevők találati aránya (helyes válaszok aránya az összes válaszhoz képest) 46–100% között mozgott (medián érték 80%, lásd 5. ábra). A cikk részletesen vizsgálja a bináris klasszifikáció további, bevett metrikáit is (TPR, TNR, MCC – Matthews korrelációs együttható), amelyek hasonló eredményeket mutattak. Tehát az emberi kiértékelés során a hibák azonosításában egy jelentős különbség tapasztalható az ideális esethez képest, ami befolyásolja a tesztgeneráló módszerek hatékonyságával kapcsolatos eddigi kiértékeléseket, és figyelembe kell venni ezen technikák alkalmazása során.

A résztvevők válaszainak elemzése során kitűnt, hogy különösen azoknál az eseteknél hibáznak az emberek, ahol nem várt bemenet szerepel vagy kivételt okoz vagy kellene okoznia az implementációnak. Megvizsgáltam az elterjedt tesztgeneráló eszközöket (Pex [TH08], Randoop [Pac+07], EvoSuite [FA13]) az ilyen speciális esetekre, és azt találtam, hogy nem egységes a működésük. Egyes eszközök például mindig rögzítik a kivételeket, de nem generálnak hibát, míg mások a kivétel típusától függően sikeres vagy sikertelen tesztkimenetet állítanak be. Sőt, a szakirodalom azt sem összesíti, hogy milyen eseteket kellene egyáltalán kezelni.

Ezért azonosítottam, hogy mik azok a faktorok, amik leginkább befolyásolják a generált tesztek osztályozását, és javasoltam egy *klasszifikációs keretrendszert* ezek kombinációjának kezelésére (1. táblázat). A keretrendszer megadja az elvárt felhasználói akciót és a teszt javasolt kimenetelét. A 8 kombinációból 2 nem lehetséges, 2 esetben helyesnek, míg 4 esetben hibásnak kellene osztályozni a tesztet.

[5]Egy szoftvereszközökkel kapcsolatos áttekintő cikk [KLB13] által vizsgált 345 darab kiértékelésben a résztvevők számának medián értéke 36 volt, így a szakterületen belül a 106 résztvevő soknak számít.

1. táblázat. Klasszifikációs keretrendszer a generált tesztek osztályozását befolyásoló faktorokkal

ID	*Faktorok*			Teszt rögzíti	Felhasználói akció	Klasszifikáció	Teszt kimenet
	Kivétel elvárt?	Hiba fellép?	Kivételt okoz?				
C1	F	F	F	elvárt viselkedés	jóváhagyni a tesztet	HELYES	sikeres
C2	F	F	T	–	–	–	–
C3	F	T	F	nem várt viselkedés	észrevenni, hogy hibás a kód	HIBÁS	sikeres
C4	F	T	T	nem várt viselkedés	felismerni, hogy a kivétel hibára utal	HIBÁS	sikeres vagy sikertelen
C5	T	F	F	–	–	–	–
C6	T	F	T	elvárt viselkedés	jóváhagyni a tesztet	HELYES	sikeres vagy sikertelen
C7	T	T	F	nem várt viselkedés	észrevenni, hogy kivétel hiányzik	HIBÁS	sikeres
C8	T	T	T	nem várt viselkedés	felismerni a helytelen kivételt	HIBÁS	sikeres vagy sikertelen

A nem egyértelmű esetek és az eszközökben talált hiányosságok kezelése hozzájárulhat ahhoz, hogy jobban érthető teszteket generáljanak az eszközök, ezáltal növelve a mérnökök által ténylegesen azonosított hibák számát.

Tézis A bemutatott eredmények tézisszerű összefoglalása az alábbi.

1. tézis. Elemzések és kísérletsorozat segítségével rámutattam, hogy a modellezési nyelvek és generált tesztek szemantikájának értelmezésbeli inkonzisztenciái negatívan befolyásolják az ezeket használó ellenőrzési módszerek által detektálható hibák számát.

1.1. Értelmezésbeli eltérések vannak a modellezők, valamint a szimulátor és verifikációs eszközök fejlesztői által vélt lehetséges lefutások halmazaiban viselkedésleíró modellezési nyelvek esetén. Az eltérések fajtáit a PSSM specifikáció vizsgálatán keresztül azonosítottam [j1].

1.2. A kódalapú tesztgeneráló eszközök által generált tesztek kiértékelése során az emberek kevesebb hibát azonosítanak, mint az a hibás és helyes verziókat összehasonlító kiértékelés alapján várható volna. Megterveztem egy, a generált tesztek emberi kiértékelésének teljesítményét mérő kísérlet koncepcióját, aminek a végrehajtása alátámasztotta az eltérést. Az eredmények alapján javasoltam egy klasszifikációs keretrendszert [j2].

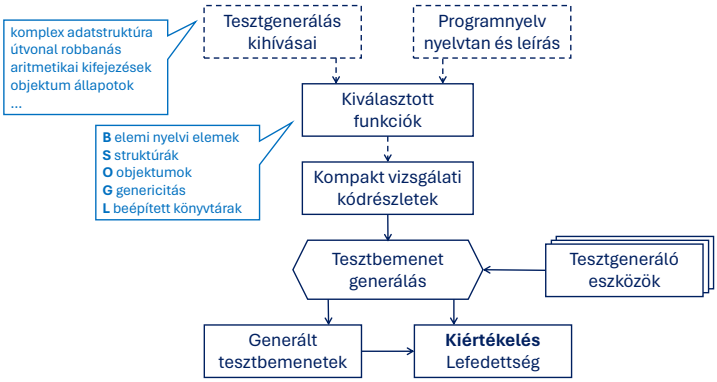
Az 1.1. tézisponthoz tartozó folyóiratcikk [j1] Elekes Márton doktoranduszommal és Molnár Vincével közös. Saját munkám az értelmezésbeli eltérések fajtáinak szisztematikus azonosítása. A PSSM specifikáció részletes elemzése Elekes Márton, a kapcsolódó szereplők és termékek elemzése Molnár Vince saját munkája.

Az 1.2. tézisponthoz tartozó folyóiratcikk [j2] Honfi Dávid doktoranduszommal közös. Saját hozzájárulásom a kísérlet koncepciója és a generált tesztek osztályozási problémáját leíró keretrendszer, ami a tesztek érthetőségét befolyásoló faktorok figyelembe vétele alapján segíthet jobban érthető teszteket generálni. A kísérlet részleteinek megtervezése, végrehajtása és az adatok elemzése Honfi Dávid PhD-disszertációjának része [Hon21].

2.2. Ellenőrzési eszközök kiértékelése

A célkitűzéseknél bemutatott tanulmányok és felmérések alapján az elérhető ellenőrzési eszközök hiányosságai és korlátai is fontos akadályozó tényezők a fejlett, automatikus ellenőrző eszközök elterjedésében. Kutatásaim során különböző formális verifikációs és tesztelési eszközöket értékeltem ki kísérletek segítségével, hogy azok erősségeit és gyengeségeit azonosítani tudjam.

Kódalapú tesztgeneráló eszközök Az irodalomban rengeteg forráskódból kiinduló tesztgeneráló módszert és eszközt[6] javasoltak [Ana+13]. Léteznek kísérletek [Sha+15] és versenyek [DPG20], amiknek során több eszközt is kiértékelnek ugyanazon a szoftvereken futtatva. Azonban ezek általában összesített metrikákat közölnek (pl. elért átlagos lefedettség), amik magas szintű következtetések levonására alkalmasak. Így *részletes visszajelzést* nehéz kiolvasni belőlük azzal kapcsolatban, hogy pontosan milyen nyelvi konstrukciót vagy programozási mintát nem képesek az egyes eszközök kezelni, és mivel lehetne azok képességeit javítani. Egy ilyen részletes tanulmány volt publikálva az irodalomban [GA14], azonban az ebben felhasznált tesztkódok és részletes eredmények nem voltak elérhetők.



6. ábra. Módszer tesztgeneráló eszközök nyelvi funkció alapú kiértékelésére [j3]

Kutatásomban javasoltam egy módszert (6. ábra), hogy hogyan lehet a kódalapú tesztgeneráló eszközök képességeit részletesen kiértékelni egy olyan kompakt kódrészletekből álló készlet segítségével, ami az imperatív programozási nyelvek tesztgenerálás szempontjából releváns fogalmait és elemeit lefedi. A lefedendő funkciók kiválasztása és a kódrészletek megkonstruálása a tesztgenerálási algoritmusok ismert kihívásai és az egyes nyelvi elemek gyakorisága alapján történt. Valamint törekedni kellett arra, hogy az egyes kódrészletek lehetőleg egy funkciót fedjenek le, hogy egy funkció támogatásának hiánya ne maszkolja el a többit. Az egyes tesztgeneráló eszközöket ezeken a kódrészleteken futtatva a generált tesz-

[6] A honlapomon elérhető egy lista a tesztgeneráló eszközökről:
`http://mit.bme.hu/~micskeiz/pages/code_based_test_generation.html`

tek tartalmát és a bevett tesztelési metrikákat elemezve következtettem arra, hogy az adott eszköz milyen mértékben képes kezelni a vizsgált elemet.

```
1  public static int startsEnds(String s) {
2    if (s == null) {
3      return -1;
4    } else if (s.startsWith("test")) {
5      if (s.endsWith("error")) {
6        return 1;
7      } else {
8        return 2;
9      }
10   } else {
11     return 0;
12   }
13 }
```

7. ábra. Példa tesztgenerálás képességeit vizsgáló kompakt kódrészletre

A 7. ábra mutat példát egy ilyen kódrészletre. A tesztgeneráló eszköznek olyan `s` szöveg bemeneteket kell generálnia, amivel az összes utasítást le tudja fedni a vizsgált kódrészletben (szándékosan külön `return` utasítás tartozik minden döntési ághoz az értékelés miatt). Az összes utasítás lefedéséhez a megadott kezdetű vagy végű szövegek generálása is szükséges. Ehhez vagy értelmeznie kell a program struktúráját és a megfelelő `String` metódusokat, vagy legalább felhasználnia a kódban található literálokat és azok különböző kombinációjával próbálkozni.

Cikkünkben [j3] 363 ilyen kódrészlet segítségével értékeltük ki 6 tesztgeneráló eszköz képességeit a legalapvetőbb vezérlési elemektől kezdve a tesztgenerálás szempontjából kihívást jelentő funkciókig (például többszálú vagy hálózati kommunikációt használó programok). A kiértékeléseket ismételten futtattunk, hogy a véletlen algoritmusok hatását kezeljük. Egy-egy kódrészleten futtatást a következő öt státusz egyikébe soroltuk: az eszköz nem tudta elindítani a tesztgenerálást (*N/A*), az eszköz kivételt dobott generálás közben (*EX*), az eszköz kifutott az idő- vagy memóriakorlátból (*T/M*), a generált tesztek nem fedték le az összes lehetséges utasítást (*NC*), a generált tesztek lefedték az összes utasítást (*C*).

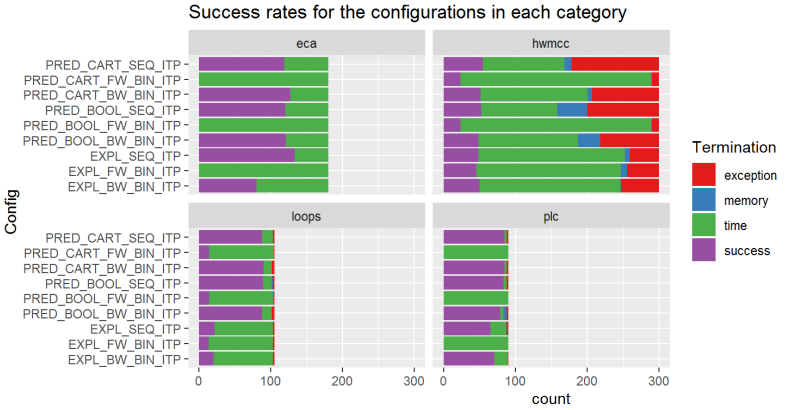
A 8. ábra bemutatja az eredmények egy részének összesítését. Az egyes sorok a tesztgeneráló eszközöket, az oszlopok a kódrészletek kategóriáját jelölik. Egy cellában pedig az adott kategóriába tartozó kódrészletekre kapott státuszok eloszlása látszik. A cikkben további metrikákat (lefedettség, generált tesztek száma és mérete, mutációs pontszám) és a generáláshoz szükséges időt is elemeztük. A részletes eredmények megtekintése nélkül is látszik, hogy jelentős különbségek vannak az egyes eszközök részletes képességei között. Továbbá bizonyos kategóriák, például a külső könyvtárak és függőségek kezelése, különös nehézséget okozott a legtöbb eszköznek. A kutatási eredmények hatása, hogy i) a részletes visszajelzések alapján célzott módon lehet javítani a tesztgeneráló eszközöket, ii) megkereshető, hogy adott funkciót melyik eszköz vagy eszközök kombinációja képes a legjobban lefedni, és iii) az összes eszköznek kihívást okozó funkciók kijelölik, hogy hol szükséges a tesztgenerálás alapjául szolgáló algoritmusok fejlesztése.

8. ábra. Tesztgeneráló eszközök kiértékelésének összesítése [j3]. (Státuszok: Not/Available, EXception, Timeout/Memory, Not Covered, Covered)

Modellellenőrzési algoritmusok Az első modellellenőrző algoritmus megjelenése óta számos változatot és kiegészítést javasoltak ehhez a formális verifikációs módszerhez [Cla+18]. Egy széles körben alkalmazott változat a *Counterexample-guided Abstraction Refinement* (CEGAR), ami a modell egy durva absztrakciójából kiindulva automatikus finomítási lépések során deríti fel az állapotteret. Egy CEGAR-alapú modellellenőrző eszköz általában többféle absztrakciós és finomítási módszert tartalmaz, amelyeknek különböző kombinációi más-más típusú modellen bizonyulnak hatékonynak. A módszerek és algoritmusok ilyen fajta kombinációjának lehetőségét az is indokolja, hogy általában egy modellellenőrző különböző bemeneti modelleket támogat. Például a kutatócsoportunkban fejlesztett Theta [Tót+17] eszköz modelltranszformációk segítségével képes állapotgépeket, hardvermodelleket vagy akár C kódot is ellenőrizni (ezek belül általában más-más formális reprezentációra képződnek le, de maguk az alap algoritmusok közösek az egyes formalizmusokra). Azonban, ha a Theta összes paraméterét figyelembe vesszük, akkor több ezer különböző konfigurációval lehetne vizsgálni az eszközt.

A módszerek és algoritmusok ilyen nagy számú változatát nem reális végigpróbálni egy-egy új modell esetén. Az irodalomban javasolt megoldás általában az, hogy a modell fajtájától és esetlegesen bizonyos jellemzőitől függően adott alapértelmezett értékeket használunk vagy a sikeres kombinációkból képzett portfóliót futtatunk [Ádá+22]. Az alapértelmezett értékek és a portfólió elkészítéséhez viszont részletes előzetes mérések szükségesek, amik a főbb módszerek teljesítményét megmérik nagyszámú ellenőrizendő modell esetén. Egy kísérletsorozatot javasoltam, amivel az egyes CEGAR algoritmusvariánsok teljesítménye kiértékelhető különböző fajtájú és jellegű modellek segítségével, és ami megfelel a terület

kísérleti módszertani ajánlásainak [Woh+12; Ral+21].



9. ábra. Részlet modellellenőrző algoritmusok kiértékeléséből [j4]

Cikkünkben [j4] a Theta keretrendszerben implementált algoritmusváltozatok hatékonyságát értékeltük ki. A vizsgálathoz az eredmények érvényességét növelendő több forrásból választottunk modelleket, amik az területen reprezentatívnak számítanak: 445 darab az International Competition on Software Verification (SV-COMP) verseny benchmark készletéből származó C-programot, 300 darab a Hardware Model Checking Competition (HVMCC) verseny benchmark készletéből származó hardvermodellt és 90 darab, a CERN-től származó PLC-modellt [Adi+15] használtunk fel. A kísérleti elrendezések megtervezésében a fő hozzáadott érték az volt, hogy a több ezer lehetséges konfigurációból azokat a paramétereket választottuk ki faktoroknak, amiknek a korábbi mérési eredmények és a szakirodalom alapján jelentős hatása lehet a verifikáció sikerességére. Hat kutatási elrendezést alakítottunk ki a CEGAR algoritmus különböző lépéseire koncentráló faktorokkal. Fontos volt továbbá, hogy az összes lehetséges absztrakt tartományt minden egyes kísérletnél vizsgáljuk (úgynevezett „blocking factor”), mivel ezeknek döntő szerepe van az algoritmus működésében. Végül több mint 60 konfigurációt vizsgáltunk meg, és az eredményeket összehasonlítottuk az SV-COMP verseny eszközeivel.

A 9. ábra az egyik méréssorozat futásaival illusztrálja az eredményeket. A sorok az egyes algoritmus konfigurációkat, az egyes blokkok a bemeneti modell típusokat jelölik. A lila szín jelzi a sikeres verifikációk számát, a többi szín a sikertelen lefutásokat jelzi. Ezek a mérések például azt mutatják, hogy a tradicionális bináris interpolációt (`*_FW_BIN_ITP`) használó konfigurációk jelentősen rosszabbul teljesítenek mint a másfajta finomítási stratégiát használók. Továbbá, míg az összetett vezérlési struktúrákat tartalmazó *eca* kódnevű C-programokra az explicit absztrakció (`EXPL_*`) volt a leghatékonyabb, addig a végrehajtási ciklusokban működő PLC-programok esetén a predikátum absztrakciót (`PRED_*`) használó kombinációk sikeresebbek voltak. A mérések hozzájárultak, hogy azonosítsuk az egyes algoritmusok erősségeit, a modellek fajtájától függően ajánljuk javasolt konfigurációkat

és új algoritmusokat dolgozzanak ki kutatócsoportunk munkatársai. A kidolgozott mérési eljárást azóta is használják a Theta eszközzel kapcsolatos kutatások. Kapcsolódó cikkünkben [HM17] az előzetes mérési eredményeken az egyes algoritmusparaméterek hatását vizsgáltuk többek között döntési fák segítségével.

Mutációs tesztelés A *mutációs tesztelés* (mutation testing) [JH11] segítségével meglévő tesztkészletek minőségét lehet kiértékelni, majd az eredmények alapján kiegészíteni azokat. A technika alapelve, hogy a tesztelendő programba apró módosításokat helyezünk el (úgynevezett mutánsokat hozunk létre), amik tipikus programozási hibákat imitálnak. Ezeket a módosításokat mutációs operátorok alkalmazásával hozzuk létre, ami lehet például egy feltételben a relációs jel felcserélése. Az így előálló mutánsok halmazára lefuttatjuk a meglévő tesztkészletünket. Ha sikertelen lesz legalább egy teszteset eredménye egy mutánson, akkor úgy vélhetjük, hogy ezt a hibát a tesztkészlet fel tudja deríteni („megöli" a mutánst). Az életben maradt mutánsokhoz tartozó hibákat viszont a tesztkészlet nem tudja felderíteni, így ennek az okát érdemes megvizsgálni és a tesztkészletet bővíteni.

A mutációs tesztelést ugyan több mint 40 évvel ezelőtt javasolták először, de csak az utóbbi időben kezdett elterjedni a használata (részben a rendelkezésre álló eszközök hiányosságai miatt). Meglehetősen kevés cikk közöl kiértékeléseket a mutációs tesztelés kritikus rendszerekben való használatáról. Megterveztem egy esettanulmányt, ami a mutációs tesztelés alkalmazhatóságát vizsgálja vasúti rendszerek szoftverkomponenseihez tartozó tesztkészletek kiértékelésében. Tudomásom szerint nincs korábbi publikált tanulmány, ami vasútipari szoftvereken vizsgálta a mutációs tesztelés alkalmazhatóságát.

Cikkünkben [j5] a Knorr-Bremse céggel együttműködve több kísérletet végeztünk el és elemeztük az eredményét.

- Az első kísérlet során egy kódalapú tesztgeneráló eszköz által utasítás és MC/DC lefedettség céljából generált tesztkészlet mutációs tesztelését végeztük el a szakterületen bevett referencia projekten. A forráskód alapján generált tesztkészlet átlagosan 74,22%-os mutációs pontszámot ért el (a megölt mutánsok aránya az összes mutánshoz). A mutánsok életben maradását a globális és statikus változók hiányos ellenőrzése okozta.
- A második kísérlet során egy 15 darab szoftverkomponenshez tartozó, 734 tesztesetből álló tesztkészletet értékeltünk ki egy saját mutációs tesztelő eszköz segítségével. A szoftverkomponensek C nyelven íródtak, összesen 7 251 sor kódot tartalmaztak, és SIL 2-es (Safety Integrity Level) besorolásúak. A meglévő tesztkészlet összesítve 80,79%-os mutációs pontszámot ért el. Az eredmények egyrészt rámutattak, hogy milyen tipikus hiányosságok voltak a tesztkészlet létrehozása során (például a bonyolult feltételek nem lettek mind lefedve az elágazásokban), valamint, hogy melyik mutációs operátor bizonyult kevésbé hatékonynak (a *Remove condition* operátor bár nagyszámú mutánst generál, de azok egy része hibás).

Az eredmények továbbá igazolták, hogy alkalmazható a mutációs tesztelés biztonságkritikus szoftverkomponensek tesztkészletének kiértékelése, és megerősítették, hogy az egyik fontos gyakorlati kihívás egy olyan mutációs eszköz kiválasztása vagy kifejlesztése, ami az adott környezetben használt C-fordító és build eszközkészlettel együtt tud működni.

Tézis A bemutatott eredmények tézisszerű összefoglalása az alábbi.

2. tézis. Kísérleteket terveztem, amelyek szisztematikusan kiértékeltek különböző tesztelési és verifikációs módszereket és szoftver eszközöket, valamint korlátokat azonosítottam az eszközök jelenlegi alkalmazhatóságával és skálázhatóságával kapcsolatban.

2.1. Javasoltam egy nyelvi funkciók lefedésén alapuló, kompakt vizsgálati kódrészleteket használó módszert és kísérletet forráskód-alapú tesztgeneráló eszközök összehasonlítására. Az adatok kiértékelésével azonosítottam a vizsgált eszközök tipikus hiányosságait, megerősítve a tesztgeneráló algoritmusok elméleti és gyakorlati korlátait [j3; c8].

2.2. Javasoltam egy kísérletsorozatot predikátum és explicit absztrakciót használó CEGAR modellellenőrzési algoritmusváltozatok kiértékelésére szoftver- és hardvermodelleken. Az eredmények azonosították, hogy melyik bemeneti modell fajtán melyik konfigurációk hatékonyak [j4].

2.3. Javasoltam egy kísérletsorozatot mutációs tesztelés használhatóságának vizsgálatára beágyazott biztonságkritikus szoftverek környezetében. Az eredmények alapján mind az MC/DC lefedettséget megcélzó tesztgenerátor, mind a szigorú előírásoknak megfelelő, tesztelők által előállított tesztkészletben is tud hiányosságot találni a mutációs tesztelés [j5].

A 2.1. tézisponthoz tartozó közlemények [j3; c8] Cseppentő Lajos hallgatómmal közös munka. Saját munkám a kiértékelő módszer elvének kidolgozása és az adatok elemzése. Az összehasonlító tesztkészlet megtervezése és a kiértékelő keretrendszer implementálása Cseppentő Lajos saját munkája [Cse16].

A 2.2. tézisponthoz tartozó folyóiratcikk [j4] Hajdu Ákos doktoranduszommal közös munka. Saját hozzájárulásom a kísérleti elrendezések megtervezése. A továbbfejlesztett algoritmusváltozatok kidolgozása és megvalósítása, valamint a mérések elemzése Hajdu Ákos PhD-disszertációjának része [Haj20].

A 2.3. tézisponthoz tartozó folyóiratcikk [j5] Serban Andrada Alexia hallgatómmal közös munka. Saját munkám a kísérlet koncepciója. A mutációs eszköz megtervezése, megvalósítása, a mérések elvégzése és az egyes mutánsok életben maradását okozó tényezők vizsgálata Serban Andrada Alexia saját munkája.

2.3. Új ellenőrzési módszerek és eszközök

Az előzőekben azonosított és az irodalom alapján ismert kihívások megoldására új, ellenőrzési feladatokat támogató módszereket és eszközöket dolgoztam ki. A módszerek egy része a modellalapú rendszertervezést, rendszertesztelést, míg egy másik csoportjuk a kódalapú tesztelést támogatja.

SysML modellek verifikációja A modellalapú rendszertervezés során egyre gyakoribb, hogy a modelleket nem pusztán kommunikációra és dokumentációra használják, hanem azokat olyan részletességgel készítik el, hogy *végrehajthatók* legyenek [CMS19]. A végrehajtható modellek ellenőrzését tipikusan emberi átvizsgálással vagy szimulációval végzik el. A szimulációs során vagy a fontosabbnak gondolt forgatókönyvek mentén vizsgálják a rendszer funkcionális követelményeknek való megfelelését, vagy nagyszámú véletlen futtatás alapján becslik a

rendszer bizonyos jellemzőit (például válaszidő vagy megbízhatóság). Ezen ellenőrzési módszerek mind fontos információt szolgáltatnak a modellezett rendszerről, azonban nehezen tudnak olyan hibákat felderíteni, amihez a bemenetek ritka kombinációja szükséges vagy amihez egy bonyolult eseményszekvencia vezet.

A modellellenőrző módszerek elméletben a teljes lehetséges állapotteret felderítik minden bemeneti kombináció és lehetséges lefutás megvizsgálásával. Azonban a gyakorlati alkalmazásuk során két fő kihívással kell megküzdeni.

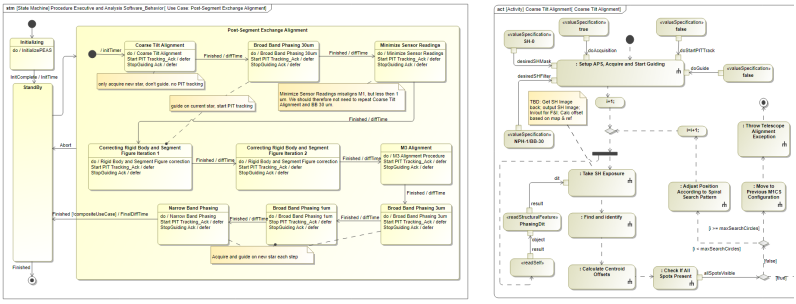
- *Szemantikai szakadék*: A modellellenőrző eszközök bemenete egy alacsony szintű matematikai formalizmus (például vezérlési folyam automata), míg a rendszermodell egy magas szintű, összetett elemeket tartalmazó modellezési nyelven van megfogalmazva. A kettő közötti leképezés magas szakértelmet igényel, és a modellezési nyelvek szemantikájának értelmezési nehézségei (lásd 2.1. fejezet) miatt nem egyértelmű feladat.

- *Skálázhatóság*: A modellellenőrző által bejárandó állapottér komplex rendszerek esetén hamar kezelhetetlen méretűvé válik. A szimbolikus vagy absztrakció alapú technikák segítenek az állapottér robbanás kezelésében, de további optimalizációk vagy heurisztikák alkalmazására van szükség.

Egy nemzetközi együttműködés keretében [j6] azt a célt tűztük ki, hogy az ipari gyakorlatban is alkalmazható verifikációs eszközkészletet készítünk SysML rendszermodellekhez. A legfontosabb követelmény az volt, hogy a módszernek támogatnia kell tudnia az *Executable Systems Engineering Method* (ESEM) és az OpenSE Coookbook[7] ajánlásai szerinti modellezési gyakorlatokat. Az ajánlásokat alkalmazó legnagyobb publikus modell a *Thirty Meter Telescope* (TMT[8]) rendszerterve. Verifikációs szempontból különös kihívást jelent, hogy a modell nagyméretű állapotgépeket tartalmaz, amiknél az egyes állapotokhoz vagy átmenetekhez tartozó viselkedések további összetett aktivitásdiagramokkal vannak megadva. Az állapotgépek reaktív, eseményalapú szemantikával, az aktivitásmodellek viszont adatfolyam-alapú szemantikával rendelkeznek, és a teljes modell verifikálásához a kettőt kombinálni kell. A 10. ábra mutat egy példát a modellek komplexitására.

A NASA Jet Propulsion Laboratory (JPL), az European Southern Observatory (ESO) és a TMT rendszermérnökeivel folytatott megbeszélések, a modellek tanulmányozása, valamint a felhasznált szimulációs eszközökön folytatott vizsgálatok alapján azonosítottam, hogy mik azok a modell elemek, amik egy olyan részhalmazát képezik a SysML-nyelvnek (úgynevezett *„pragmatic subset”*), amire később alkalmazható a formális verifikáció, valamint a mérnökök által gyakran alkalmazott minták megvalósíthatók velük. Külön figyelmet kellett fordítanom arra, hogy ezen elemek szemantikája összeállítható legyen úgy, hogy minél egyértelműbb legyen, és, hogy azt mind a szimulátor, mind a verifikációs eszköz, mind a mérnökök konzisztensen érthessék. Valamint a választott szemantikai variánsok megfeleljenek az egyes szereplők céljainak (pl., a különböző régiókban szereplő viselkedések egymásra hatásának lehetősége volt szemantikai szempontból kérdéses). Az ipari szereplők visszajelzése alapján a részhalmaz *elég nagy kifejező erejű*, hogy hasznos

[7] OpenSE Cookbook: `https://github.com/Open-MBEE/OpenSE-Cookbook`
[8] Thirty Meter Telescope SysML modell: `https://github.com/Open-MBEE/TMT-SysML-Model`



(a) Állapotgép (b) Aktivitás

10. ábra. A SysML-alapú rendszertervek nagyságrendjének szemléltetése: a bal oldali állapotgép mind a 10 állapotához tartozik a jobb oldali aktivitás diagramhoz hasonló, akár 20–30 elemi akcióval definiált részletes viselkedés.

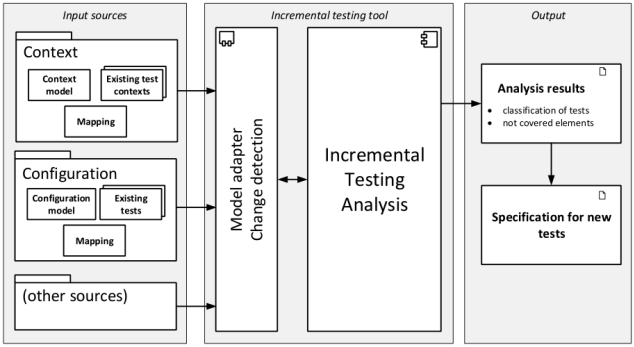
modelleket lehessen vele készíteni. A meglévő modellek közül az egyik legbonyolultabb átalakítható volt, hogy a kiválasztott részhalmaz elemeit használja.

Cikkünkben [j6] bemutattunk egy olyan eszközkészletet, ami a SysML nyelv kiválasztott részhalmazát képes automatikusan leképezni modellellenőrző eszközökre, elvégezni a formális verifikációt, majd az eredményt visszavetíteni a SysML modell szintjére. A leképezés első lépése a Gamma [Gra+20] keretrendszer reaktív komponensnyelvére történt, majd a Gamma eszköz transzformálja tovább azt különböző modellellenőrzőkre (például UPPAAL vagy Theta). A számításigényes formális verifikáció egy konténerizált környezetben történik, így az egyes modellekhez a mérnökök külön verifikációs végrehajtási környezetet indíthatnak. A teljes munkafolyamat be lett csomagolva az IncQuery Suite termékcsalád részét képező Dynamic Verification Toolkit nevű megoldásba.

A verifikáció sikeresen lefutott a vizsgált rendszermodell legnagyobb állapotgépére. Az állapotgép az iparban szokásosan használt állapotgépeknél nagyobb méretű: 61 állapotot, 93 átmenetet és 2310 aktivitásbeli akciót tartalmazott. Az eredmények alapján az automatikus verifikáció skálázódik ekkora ipari modellekre is, ha a bemeneti modell a kiválasztott nyelvi részhalmazra korlátozódik, és a szemantikai variánsokból a verifikációnak is kedvezőt állítunk be (például az ortogonális régiókban szereplő viselkedéseknek függetlennek kell egymástól lennie).

Modellalapú regressziós tesztelés Regressziós tesztelésnek [YH12] egy változtatás utáni újratesztelést hívunk, aminek a célja, hogy megbizonyosodjunk arról, hogy a változtatás nem hozott be új hibákat az eddig működő részekbe. Nagyméretű tesztkészlet esetén az újratesztelés idő- és erőforrásigényének csökkentése érdekében szelektív újratesztelést lehet alkalmazni, amikor megkeressük a változtatás által érintett teszteket és a futtatást leszűkítjük ezekre. Ez a módszer a tesztelés összes szintjén alkalmazható, az egységtesztektől egészen a rendszertesztekig.

Forráskód regressziós tesztelése esetén kiforrott algoritmusok és eszközök érhetők el. Azonban modellezési nyelvek, főleg *szakterület-specifikus nyelvek* esetén kevés megoldás áll rendelkezésre. Különösen érdekes a regressziós tesztkiválasz-



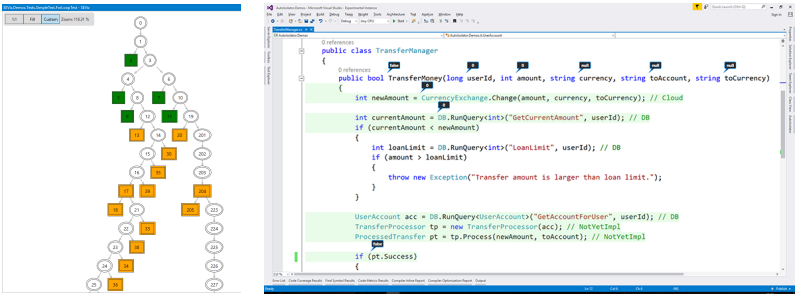
11. ábra. Nyelvfüggetlen modellalapú regressziós tesztelési megközelítés

tás kérdése, ha a vizsgált rendszert többféle modellezési nyelven készített modellek írják le. Az R5-COP nemzetközi K+F projekt keretében autonóm felderítő robotok tesztelését vizsgáltuk. Szakterület-specifikus nyelveket alkottunk egyrészt a robotok képességeinek és komponenseinek leírására, másrészt a robotokat vizsgáló tesztpályák és tesztek leírására. A feladat az volt, hogy egy komponens változtatása esetén meg kellett mondani, hogy melyik szabványos tesztelrendezés melyik tesztforgatókönyvét kell végrehajtani, hogy a változtatás hatását teszteljük.

Kutatásomban javasoltam egy olyan *általános*, a *modellezési nyelvtől független* megközelítést, amivel különböző modellezési nyelvekhez tartozó regressziós tesztelési feladatokat lehet kezelni (11. ábra). A módszer lényege, hogy a tesztelendő modellelemek és a tesztek között egy leképezést definiálunk, majd a modellverziók közötti automatikus változásdetektálás segítségével azonosítjuk, hogy melyik tesztelendő elemek változtak meg. Ezután a regressziós tesztkiválasztáshoz javasolt ismert algoritmusok segítségével osztályozzuk a meglévő teszteseteket (újratesztelendő, újrafelhasználható, felesleges stb.). Így azonosítható, hogy ha egy tesztet nem kell újrafuttatni vagy bizonyos tesztelrendezés hiányzik.

Cikkünkben [c9] a módszert implementáltuk egy, az Eclipse modellezési keretrendszerére épülő eszközben. Az eszköz képes a modellek állapotáról ellenőrző pontokat készíteni, az azok közötti különbségeket kiszámolni, majd a regressziós tesztkiválasztást egy egyszerű mohó, halmazlefedésre visszavezető algoritmussal végzi el. Az eszközt alkalmaztuk az autonóm felderítő robotok regressziós tesztelésére is, ahol a megoldás jól skálázódott a kívánt problémaméretre. A tesztkiválasztáshoz szükséges idő növekedése a változtatott elemek számával lineárisan volt arányos. Az esettanulmány fő tanulsága, hogy a tesztelés absztrakciós szintjét jól eltaláló modellezési nyelv megalkotása több iterációt igényelt, ezt az ipari partnerrel való többszöri egyeztetés után sikerült megtalálni. Például kezdetben a tesztpályák részelemeit modelleztük, de ez túl részletes modellt eredményezett; végül csak a tesztforgatókönyveket kellett a robot képességeihez rendelni.

Kódalapú tesztgenerálás támogatása A kódalapú tesztgeneráló módszerek közül a *szimbolikus végrehajtással* foglalkoztam részletesebben. A módszer lényege, hogy a vizsgált programkód bemeneteit szimbolikus változókként kezelve megpróbáljuk felderíteni a program lehetséges útvonalait. Az utasításokat értelmezve rögzítjük azok hatását a szimbolikus változókra, majd egy-egy elágazás esetén egy úgynevezett útvonalkényszerbe gyűjtjük össze, hogy milyen feltételek teljesülése esetén tudjuk az adott döntési ágat választani. A végrehajtás végére érve a kapott kényszereket egy kényszermegoldóval megoldva olyan tesztbemeneteket kapunk, ami pont az adott útvonalat járja be. A végrehajtást többször megismételve olyan tesztkészletet tudunk generálni, ami a kód struktúrájának minél nagyobb részét járja be. Az alap technikához sok továbbfejlesztést javasoltak, de még így is sok megoldatlan kihívás maradt (lásd a 2.2. fejezet kiértékelését).



(a) Interpretáció (SEViz) (b) Függőségek automatikus izolálása (AutoIsolator)

12. ábra. Új technikák és eszközök szimbolikus végrehajtás támogatására

Kutatásaim során két koncepciót javasoltam, hogy hogyan lehetne a szimbolikus végrehajtást használó tesztgeneráló eszközök hatékonyságán javítani.

- *Interpretáció*: Egyrészt a szimbolikus végrehajtást használó eszközök a gyakorlatban összetett algoritmusok kombinációját használják. Így egy-egy program esetén sokszor nehéz megérteni, hogy miért nem tudott egy adott részt lefedni megfelelő bemenetek generálásával. A szimbolikus végrehajtás lefutásának vizualizálása segít interpretálni, hogy pontosan milyen sorrendben próbálta az eszköz felderíteni és értelmezni az adott kódot. A kapott szimbolikus végrehajtási fa tanulmányozása segít a tesztelőnek az eszköz más paraméterekkel való újrafuttatásában és a lefedettség növelésében.

- *Függőségek izolálása*: Másrészt a tesztgenerálás során a külső függőségek sokszor jelentenek kihívást az eszközöknek. Ezeket úgynevezett teszt dublőrök (például mock vagy stub) segítségével le lehetne választani az egységtesztek írása során, azonban ez meglévő, régi kódbázis esetén sokszor nehezen megvalósítható. A függőségek automatikus leválasztása és egy parametrizálható, a tesztgenerátorral együttműködni képes befoglaló környezet létrehozása segít az izolációs kihívás megoldásában.

A végrehajtás interpretálását segítő vizualizáció megvalósítására cikkünkben [c10] bemutattuk a SEViz[9] eszközt, ami képes a Visual Studio fejlesztőkörnyezetbe épülő Pex/IntelliTest eszköz tesztgenerálási folyamatát nyomon követni, és a tesztgenerálással kapcsolatos metrikák és információk megjelenítésével segíteni a tesztelőt a szűk keresztmetszetek és problémák azonosításában. A módszer működését mesterséges és egy nyílt forráskódú projekt programkódján is demonstráltuk.

Az automatikus izoláció megvalósítására cikkünkben [j7] részletes algoritmusokat javasoltunk, és bemutattuk az AutoIsolator[10] eszközt. Az AutoIsolator szintén a Pex/IntelliTest eszközt egészíti ki, és az absztrakt szintaxis fán (AST) definiált *forráskód-transzformációk* segítségével képes a külső függőségeket automatikusan leválasztani, majd azok helyét elérhetővé tenni a tesztgenerátor számára, hogy vezérelni tudja a vizsgált egységet. A módszert 10 nyílt forráskódú projekt több mint 38 000 sornyi forráskódján értékeltük ki.

A 12. ábra mutat egy-egy példát a használatról: a SEViz egy ciklus szimbolikus végrehajtásán ábrázolja, hogy hol választott ki teszteseteket az eszköz; míg az AutoIsolator azt mutatja, hogy a zölddel jelölt sorok lefedéséhez milyen értékeket kellene a paraméterekhez és a függőségekhez a tesztgenerátornak hozzárendelnie.

Tézis A bemutatott eredmények tézisszerű összefoglalása az alábbi.

3. tézis. Olyan új eszközöket és módszereket dolgoztam ki, amik segítenek a mérnöki gyakorlatban előkerülő problémák leküzdésével hatékonyabbá tenni a rendszermodellek ellenőrzését és a tesztgenerálást.

- 3.1. Kiválasztottam egy olyan konzisztens részhalmazát a SysML rendszermodellezési nyelv elemkészletének és a szemantikai variánsainak („pragmatic subset"), ami lehetővé teszi a részhalmaznak megfelelő, ipari méretű modellek ellenőrzését. A részhalmazt a kapcsolódó cikk definiálja [j6].
- 3.2. Módszert dolgoztam ki a regressziós tesztelés modellalapú támogatására, ami a bemeneti modellezési nyelvek elemeinek egy általános regressziós tesztkiválasztási metamodellre való leképezésén alapszik. A módszer előnye, hogy független a bemeneti modellezési nyelvektől [c9].
- 3.3. Koncepciót javasoltam szimbolikus végrehajtás alapú tesztgenerálás támogatására i) a generálás tesztelő általi interpretációját segítő vizualizáció és ii) a függőségeket automatikusan leválasztó forráskód-transzformációk alkalmazásával [j7; c10].

A 3.1. tézisponthoz tartozó folyóiratcikk [j6] egy nemzetközi kutatás-fejlesztési együttműködés eredménye, a cikk társszerzői az IncQuery Labs, a NASA JPL, a TMT és a Johannes Kepler University (JKU) kutatói. Az együttműködés kutatási jellegű feladatait én vezettem. Saját munkám az úgynevezett „pragmatic subset" nyelvi részhalmaz kiválasztása. A keretrendszer, a transzformációk és a validációs szabályok megvalósítása főként Horváth Benedek munkája.

A 3.2. tézisponthoz tartozó konferenciaközlemény [c9] kollégáimmal közös. Saját hozzájárulásom az általános módszer kidolgozása. A módszer alkalmazhatóságát és skálázódását egy autonóm robotok tesztelését modellező esettanulmány

[9] SEViz eszköz: `https://ftsrg.mit.bme.hu/seviz/`
[10] AutoIsolator eszköz: `https://ftsrg.mit.bme.hu/autoisolator/`

támasztotta alá. Molnár Gábor hallgatóm implementálta a regressziós keretrendszert [Mol15], Honfi Dávid doktoranduszom valósította meg az esettanulmányt.

A 3.3. tézisponthoz tartozó közlemények [j7; c10] Honfi Dávid doktoranduszommal közös munka. Saját hozzájárulásom a módszerek koncepciójának kidolgozása. Az algoritmusok részletes kidolgozása, az eszközök megvalósítása és a kísérletek futtatása Honfi Dávid PhD-disszertációjának része [Hon21]. A koncepció hatékonyságát nyílt forráskódú projekteken végzett kísérletek igazolták: az eszközök használata megnövelte a korszerű tesztgenerátor által elért lefedettséget.

3. Az eredmények hasznosulása

A tézisfüzetben bemutatott eredmények nemzetközi K+F projektek vagy ipari együttműködések keretében születtek az elmúlt több mint 15 évben.

Nemzetközi K+F projektek Három EU-s projektben vezetőként, valamint további öt projektben közreműködőként vettem részt. A fenti eredmények az alábbi projektekhez kapcsolódtak.

- *Modellezési nyelvek*: A modellezési nyelvek ellenőrzésével kapcsolatos munkát [j1] részben az EMBRaCE [11] és ADVANCE [12] projektekben végeztük. Az EMBRaCE projekt keretében egy követelménymodellezési nyelv és a SysML nyelv integrációját vizsgáltuk. A tapasztalatokat visszacsatoltuk a készülő SysMLv2 nyelvbe (a nyelv szabványosításában Dr. Molnár Vince kollégám vesz részt). Az ADVANCE projekt keretében egy nemzetközi kutatócsapat tagjaként gyűjtöttünk össze és dolgoztunk ki módszereket kiberfizikai rendszerek ellenőrzésére. A tézisfüzetben bemutatott eredményeket felhasználtuk az olasz partnerek által kidolgozott modellező és biztonsági analízis eszköz [BSB22] vizsgálatakor. A projekteknek a BME-s témavezetője voltam.
- *Verifikációs eszközök*: a bemutatott verifikációs algoritmusok [j4] és a Theta modellellenőrző eszköz továbbfejlesztését az Arrowhead Tools[13] projekt keretében végezte vezetésemmel több kollégám [Ádá+22]. A fejlesztések eredményeként a Theta eszköz el tudott indulni az International Competition on Software Verification (SV-COMP) [Bey22] versenyen a terület vezető modellellenőrző eszközei mellett. Jómagam a verifikációs feladatokat és a Kritikus Rendszerek Kutatócsoport résztvevőinek munkáját koordináltam, a teljes projekt BME-s témavezetője Dr. Varga Pál volt.
- *Regressziós tesztelés*: a bemutatott modellalapú regressziós tesztelési megoldást [c9] az R5-COP[14] projektben dolgoztuk ki. A feladatot és az esettanulmányt a PIAP lengyel intézet biztosította, akik mentő és felderítő robotokat gyártanak. A projektben a kapcsolódó task vezetője voltam, a BME-s témavezető Dr. Majzik István volt.

Ipari együttműködések Az eredmények egy része közvetlen ipari együttműködések keretében jött létre, vagy későbbi együttműködésekben felhasználtuk azokat ipari rendszerek ellenőrzésekor.

- *thyssenkrupp*: A thysenkrupp budapesti fejlesztőközpontjával sokrétű együttműködést folytattunk. A tesztgeneráló eszközök kiértékelése során szerzett tapasztalatok alapján vizsgáltuk, hogy hogyan lehetne teszteket generálni az autóipari komponensek C nyelvű forráskódjából. A szekvencia diagramok szemantikájának vizsgálata alapján egy hallgatóm elkészített egy olyan eszközt, amivel a rendszertervek szekvenciái vethetők össze az integrációs

[11] Environment for model-based rigorous adaptive co-design and operation of CPS (ITEA 18039)
[12] Addressing Verification and Validation Challenges in Future CPS (H2020 823788)
[13] Arrowhead Tools for Engineering of Digitalisation Solutions (ECSEL 826452)
[14] Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems (ARTEMIS 621447)

tesztek során megfigyelt viselkedéssel. A modellezési módszerekkel kapcsolatos tapasztalataimat pedig a Kompetencia Központ[15] közös pályázat során használtuk fel a thyssenkrupp rendszertervezési módszertanának továbbfejlesztése során. A három éves projektnek Dr. Dabóczi Tamás mellett társvezetője voltam.

- *IncQuery Group és NASA JPL*: a SysML állapotgépek ellenőrzésével kapcsolatos megoldást [j6] egy nemzetközi együttműködés[16] keretében dolgoztuk ki. Az eszközkészlet Dynamic Verification Toolkit (DVT) néven az IncQuery Suite termékcsalád része. A Thirty Meter Telescope rendszermodelljét használtuk az eszközkészlet alkalmazhatóságának vizsgálatára. A NASA JPL, TMT és ESO mérnökeivel folytatott megbeszélések szolgáltattak alapot a modellezési nyelvek szemantikájának ellenőrzésével kapcsolatos munkához.
- *Knorr-Bremse*: A mutációs tesztelés beágyazott környezetben való alkalmazhatóságának vizsgálata [j5] a Knorr-Bremse környezetében és szoftvermoduljain történt.

Eredmények hatása Az eredményeimet más kutatók is felhasználják munkájuk során, hasznosulásuk például a cikkeim idézésében követhetők nyomon.

- *Tesztgeneráló eszközök kiértékelése*: A tesztgeneráló eszközök kiértékelésével kapcsolatos eredményeimet [j3] felhasználták a vizsgált eszközök. A cikket hivatkozni szokták a tesztgenerálás egyes azonosított nehézségeit igazolandó, valamint hivatkozza egy friss, tesztelési technikák kiértékelését áttekintő cikk is [MA21].
- *Generált tesztek osztályozása*: A generált teszteket vizsgáló kísérlet [j2] adathalmazára építve dolgoztak ki Setiani és társai [SFH20] egy teszteset érthetőségi modellt. Továbbá a kísérlet szerepel egy friss, a tesztkódok olvashatóságával kapcsolatos kutatásokat áttekintő cikkben [WUR22].
- A tézisfüzetben nem ismertetett munkámban [Mic+12] autonóm rendszerek teszteléséhez forgatókönyv-alapú rendszertesztelési módszert javasoltunk kollégáimmal együtt. Ezt több áttekintő cikk is ismerteti [Siq+21; TA20], valamint a kutatócsoportunkban ebből egy új kutatási irány indult [Maj+19].
- Egy korábbi, a PhD-disszertációm részét képező cikkünk [MW11] megjelenése után 10 évvel a hatása alapján elnyerte a *Software and Systems Modeling* folyóirat „Most Influential Regular Paper” díját.

Oktatás A tapasztalatokat folyamatosan beépítettem az általam oktatott tantárgyakba. Például a tézisfüzetben bemutatott módszerek és kiértékelések megjelentek a *Szoftver- és rendszerellenőrzés* (VIMIMA01) tantárgyban, amelyben a tesztgenerálást vizualizáló eszközt [c10] is használtuk egy labormérés keretében.

[15] Biztonságtudományi és Technológiai Kompetencia Központ (NKFIH 2019.1.3.1-KK-2019-00004)
[16] Accelerated Simulation for Industry-Scale Behavioral Modeling (incquery.io)

Publikációk

Tézisekhez kapcsolódó 10 válogatott publikáció

Folyóiratcikkek

[j1] M. Elekes, V. Molnár, and **Z. Micskei**. "Assessing the specification of modelling language semantics: a study on UML PSSM". in: *Software Quality Journal* 31 (2 2023), pp. 575–617. DOI: 10.1007/s11219-023-09617-5

[j2] D. Honfi and **Z. Micskei**. "Classifying generated white-box tests: an exploratory study". In: *Software Quality Journal* 27 (3 2019), pp. 1339–1380. DOI: 10.1007/s11219-019-09446-5

[j3] L. Cseppentő and **Z. Micskei**. "Evaluating code-based test input generator tools". In: *Software Testing, Verification and Reliability* 27 (6 2017), pp. 1–24. DOI: 10.1002/stvr.1627

[j4] Á. Hajdu and **Z. Micskei**. "Efficient Strategies for CEGAR-Based Model Checking". In: *Journal of Automated Reasoning* 64 (2020), pp. 1051–1091. DOI: 10.1007/s10817-019-09535-x

[j5] A. A. Serban and **Z. Micskei**. "Application of Mutation testing in Safety-critical Embedded Systems: A Case Study". In: *Acta Polytechnica Hungarica* 21 (8 2024), pp. 87–106. DOI: 10.12700/APH.21.8.2024.8.5

[j6] B. Horváth, V. Molnár, B. Graics, Á. Hajdu, I. Ráth, Á. Horváth, R. Karban, G. Trancho, **Z. Micskei**. "Pragmatic Verification and Validation of Industrial Executable SysML Models". In: *Systems Engineering* 26.6 (2023), pp. 693–714. DOI: 10.1002/sys.21679

[j7] D. Honfi and **Z. Micskei**. "Automated Isolation for White-box Test Generation". In: *Information and Software Technology* 125 (2020), pp. 1–16. DOI: 10.1016/j.infsof.2020.106319

Nemzetközi konferenciaközlemények

[c8] L. Cseppentő and **Z. Micskei**. "Evaluating Symbolic Execution-based Test Tools". In: *IEEE Int. Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10. DOI: 10.1109/ICST.2015.7102587

[c9] D. Honfi, G. Molnár, **Z. Micskei**, I. Majzik. "Model-Based Regression Testing of Autonomous Robots". In: *SDL 2017: Model-Driven Engineering for Future Internet"*. Springer, 2017, pp. 119–135. DOI: 10.1007/978-3-319-68015-6_8

[c10] D. Honfi, A. Vörös, and **Z. Micskei**. "SEViz: A Tool for Visualizing Symbolic Execution". In: *IEEE Int. Conf. on Software Testing, Verification and Validation (ICST)*. Tool track. IEEE, 2015, pp. 1–10. DOI: 10.1109/ICST.2015.7102631

	[j1]	[j2]	[j3]	[j4]	[j5]	[j6]	[j7]	[c8]	[c9]	[c10]
1. tézis	•	•								
2. tézis			•	•	•			•		
3. tézis						•	•		•	•

Hivatkozások

[Ádá+22] Z. Ádám et al. “Theta: portfolio of CEGAR-based analyses with dynamic algorithm selection (Competition Contribution)”. In: *TACAS*. Springer, 2022, pp. 474–478. DOI: 10.1007/978-3-030-99527-0_34.

[Adi+15] B. F. Adiego et al. “Applying Model Checking to Industrial-Sized PLC Programs”. In: *IEEE Trans. Ind. Informatics* 11.6 (2015), pp. 1400–1410. DOI: 10.1109/TII.2015.2489184.

[AGD18] D. Akdur, V. Garousi, and O. Demirörs. “A survey on modeling and model-driven engineering practices in the embedded software industry”. In: *J. Syst. Archit.* 91 (2018), pp. 62–82. DOI: 10.1016/j.sysarc.2018.09.007.

[Ana+13] S. Anand et al. “An orchestrated survey of methodologies for automated software test case generation”. In: *J. Syst. Software* 86.8 (2013), pp. 1978–2001. DOI: 10.1016/j.jss.2013.02.061.

[And+23] É. André et al. “Formalizing UML State Machines for Automated Verification – A Survey”. In: *ACM Comput. Surv.* (2023). DOI: 10.1145/3579821.

[Bar+15] E. T. Barr et al. “The Oracle Problem in Software Testing: A Survey”. In: *IEEE Trans. Software Eng.* 41.5 (2015), pp. 507–525. DOI: 10.1109/TSE.2014.2372785.

[BCW12] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012. ISBN: 9781608458820.

[Bey22] D. Beyer. “Progress on Software Verification: SV-COMP 2022”. In: *TACAS*. Springer, 2022, pp. 375–402. DOI: 10.1007/978-3-030-99527-0_20.

[BKP20] D. Bork, D. Karagiannis, and B. Pittl. “A survey of modeling language specification techniques”. In: *Inf. Syst.* 87 (2020). DOI: 10.1016/j.is.2019.101425.

[BSB22] I. Bicchierai, E. Schiavone, and F. Brancati. “Modelling and Assessing the Risk of Cascading Effects with ResilBlockly”. In: *IEEE Int. Conf. on Cyber Security and Resilience CSR*. IEEE, 2022, pp. 261–266. DOI: 10.1109/CSR54599.2022.9850342.

[Buc+20] A. Bucchiarone et al. “Grand challenges in model-driven engineering: an analysis of the state of the research”. In: *Softw. Syst. Model.* 19.1 (2020), pp. 5–13. DOI: 10.1007/s10270-019-00773-6.

[Cla+18] E. M. Clarke et al. *Handbook of model checking*. Springer, 2018. DOI: 10.1007/978-3-319-10575-8.

[CMS19] F. Ciccozzi, I. Malavolta, and B. Selic. “Execution of UML models: a systematic review of research and practice”. In: *Softw. Syst. Model.* 18.3 (2019), pp. 2313–2360. DOI: 10.1007/s10270-018-0675-4.

[Cse16] L. Cseppentő. “Evaluating Code-based Test Generator Tools”. MSc thesis. Budapest University of Technology and Economics, 2016.

[DPG20] X. Devroey, S. Panichella, and A. Gambi. “Java Unit Testing Tool Competition: Eighth Round”. In: *ICSE Workshops*. 2020. DOI: 10.1145/3387940.3392265.

[FA13] G. Fraser and A. Arcuri. “Whole Test Suite Generation”. In: *IEEE T. Software Eng.* 39.2 (2013), pp. 276–291. DOI: 10.1109/TSE.2012.14.

[Fra+15] G. Fraser et al. “Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study”. In: *ACM Trans. Softw. Eng. Methodol.* 24.4 (2015), 23:1–23:49. DOI: 10.1145/2699688.

[GA14] S. J. Galler and B. K. Aichernig. "Survey on test data generation tools". In: *STTT* 16.6 (2014), pp. 727–751. DOI: 10.1007/s10009-013-0272-3.

[GPW23] M. Gleirscher, J. van de Pol, and J. Woodcock. "A manifesto for applicable formal methods". In: *Softw. Syst. Model.* 22.6 (2023), pp. 1737–1749. DOI: 10.1007/S10270-023-01124-2.

[Gra+20] B. Graics et al. "Mixed-semantics composition of statecharts for the component-based design of reactive systems". In: *Softw. Syst. Model.* 19.6 (2020), pp. 1483–1517. DOI: 10.1007/s10270-020-00806-5.

[Gri+11] W. Grieskamp et al. "Model-based quality assurance of protocol documentation: tools and methodology". In: *Softw. Test. Verification Reliab.* 21.1 (2011), pp. 55–71. DOI: 10.1002/STVR.427.

[Gue+15] S. Guermazi et al. "Executable Modeling with fUML and Alf in Papyrus: Tooling and experiments". In: *EXE*. Vol. 1560. CEUR-WS, 2015, pp. 3–8.

[Haj20] Á. Hajdu. "Effective Domain-Specific Formal Verification Techniques". PhD dissertation. BME, 2020. DOI: 10.5281/zenodo.3892346.

[HM17] Á. Hajdu and Z. Micskei. "Exploratory Analysis of the Performance of a Configurable CEGAR Framework". In: *Proceedings of the 24th PhD Mini-Symposium*. 2017, pp. 34–37. DOI: 10.5281/zenodo.291895.

[Hon21] D. Honfi. "Evaluating and improving white-box test generation". PhD dissertation. BME, 2021. URL: http://hdl.handle.net/10890/15132.

[HR04] D. Harel and B. Rumpe. "Meaningful Modeling: What's the Semantics of "Semantics"?" In: *Computer* 37.10 (2004), pp. 64–72. DOI: 10.1109/MC.2004.172.

[IEE10] Institute of Electrical and Electronics Engineers. *Systems and software engineering – Vocabulary*. Std 24765:2010. 2010. DOI: 10.1109/IEEESTD.2010.5733835.

[JH11] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Trans. Software Eng.* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62.

[Kin76] J. C. King. "Symbolic execution and program testing". In: *Commun. of the ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[KLB13] A. J. Ko, T. D. LaToza, and M. M. Burnett. "A practical guide to controlled experiments of software engineering tools with human participants". In: *Empir. Softw. Eng.* 20.1 (2013), pp. 110–141. DOI: 10.1007/s10664-013-9279-3.

[Kle+14] G. Klein et al. "Comprehensive formal verification of an OS microkernel". In: *ACM Trans. Comput. Syst.* 32.1 (2014). DOI: 10.1145/2560537.

[LMM99] D. Latella, I. Majzik, and M. Massink. "Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker". In: *Form. Asp. Comput.* 11.6 (1999), pp. 637–664. DOI: 10.1007/s001659970003.

[MA21] M. Mayeda and A. Andrews. "Evaluating software testing techniques: A systematic mapping study". In: *Advances in Computers*. Elsevier, 2021, pp. 41–114. DOI: 10.1016/bs.adcom.2021.01.002.

[Maj+19] I. Majzik et al. "Towards System-Level Testing with Coverage Guarantees for Autonomous Vehicles". In: *MODELS*. 2019. DOI: 10.1109/models.2019.00-12.

[Mic+12] Z. Micskei et al. "A Concept for Testing Robustness and Safety of the Context-Aware Behaviour of Autonomous Systems". In: *Agent and Multi-Agent Systems*. Vol. 7327. LNCS. 2012, pp. 504–513. DOI: 10.1007/978-3-642-30947-2_55.

[Mol15] G. Molnár. “Developing a Model-based Regression Testing Tool”. MSc thesis. Budapest University of Technology and Economics, 2015.

[MW11] Z. Micskei and H. Waeselynck. “The many meanings of UML 2 Sequence Diagrams: a survey”. In: *Softw. Syst. Model.* 10 (4 2011), pp. 489–514. DOI: 10.1007/s10270-010-0157-9.

[OMG17] OMG. *OMG Unified Modeling Language (UML)*. formal/17-12-05. 2017.

[OMG19a] OMG. *OMG Systems Modeling Language (SysML)*. formal/19-11-01. 2019.

[OMG19b] OMG. *Precise Semantics of UML State Machines (PSSM)*. formal/19-05-01. 2019.

[Pac+07] C. Pacheco et al. “Feedback-Directed Random Test Generation”. In: Int. Conf. on Software Engineering (ICSE). 2007, pp. 75–84. DOI: 10.1109/ICSE.2007.37.

[Par98] D. L. Parnas. ““Formal methods” technology transfer will fail”. In: *J. Syst. Software* 40.3 (1998), pp. 195–198. DOI: 10.1016/S0164-1212(97)00166-0.

[Ral+21] P. Ralph et al. *Empirical Standards for Software Engineering Research*. 2021. DOI: 10.48550/arXiv.2010.03525. arXiv: 2010.03525 [cs.SE].

[SF18] K.-J. Stol and B. Fitzgerald. “The ABC of Software Engineering Research”. In: *ACM Trans. Softw. Eng. Methodol.* 27.3 (2018). DOI: 10.1145/3241743.

[SFH20] N. Setiani, R. Ferdiana, and R. Hartanto. “Test Case Understandability Model”. In: *IEEE Access* 8 (2020). DOI: 10.1109/access.2020.3022876.

[Sha+15] S. Shamshiri et al. “Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges”. In: *Int. Conf. on Automated Software Engineering (ASE)*. 2015, pp. 201–211. DOI: 10.1109/ASE.2015.86.

[Siq+21] B. R. Siqueira et al. “Testing of adaptive and context-aware systems: approaches and challenges”. In: *Softw. Test. Verification Reliab.* 31.7 (2021). DOI: 10.1002/stvr.1772.

[TA20] Z. Tahir and R. Alexander. “Coverage based testing for V&V and Safety Assurance of Self-driving Autonomous Vehicles: A Systematic Literature Review”. In: *IEEE AITest*. IEEE, 2020. DOI: 10.1109/aitest49225.2020.00011.

[TH08] N. Tillmann and J. de Halleux. “Pex–White Box Test Generation for .NET”. In: Tests and Proofs. 2008, pp. 134–153. DOI: 10.1007/978-3-540-79124-9_10.

[Tót+17] T. Tóth et al. “Theta: a Framework for Abstraction Refinement-Based Model Checking”. In: *FMCAD*. 2017, pp. 176–179. DOI: 10.23919/FMCAD.2017.8102257.

[Var04] D. Varró. “Automated formal verification of visual modeling languages by model checking”. In: *Softw. Syst. Model.* 3.2 (2004). DOI: 10.1007/S10270-003-0050-X.

[VP03] D. Varró and A. Pataricza. “VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML”. In: *Softw. Syst. Model.* 2.3 (2003), pp. 187–210. DOI: 10.1007/S10270-003-0028-8.

[Woh+12] C. Wohlin et al. *Experimentation in Software Engineering*. Springer, 2012. DOI: 10.1007/978-3-642-29044-2.

[WUR22] D. Winkler, P. Urbanke, and R. Ramler. “What Do We Know About Readability of Test Code? - A Systematic Mapping Study”. In: *IEEE SANER*. 2022. DOI: 10.1109/saner53432.2022.00135.

[YH12] S. Yoo and M. Harman. “Regression testing minimization, selection and prioritization: a survey”. In: *Softw. Test. Verification Reliab.* 22.2 (2012), pp. 67–120. DOI: 10.1002/stvr.430.