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

Mutációs tesztelés alkalmazása biztonságkritikus beágyazott rendszerek tesztelésében

Készítette
Serban Andrada Alexia

Konzulens
Micskei Zoltán, PhD
Rozgonyi Péter

2022

TARTALOMJEGYZÉK

Összefoglaló

A biztonságkritikus beágyazott rendszerek változatos területeken – például a közlekedésiparban, az energiaellátásban vagy az egészségügyben – látnak el kulcsfontosságú feladatokat. Egy potenciális meghibásodás számos ember életét követelhetné vagy hatalmas anyagi kárt okozhatna, ezért zökkenőmentes működésüket többek között szabványok által előírt szigorú teszteléssel biztosítják.

Mindezek ellenére felmerül a kérdés: *Quis custodiet ipsos custodes?* – Ki őrzi az őröket? Honnan tudjuk, hogy a tesztkészletünk lefed minden működést és kiküszöböltünk minden lehetséges hibát? Ugyan bizonyos biztonsági szinteken előírt a forráskód teljes lefedettsége a tesztek által, de ez bizonyítottan nem biztosíték arra, hogy minden hibát kiküszöböltünk és megfelelő mennyiségű és minőségű teszttel ellenőriztük kódunk megbízható működését.

A fenti problémára nyújt megoldást a mutációs tesztelés, mely a forráskód apró változtatásainak hatását vizsgálja a tesztek eredményeire. Amennyiben megfelelő tesztkészlettel rendelkezünk, a megváltoztatott, úgynevezett mutáns kód jelenlétét mindenképpen jelezni fogja. Ugyan a módszer elvi hátterét közel öt évtizede dolgozták ki, és azóta is jelentős tudományos munka tárgya volt, az ipari térnyerés azonban csak az elmúlt 10-15 évben kezdődött meg, és most sem mondható széles körben elterjedtnek. Ez részben a laboratórium kereteken túllépő, ipari komplexitású kód feldolgozásának nehézségei, részben pedig a módszert jellemző, ipari környezetben szintén fokozódó számítási kapacitásigénynek tudható be.

Egy ipari esettanulmány kapcsán biztonságkritikus beágyazott rendszerek – vasúti fékrendszerek – tesztelésével foglalkoztam. Megvizsgáltam a mutációs tesztelés bevezetésének lehetőségét, azonban az elérhető eszközök közül egyik sem volt kompatibilis az adott környezeten belül használt szoftverekkel. Így egyértelművé vált, hogy saját szoftver fejlesztésére lesz szükség.

Munkám során egy olyan programot hoztam létre, mely alkalmas az ipari léptékű C kód feldolgozására és a cég által használt tesztszoftverekkel integrált mutációs tesztelésére, emellett rendelkezik a fejlesztői és tesztelői igényekhez szabható funkcionalitással és optimalizációs lehetőségekkel.

Az elkészített szoftver az elérhető nyílt forráskódú szoftverekkel összemérhető funkcionalitású: hasonló mennyiségű és típusú mutációra képes. Működését előbb kisebb, nyilvánosan elérhető forráskódokon és a hozzájuk generált teszteken, majd egy korábbi ipari projekt tesztkészletén értékeltem ki. Összehasonlítottam a generált, illetve a külső és belső dolgozók által írt tesztek minőségét, illetve megvizsgáltam, hogy elkülöníthetők-e hatékonyabb operátorok vagy jellegzetes hibák a kapott eredmények alapján.

A későbbiekben az eredmények a tesztelési és fejlesztési folyamatok javítására használhatóak, a mutációs tesztelés integrációja ezekben a folyamatokban előreláthatólag emeli majd a megírt tesztek minőségét és mennyiségét is.

Abstract

Safety-critical embedded systems play key roles in various areas of our life, such as transportation, healthcare, or the electric power industry. A potential fault could harm human life or cause severe financial damage, therefore correct operation is ensured by rigorous testing – amongst other practices – specified by multiple standards.

Despite the aforementioned regulations, the question emerges: *Quis custodiet ipsos custodes?* – Who watches the watchers? How can we be assured that our test suite covers all functionality and that we prevented all possible faults? Even though complete code coverage is required on certain safety levels, it has been proved that it is no guarantee of reliable operation, nor is it of the suitable amount or quality of tests.

Mutation testing provides a solution for the above problem: by making a small modification to the source code and monitoring its effects on the test results we obtain insights about the quality of our test suite. If our test suite is comprehensive enough, the modified – "mutant" – code will always cause at least one test to fail. Even though the theoretical background of this method was developed almost half a century ago and has been the subject of thorough academic research ever since, industrial applications only started emerging in the past 10-15 years, and to this day it cannot be considered a widespread practice. This is partly due to the difficulties surrounding parsing and processing code of industrial complexity, but also caused by the computing-intensity of the testing process that is only amplified in an industrial setting.

I dealt with testing safety-critical embedded systems – specifically rail braking systems – as a part of an industrial case study. Upon assessing the possibilities of introducing mutation testing, I found that none of the available tools were compatible with software used in the given environment. Thus, it became obvious that the development of a custom software was required.

In my work I created a program that is capable of processing industrial grade C code and is compatible with the test environment used in the company. It is also configurable to developers' and testers' needs and has optimalization possibilities.

The result is a software comparable to available open-source tools: it is capable of similar types and quantities of mutation. I first tested its functionality on smaller, publicly available source code and generated test suites, then ran it on an earlier industrial project. I compared the quality of generated, external, and internal tests and investigated whether the results revealed more efficient operators or any typical errors in source or test code.

Subsequently the results can be used to improve development and testing processes and continuous integration of mutation testing into these processes will elevate both quality and quantity of the tests created.

1. Bevezetés

Munkámban egy tesztvalidációst eljárást, a *mutációs tesztelés* alkalmazását vizsgálom meg és értékelem ki egy biztonságkritikus beágyazott rendszerek szoftvertesztelésével kapcsolatos ipari esettanulmány részeként. Ugyan a módszert az elmúlt évtizedekben részletes méréseknek, kutatásoknak vetették alá, valós, ipari eredetű forráskódon, különösen biztonságkritikus beágyazott rendszerek tesztelésében történő alkalmazásáról kevés vizsgálati eredmény áll rendelkezésre [1].

A biztonságkritikus (beágyazott) rendszerek tesztelésére vonatkozó irányelveket szigorú szabványok rögzítik, melyek egységtesztek (unit test) szintjén elsősorban különböző mélységű kódlefedettséget követelnek meg. A kódlefedettség megléte kulcsfontosságú és alapvető követelmény, azonban, ha annak kevésbé szigorú változatát követeljük csak meg közel nem biztosítja a kód lehetséges hibáinak és teljes funkcionalitásának lefedettségét [2] [3] [4].Emellett a tesztbemenetek nem megfelelő megadása, a határértéken történő tesztelés (boundary value testing) elmaradása tipikus hibája a manuálisan elkészített teszteknek.

A felsorolt hibák kiküszöbölése emberi úton nehezen kivitelezhető teljes bizonyossággal, azonban a mutációs tesztelés módszertana egy automatizálható megoldást nyújt a problémára. A módszer működési elve, hogy a teljesnek gondolt tesztkészlettel szemben megfelelő, ezáltal hibátlannak minősített kódon apró változtatásokat ejt úgynevezett mutációs operátorok használatával. Amennyiben tesztkészletünk valóban teljesen lefedi a tesztelt program működését, semmilyen más kódvariánst nem engedhet át, hiszen az nem ugyanazt a működést valósítja meg: legalább egy tesztesetnek buknia kell. Amennyiben nem bukik teszteset, a mutáns életben maradt, ennek okai vizsgálandóak, az esetleges tesztelési hiányosságok pótlandóak. Ezen módszer használatának lehetséges motivációit, elméleti hátterét, kialakulásának és fejlődésének történetét és néhány alkalmazását mutatom be dolgozatom 2. fejezetében.

A vizsgált ipari esettanulmányban felmerült a mutációs tesztelés alkalmazásának igénye. A 2. fejezetben összefoglalt kutatásaim, illetve az ipari környezetből érkező követelmények összevetését követően megállapítottam, hogy az igényeknek megfelelő mutációs tesztelést megvalósító eszköz nem áll rendelkezésemre, így egy saját szoftver fejlesztése vált szükségessé. A fejlesztés során több ízben okozott problémát az ipari kód mérete és komplexitása, így végül egy olyan eszköz jött létre, mely a nyilvánosan elérhető, nyílt forráskódú eszközöktől legfeljebb kismértékben elmaradó, azokkal összemérhető funkcionalitással rendelkezik, azonban képes kezelni az esettanulmányban használt tesztkörnyezetet. Dolgozatom 3. fejezetében a szoftverrel szemben támasztott követelményeket, a tervezés során meghozott döntéseket, illetve a létrejött program felépítését és működését részletezem.

A munkám során fejlesztett mutációs tesztelést megvalósító eszközt először nyilvánosan elérhető, a kapcsolódó cikkekben más hasonló eszközök felmérésére használatos

forráskódokon és tesztkészleteken teszteltem, majd az ipari környezetben használt automatikus tesztelést is megvalósító szoftver által generált tesztkészleteket vizsgáltam meg. Végül a valós bemenetét – biztonságkritikus beágyazott szoftver és annak komponenstesztjeit – is mutációs tesztelésnek vetettem alá. A kapott eredményeket az általam fejlesztett eszköz kiértékelésére, más kutatási eredmények jelen környezetben való megjelenésének vizsgálatára, illetve a beágyazott szoftver tesztjeinek tipikus hiányosságainak és hibáinak felderítésére használtam. Dolgozatom 4. fejezetében ezen laboratóriumi és ipari futtatások eredményeit, illetve az eszköz továbbfejlesztési lehetőségeit mutatom be.

Dolgozatom elméleti és gyakorlati eredményeinek összefoglalása:

- Az ipari esettanulmány követelményeit figyelembe véve megterveztem egy olyan mutációs tesztelő eszközt, ami skálázódik az ott felhasznált bonyolultságú programokra. Kísérleti mérésekkel összehasonlítottam az eszközt más, hasonló eszköz teljesítményével.
- Kísérleti mérésekkel összehasonlítottam az eszközöm által megvalósított optimalizáció hatását, más, hasonló optimalizációt alkalmazó mérések eredményeivel.
- Megvizsgáltam az ipari esettanulmányban használt tesztszoftver automatikus, kódlefedettséget biztosító tesztgenerálását az általam létrehozott mutációs tesztelő eszköz segítségével.
- Az ipari esettanulmányon kiértékeltem, hogy milyen operátorok hatékonyak és azok milyen tesztelési vagy fejlesztési hibák felderítésére alkalmasak. A kapott eredményeket összevetettem a hatékony operátorokról szóló irodalmi eredményekkel.

A kapott eredményeket az esettanulmányban résztvevő tesztelői, illetve fejlesztői csoportnak bemutatva igyekszem minél hatékonyabb tesztelői szemléletet kialakítani. Emellett célom a létrehozott eszköz mindennapos tesztelésbe integrálása és további adatok gyűjtése. Ez hosszútávon a létrehozott tesztek minőségének és mennyiségének javítását eredményezheti.

2. Mutációs tesztelés áttekintése

Ebben a fejezetben ismertetem a mutációs tesztelés alkalmazásának motivációját, lehetséges indokait, majd a módszertanát és elméleti alapjait. Rövid áttekintést adok a módszer születéséről és az azt követő tudományos eredményekről, majd bemutatom a kutatásaim során általam vizsgált mutációs tesztelést megvalósító eszközöket.

2.1. Motiváció: Tesztek ellenőrzése

A(z ipari esettanulmányban szereplő) vasúti fékrendszerekre az IEC 61508-as szabványban megfogalmazott, biztonságkritikus funkciót ellátó programozható elektronikus rendszerekre vonatkozó irányelvek mérvadóak. Eszerint a tervezés során szükséges a rendszer biztonsági követelményeinek megállapítása, majd ezek elosztása részrendszerekre. Ezen részrendszerekhez kvantitatív vagy kvalitatív módszerekkel úgynevezett biztonságintegrási szinteket (Safety Integrity Level, SIL) rendelnek, melyek meghatározzák a továbbiakban rájuk vonatkozó biztonsági előírásokat [5].

A tanulmányomban szereplő rendszerek zömében a SIL 2-es szinthez tartoznak, így szigorú előírásoknak megfelelő tesztelési folyamaton mennek keresztül. Előírt a kód minden sorának tesztek általi lefedettsége, de ez gyakran nem fedi le annak teljes funkcionalitását. Ugyan ez a lefedettség elengedhetetlen a szoftver és a tesztek minőségbiztosításához, több tanulmány is kimutatta, hogy korántsem biztosítja tesztkészletünk hatékonyságát [3] [2] [4].

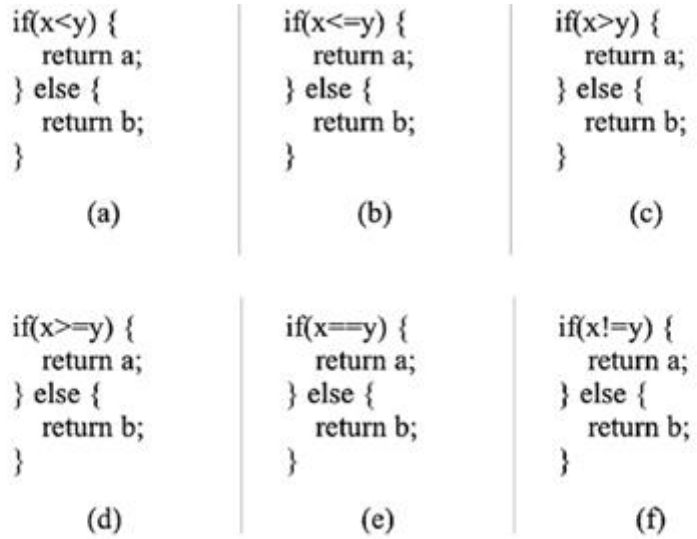
Emellett a szigorú előírások betartása, illetve – méretéből és komplexitásából kifolyólag – az iparban előálló kód is nagy mennyiségű tesztet generál. Ezeknek a teszteknek a megírása és ellenőrzése szinte teljes mértékben emberi feladat, és bármilyen jó szakembereket alkalmazunk, az emberi tényező mindig hibalehetőséget rejt magában. Erre nyújt részleges megoldást a különböző tesztszoftverek automatizált tesztelési funkciója, amivel a lefedettséget biztosító tesztesetek generálhatóvá válnak. Ebben is rejtőznek sajátos korlátok: összetettebb adatszerkezetek vagy feltételrendszerek esetén a tesztgenerálás már nem lehetséges, és a generált tesztek emberi ellenőrzése és kiegészítése is mindenképpen szükséges.

A fent felsorolt hibalehetőségek – mind a gépi hiányosságok, mind az emberi tényező – ellenőrzésére, kiküszöbölésére nyújt megoldást a mutációs tesztelés, melynek elvi hátterét a következő pontban részletezem.

2.2. A mutációs tesztelés elméleti alapjai

A *mutációs tesztelés* azon a feltételezésen alapul, hogy amennyiben tesztkészletünk valóban átfogó, a szoftver teljes működését és a kód minden sorát lefedő tesztekből áll, akkor a forráskódon ejtett legapróbb funkcionális változtatást is bukott tesztesettel kell jeleznie. Ha például két szám összehasonlítását átírjuk „kisebb”-ről „kisebb vagy egyenlő”-re a határon viselkedést vizsgáló tesztesetnek ezzel buknia kell.

Mutáció során a forráskódban megkeressük azokat a pontokat, ahol a fenti példához hasonlatos változtatások eszközölhetők a kódon. Az így létrejött egyetlen, elemi változtatást tartalmazó forráskódvariánsokat hívjuk (elsőrendű) *mutáns*nak, a változtatás szabályait *mutációs operátorok*nak (lásd 1. ábra [6]). A tesztelés során minden mutánson lefuttatjuk a tesztkészletünket és vizsgáljuk annak kimenetelét: amennyiben keletkezik bukott teszteset a mutánst „*megöltük*”, a kód azon pontját valóban lefedtük a megfelelő tesztesetekkel. Ellenkezőesetben pedig a mutáns *életben maradt*, mely vagy tesztkészletünk hiányosságaira mutat rá, vagy ekvivalens mutációnak köszönhető. A mutációs tesztelés futtatásának jellemző kimenete még a *mutációs pontszám* (mutation score), mely a megölt mutánsok arányát mutatja az összes létrehozott mutánshoz képest.



1. ábra: Mutációs tesztelés bemutatása: egyszerű kódrészlet (a) és mutánsai (b-f) [6]

A használható mutációs operátorok részben függenek a tesztelt kód programozási nyelvétől, de a leggyakrabban előforduló, egyszerűbb operátorok a legtöbb gyakorlatban használt nyelvben alkalmazhatóak. Az összes lehetséges operátor használata indokolatlan és teljesítményigényes feladat, számos kutatás témája is volt. Ennek részleteit és a munkámban alkalmazott operátorokat dolgozatom 3.3-as pontjában részletesebben tárgyalom.

A létrejött mutánsok életben maradása is további vizsgálatot érdemel, mivel okai nem feltétlenül a tesztkészletünkben keresendők. Megeshet, hogy olyan mutáns jön létre, mely bár szintaktikailag különbözik az eredeti forráskódtól, szemantikailag megegyezik vele. Ez az úgynevezett *ekvivalens mutánsok problémája*, melynek kiküszöbölése népszerű és nem lezárt tudományos kérdés, és a teljesítményigényesség mellett a másik fő korlátozó tényező a mutációs tesztelés elterjedésében. Az ekvivalencia tényének megállapításán kívül érdemes kivizsgálni az ekvivalencia okát, mely nem feltétlenül maga a mutált kódrészlet egyenértékű viselkedése, hanem annak „halott”, a kód lefutását nem befolyásoló mivolta is lehet.

2.3. Történeti áttekintés

A mutációs tesztelés gondolata először Richard Lepkinben fogalmazódhatott meg. Elmondása szerint 1971-ben egy egyetemi feladat kapcsán vázolta fel ötletét. Az első tudományos publikációk [7] [8] [9] 1977-ben jelentek meg, ezek FORTRAN, illetve SIMPL-T nyelvre íródott teszteszközökről szóltak. Már itt megfogalmazódott a probléma, ami végigkíséri a mutációs tesztelés további történetét: a teljesítményigényesség. Nem sokkal később, 1980-ban, Budd doktori disszertációjábana mutációs tesztelésről [10] pedig megjelenik az ekvivalens mutánsok problémája is.

Az ezeket követő 20 évben növekvő tendenciát mutat a publikációk száma, de ezek továbbra is elméleti értekezések, vagy laboratórium körülmények között működő programokról számolnak be. A Mothra projekt során létrejött az első széles körben elterjedt mutációs rendszer, mely számos kutató és egyetem keze alatt megfordulva, különböző módosításokkal alapjául szolgált tucatnyi korabeli kutatási kérdés megválaszolásának. A folyamat teljesítményigényességének csökkentésére is több szemlélet alakult ki: *kevesebb* mutáció, *okosabb* mutáció, vagy egyszerűen *gyorsabb* mutáció. Ezek közül munkámban a kevesebb mutáció alkalmazásának lehetőségeit és hatásait vizsgálom. Kevesebb mutációhoz alapvetően kevesebb, alaposan megválasztott operátor segítségével [11] [12], vagy a mutánsok valamilyen statisztikai mintavételezésével juthatunk [13] [14]. A gyorsabb mutáció alapvetően nem módszertan, mintsem inkább az alkalmazott algoritmus, idő és technológiai fejlődés kérdése, napjainkra mind a számítógépek számítási kapacitása, mind annak ára az elméletek létrejöttének időszakához képest lényegesen lecsökkent, így lehetőséget teremtve az elmélet gyakorlatban történő alkalmazására és szélesebb körben való elterjedésére.

A 2000-es évek robbanásszerű növekedést hoztak mind tudományos eredmények és publikációk, mind létrejött eszközök terén: a mutációs tesztelés alkalmazása túllépett a forráskód és egységtesztek szintjén és specifikációkra, illetve számos új programozási nyelvre is alkalmazták. Az ezredfordulóig megtett elméleti előrehaladás és a számítási kapacitás növekedése lehetővé tette a gyakorlati alkalmazások megjelenését, ma már pedig bárki hozzáférhet nyílt forráskódú mutációs tesztelést megvalósító szoftverekhez. 2014 óta alkalmazzák az eljárást a Google-nél, addig összegyűlt adataikat, azokból levont következtetéseiket és tapasztalataikat egy 2021-es tanulmányban publikálták [15], majd 2022-ben újabb cikket jelentettek meg eredményeikről [16]. Munkám során az előbbi tanulmányból két fontos optimalizációs lehetőséget használtam fel: kódsoronként egyetlen mutáns létrehozását és a szelektív mutációt, vagyis a használt operátorok megválasztását. Emellett a tanulmány során kiértékelésre került a mutációs tesztelés fejlesztésre és tesztminőségre tett hatása: a tesztelés során mutációs tesztelést folyamatosan alkalmazó fejlesztők több és hatékonyabb tesztet írnak. [17] [18]

2.4. Vizsgált mutációs eszközök

Munkám során kezdetben egy, az adott ipari környezetben felhasználható eszköz megtalálása érdekében, majd a mutációs tesztelést megvalósító szoftverek működési elveinek megismerése céljából több nyilvánosan elérhető, nyílt forráskódú eszközt is megvizsgáltam (lásd 1. táblázat) [19].

Név	Tesztelt kód nyelve	Operátorok	Kódfeldolgozás	Tesztek formátuma
dextool [20]	C/C++	ABS, AOR, LCR, ROR, UOI, COR, SDL, CR	LLVM → AST	.exe, visszatérési érték jelzi az eredményt
mull [21]	C/C++	Hiányos AOR, LCR, ROR, UOI, SDL, CR	LLVM → AST	.exe, visszatérési érték jelzi az eredményt
Mutate++ [22]	C++	Hiányos/módosított AOR, LCR, ROR, UOI, SDL, CR mellett egyéb, objektum-orientált nyelvekben használható operátorok	Reguláris kifejezések	
Proteum [23]	C	71 féle	AST generálás a forráskód része	teszteset visszatérési értéke
Milu [24]	C	20 féle	AST	teszteset visszatérési értéke
muJava [25]	Java	29 osztályszintű 16 kódszintű	AST	JUnit
PIT [26]	Java	Hiányos/módosított AOR, LCR, ROR, UOI, SDL, CR mellett egyéb, objektum-orientált nyelvekben használható operátorok	ASM (?)	JUnit
mutPy [27]	Python	20 + 7 kísérleti	AST	unittest, pytest

1. táblázat – vizsgált mutációs eszközök fontosabb tulajdonságai

Részletesen megvizsgáltam két, C nyelv mutációs tesztelésére készített szoftvert – a *dextool*-t és a *mull*-t – azok kódfeldolgozási módszereit, az elérhető operátor(csoport)okat, a tesztek kiértékelésének módját és a fejlesztésük során esetlegesen alkalmazott szakirodalmi forrásokat. Mindkét általam vizsgált szoftver működésének alapja az LLVM fordítóprogram-infrastruktúra. Ugyan először nem volt egyértelmű a döntés mögött álló ok, később világossá vált, hogy az úgynevezett AST – abstract syntax tree, vagyis *absztrakt szintaxisfa* kinyerésének egyszerűsége vezethetett ehhez. Az AST a forráskód absztrakt szintaktikai leírását reprezentáló fa, melynek minden csomópontja egy konstrukció a forráskódban. Míg más fordítóprogramokból (pl. gcc) problémás, vagy egyáltalán nem lehetséges az AST-hez való hozzájutás, mind LLVM-mel vagy Clang-gal 1-1 paranccsal könnyen feldolgozható fát kapunk. Ezt a fát használja fel mindkét program a mutációs helyek feltérképezésére és létrehozására. A két eszköz lényegében hasonló operátorokat biztosít, bár a dextool kissé bővebb, illetve a szakirodalomnak megfelelően csoportosított operátorkészlettel rendelkezik, míg a mull laikus felhasználói oldalról érthetőbb, jobban szűrt operátorokat használ. Ezt dolgozatom következő fejezetében részletesebben is tárgyalom. A tesztek kiértékelésénél mindkét eszköz a tesztapplikáció visszatérési értékéből indul ki, annak vizsgálatával dönti el, hogy egy mutáns életben maradt-e vagy sem.

Munkám során vizsgáltam egyéb mutációs tesztelést megvalósító nyílt forráskódú eszközöket is. C nyelv mutációjánál az LLVM vagy Clang által generált szintaxisfán kívül a reguláris kifejezések használata, vagy azoknak valamilyen saját implementációja (Proteum) fordult elő megoldásként a mutációs helyek feltérképezésére. Java, illetve Python nyelvek esetén is hasonló megoldások születtek. Általánosságban elmondható, hogy az elérhető szoftverek 1-1 fájl mutációjára alkalmasak, illetve 1-1 egyszerű tesztkészletet tudnak futtatni és kiértékelni, melyeknek formája rögzített, pl. parancssori input, a gtest könyvtár, .cmd fájl, Java esetén JUnit, vagy Python esetén a pytest vagy unittest könyvtár.

Vizsgálataim során megállapítottam, hogy a céges környezetbe implementálható, az ipari nagyságú kód- és tesztkészletet kezelni képes, nyilvánosan elérhető eszköz nem áll rendelkezésre. Tapasztalataim azonban megalapozták a létrehozandó eszközzel szemben támasztott követelményeket és a meghozandó fejlesztési döntéseket, melyeket a dolgozat következő fejezetében részletezek.

3. Saját mutációs tesztelő eszköz tervezése

Ebben a fejezetben először ismertetem a létrehozandó eszközzel szemben támasztott követelményeket, majd a megvalósítás során meghozott tervezési döntéseket. Külön pontban részletezem a felhasznált operátorokat, csoportosításukat, és kiválasztásuknak szakirodalmi vonatkozásait. Végül bemutatom a létrehozott program felépítését, illetve működését.

3.1. Követelmények

A vizsgált ipari esettanulmányban a biztonságkritikus beágyazott szoftver szabványoknak megfelelő unit-tesztelése a QA Systems által fejlesztett Cantata nevű tesztrendszer használatával valósul meg. A Cantata egy Eclipse alapú környezetben futó, C/C++ kód egység- és integrációs tesztelésére (unit test, integration test) alkalmas szoftver. Többek között képes automatikus tesztgenerálásra különböző kódlefedettségi követelményeknek megfelelően, különböző mértékben elkülöníthetők benne a tesztelendő függvények, és használható statikus változók és konstansok elérésére és manipulációjára is. Alapvetően grafikus kezelőfelületen keresztül érhető el, de parancssorból is futtatható. A teszteredmények egyrészt megtekinthetőek a grafikus felületen, az ott található terminálon kiírásra kerülnek, de logként is megtalálhatóak a tesztapplikáció könyvtárában.

A munkám során a mutációs szoftver tesztelésére és a mérések végrehajtására felhasznált kód egy fejlesztés alatt álló vasúti fékrendszer fékezési funkcionalitását megvalósító kártya kártyaspecifikus forráskódja és annak unit tesztjei. A forráskód kizárólag C nyelven íródott. Összesen 15 komponensből áll, a komponensek egyenként néhány .c forrásfájlból, a hozzájuk tartozó .h fejlécfájlokból és az egyéb, alapszoftverhez (base software) tartozó fejlécfájlokból, esetleg statikus könyvtárakból állnak. A tesztek komponensenként 1-1 Cantata projektben találhatóak, általában .c fájlonként vagy azokon belül funkció alapján elkülönített tesztkészletekben. Mind a szoftver végleges fordításához, mind a tesztek fordításához GNU GCC használatos, a végleges applikációban így az ehhez a fordítóprogramhoz tartozó C könyvtárak lesznek felhasználva. A forráskód nagy mennyiségű preprocesszor makrót és typedef-et tartalmaz, így a preprocesszált kód méretében és tartalmában is lényegesen eltérhet az eredeti forráskódtól, emiatt implementálandó a preprocesszált kód tesztelés forrásaként alkalmazásának lehetősége.

Elvárás volt még a létrehozandó programmal szemben a használandó operátorok „konfigurálhatósága”: az eszköz által alkalmazott operátorcsoportok tetszőleges kombinációban kiválaszthatóak és futtathatóak. Emellett szükséges még csak adott kódszakasz mutálásának lehetősége, regressziós futtatások esetére.

Elsősorban kutatási és mérési célokból implementálandó, de a folyamatos fejlesztés közbeni ellenőrzések alatt is felhasználható optimalizáció a kódsoronként egyetlen mutánst létrehozó funkció. Szintén a mérések kiértékelhetőségének érdekében fontos valamilyen log funkció létrehozása, mely a futtatási eredmények feldolgozását segíti elő.

Követelmények:

- **R1:** Cantata integráció – tesztapplikáció újrafordítása, tesztek futtatása és kiértékelése
- **R2:** Egyszerre több tesztkészlet és/vagy forrás kezelése
- **R3:** Preprocesszált forráskód mutációjának lehetősége
- **R4:** Használt operátorok beállításának lehetősége
- **R5:** Tetszőleges kódrészlet mutációjának lehetősége
- **R6:** Optimalizáció használatának lehetősége
- **R7:** Táblázatkezelő szoftver által olvasható log készítésének lehetősége

2. ábra – A létrehozandó eszközzel szemben támasztott követelmények

3.2. Tervezési döntések

A fenti körülményekből következik, hogy a létrehozott szoftvernek a leírt adatszerkezetet és könyvtárrendszert kell tudnia kezelni. A Cantata-val való integráció a parancssori interfészen keresztül valósítható meg, a teszteredmények vizsgálata pedig a logfájl feldolgozásával lehetséges. Mivel a fordítás során mindvégig GCC-t használunk, az egyéb fordítóprogramok (pl. LLVM/Clang) használata kompatibilitási problémákhoz vezethet és ezért elhagyásra került.

A követelmények teljesítése érdekében az eszköz megvalósításához a Python nyelvet választottam. A szöveges állományok feldolgozása, kezelése jelentősen egyszerűbb és kényelmesebb, mint más, általam ismert nyelveken, emellett lehetséges a parancssor elérése és így fájlok mozgatása, törlése, átnevezése, illetve más programok (GCC, Cantata) futtatása közvetlenül programból.

A forráskód feldolgozására és a mutációs helyek megtalálására korábban említett tapasztalataim alapján két lehetőség áll rendelkezésre:

a) **valamilyen parser** segítségével az absztrakt szintaxisfa kinyerése, vagy
b) **reguláris kifejezés**ek alkalmazása a kódon.

Az előbbi módszerrel jóval nagyobb számú és típusú mutáció megbízhatóbb alkalmazása lehetséges, ezért elsőként ennek alkalmazására tettem kísérletet.

Először egy kisebb, nyílt forráskódú, tisztán Python-ban íródott eszközzel, a pycparser-rel [28] próbálkoztam, ez azonban egyáltalán nem képes preprocesszor direktívák értelmezésére, ezért csak preprocesszált kód feldolgozására alkalmas. Következőnek egy összetettebb, szintén nyílt forráskódú, Java-ban íródott parser generátor, az ANTLR [29] segítségével elkészített, Pythonban futtatható C parsert hoztam létre a nyilvánosan elérhető C nyelvtant leíró fájl segítségével [30]. Ugyan hibaüzenet nélkül lefutott az ipari kódon is, a futási eredményeket kiértékelve láthatóvá vált, hogy hibásan dolgozta fel azt, mert a különálló fájlokban definiált típusokat nem megfelelően ismerte fel. Egyértelművé vált, hogy mindkét eszköz használatához preprocesszált fájlra van szükség. Ez ellehetet-

lenítette a preprocesszált fájlon való futtatás ki-bekapcsolhatóságát, de a mutációk mennyiségi és minőségi fejlődéséhez vezethetett volna, így alkalmas kompromisszumnak ígérkezett. GCC-t alkalmazva azonban a preprocesszált fájl továbbra is tartalmaz preprocesszor direktívákat: define-ok, include-ok, illetve az include-okat feltérképező egyéb sorok, emellett a sztenderd könyvtáraknak nemszabványos, fordítóspecifikus implementációi jelentek meg a kódban, ráadásul a forrásfájl mérete is jelentősen megnövekedett. A direktívák fennmaradása miatt a pycparser, a nemszabványos implementációk miatt pedig az ANTLR által generált parser sem tudott megfelelően lefutni a preprocesszált kódon, így ez a megoldás elvetésre került.

A parser programok sikertelen alkalmazási kísérletei után megkíséreltem a szintaxisfa közvetlenül GCC-ből való kinyerését. A fordító megfelelő konfigurálásával kaphatóak olyan fájlok, mely a szintaxisfát írják le, de ezek közül a szövegként feldolgozhatóak nem tartalmaztak megfelelő mennyiségű és minőségű információt.

Az ideális működést lehetővé tevő, parsert használó módszerek ipari alkalmazásának korlátjaiba ütközve végül reguláris kifejezések alkalmazása mellett döntöttem. Így a program futtatható az eredeti forráskódon és annak preprocesszált verzióján is, azonban bizonyos operátorok alkalmazása korlátokba ütközik. A * és & operátorok felismerése C-beli kétértelműségük miatt nem kivitelezhető megbízhatóan. Emellett a feltételvizsgálatok felismerése is csak akkor lehetséges, ha azokat egyetlen kódsorban íródtak, az ABS operátor megbízható alkalmazása pedig bizonyos esetekben teljesen lehetetlenné válik.

3.3. Kiválasztott operátorok

Ahogy az előző fejezetben is említettem, a mutációs tesztelés során használható operátorok részben függenek a tesztelt kód programozási nyelvétől, de a leggyakrabban előforduló, egyszerűbb operátorok a legtöbb gyakorlatban használt nyelvben alkalmazhatóak. Jellemző művelet az összehasonlító operátorokat vagy komplementerükre cserélni vagy egyenlőséggel kiegészíteni/attól megfosztani, gyakori a matematikai operátorok párjukra cserélése, a visszatérési értékek megváltoztatása, a feltételvizsgálatok kiiktatása konstanssal. Objektum-orientált esetben tovább bővülnek a lehetőségek, például a konstruktorok vagy destruktorok mutálásával, különböző példányosításokkal. Széleskörben elterjedt programozási nyelvek esetében fellelhetőek átfogó vizsgálatok a felhasználható operátorokról és azok hatékonyságáról

Az esettanulmányomban szereplő tesztelendő beágyazott kód C nyelven íródott. A C nyelvű forráskód mutációs tesztelésére használható operátorokról DeMillo és tsai.

Operator	Description	Example
ABS	Absolute Value Insertion	$a = b + c \ to \ a = 0$
AOR	Arithmetic Operator Replacement	$a = b + c \ to \ a = b - c$
LCR	Logical Connector Replacement	$a = b \& c \ to \ a = b \vert c$
ROR	Relational Operator Replacement	$while(a < b) \ to \ while(a > b)$
UOI	Unary Operator Insertion	$a = b \ to \ a = -b$

3. ábra – Offut és tsai. által elkülönített operátorcsoportok [31]

1989-ben jelentettek meg átfogó tervezetet, melyben közel 80 féle operátorcsoportot definiáltak. Ezeknek implementációja egyrészt munkaigényes, másrészt nem feltétlenül indokolt: Offut és tsai. szelektív mutációt vizsgáló tanulmányában különböző mutációs operátorok elhagyásának hatását vizsgálva arra jutottak, hogy elkülöníthető 5 kulcsfontosságú operátorcsoport (lásd 3. ábra [31]). Az általam részletesen vizsgált, nyilvánosan elérhető, nyílt forráskódú, C kód széleskörű mutációjára alkalmas eszközök (dextool, mull) ezeket az operátorokat biztosítják, de a Google-nél folytatott esettanulmányban is ezeket alkalmazták.

A két vizsgált eszköz különböző módon csoportosította az alkalmazható operátorokat. A dextool a szakirodalombeli csoportosításra hivatkozik és azt is alkalmazza. A fenti 5 csoporton kívül 3 másik csoportot is említ: COR (conditional operator replacement) – egy feltételvizsgálat feltételének kicserélése, SDL (statement deletion) – egy kódsor törlése, és CR (constant replacement) – egy változó valamilyen konstansra, pl. 0-ra cserélése. Szintén a szakirodalombeli definíciókhoz pontosan ragaszkodva, AOR, LCR, illetve ROR alkalmazásakor az adott operátort a csoportban használható minden másik operátorra cseréli, nem csak egy valamilyen szabály szerint definiált párjára. A mull dokumentációjában nincs ilyen részletes szakirodalmi hivatkozás: az operátorok szimbólumpárokként (pl. == → !=) vannak definiálva, ezek a mutációk pedig különböző hierarchikus csoportokba vannak rendezve, ezeket választhatja ki a felhasználó. Az eszköz nem is fed le minden lehetséges kombinációt, az egész programban minden szimbólumnak csak egy párja van, ezek pedig az AOR, LCR, ROR, UOI, CR és SDL kategóriákba sorolhatók.

Emellett említendő még a Java kód mutációs tesztelésére használatos PIT nevű eszköz operátorai és azok csoportosítása, mely kezdetben szintén nem a szakirodalmi csoportosítást és lefedettséget biztosította, de az újabb verziókban megkezdte az AOR, ABS, AOD, CRCR, OBBN, ROR és UOI szakirodalmi operátorok kísérleti bevezetését.

Munkámban én is csoportosítottam az alkalmazható operátorokat, azonban mullban és PIT-ben látottakhoz hasonlóan ezek a csoportok nem egyeznek meg a szakirodalmi csoportosítással. Szintén hasonlóan AOR, LCR és ROR alkalmazásakor csak bizonyos párosításokat használok, emellett az ABS és SDL operátor használatát teljesen mellőzöm, a CR és UOI operátorait pedig csak bizonyos esetekben tudom alkalmazni. Emögött vagy a kód parse-olásának hiánya és a reguláris kifejezések használata által állított korlátok, vagy a forráskódban való alkalmazásukkal járó fordítási vagy futtatási nehézségek (pl. nullával osztás, végtelen ciklus) állnak. Előnye azonban ennek a szűrésnek, hogy lecsökkenti a létrejövő mutánsok számát, miközben a tesztelés hatékonysága legfeljebb kismértékben csökken [15].

Kategória	Eredeti kód	Mutáns kód	Mothra kategória
Conditional boundary	a<b	a<=b	ROR
	a>b	a>=b	
	a<=b	a<b	
	a>=b	a>b	
Increment invert	a++, ++a	a--, --a	UOI
	a--, --a	a++, ++a	
Mathematical	a+b	a-b	AOR
	a-b	a+b	
	a*b	a/b	
	a/b	a*b	
	a&b	a|b	LCR
	a|b	a&b	
	a&&b	a||b	
	a||b	a&&b	
	a>>b	a<<b	AOR
	a<<b	a>>b	
Conditionals negation	a<b	a>=b	ROR
	a>b	a<=b	
	a<=b	a>b	
	a>=b	a<b	
	a==b	a!=b	
	a!=b	a==b	
Boolean invert	TRUE	FALSE	UOI
	FALSE	TRUE	
Remove/negate condition	if (…)	if (TRUE)	COR
		if (FALSE)	
		if (!(…))	UOI
Return Null	return (…);	return NULL;	CR

2. táblázat – a saját fejlesztésű eszközben alkalmazott mutációs operátorok

3.4. Architektúra

Az elkészített programot az átláthatóság és a tesztelhetőség érdekében több, különálló .py fájlra bontottam szét.

A *config.py* fájl egyben a felhasználói kezelőfelület is. Felsorolandóak itt az adott Cantata projektben található tesztkészletek és a hozzájuk tartozó forrásfájlok nevei, a forráskód és a Cantata workspace elérési útvonala. Emellett szükséges a Cantata kezelőfelületéről kimásolható fordítási parancs és annak argumentumai. Ugyanitt beállíthatóak a használandó operátorok, a részletes logolás, a preprocesszált fájl vagy az optimalizáció használata, illetve – ha szükséges – a mutálni kívánt kódszakasz fájlonként. Helyet kapott még két kényelmi funkció: a halott mutánsok törlése és a felhasználói visszajelzésre várás a mutánsok létrehozása után, a tesztelés futtatása előtt.

A *mutators.py* fájlban helyeztem el a különböző operátorokat és azoknak reguláris kifejezésekkel való leírását, Python dictionary-kbe rendezve. Az egyes dictionary-k különböző az előző pontban részletezett operátorcsoportokat jelképezik.

A *mutant.py* fájl egy mutánsok adatainak tárolására létrehozott osztály implementációja. A *findMut.py* fájlban találhatóak a forráskódot feldolgozó függvények, a *createMut.py* fájlban a megtalált mutációs helyeken a mutáns fájlokat létrehozó függvények, a *testMut.py* fájlban pedig a mutánsokon a tesztek futtatását és kiértékelését végző függvények.

A *MUTT.py* fájl maga a felhasználó által futtatandó script, mely az eddig említett forrásokat megfelelően hívva és a felhasználói inputot kezelve elvégzi a mutációs tesztelést és létrehozza a statisztikát. Emellett az eszköz tesztelése során létrehoztam egy *test.py* scriptet melyet a különböző egységek működésének tesztelésére használtam.

3.5. Algoritmus

Az alkalmazott algoritmus egyszerű iteratív módszerekkel dolgozik. Ugyan kisebb mértékben elképzelhető, hogy optimalizálható lenne a futási idő (tipikusan másodpercek), de ez a forráskódok és tesztek újrafordításából adódó időhöz (tipikusan órák) képest nagyságrendileg elhanyagolható mértékű lenne.

Kezdetben a program minden megadott fájlból eltávolítja a kommenteket vagy átadja a preprocesszornak. Az így létrejött nyers fájlból reguláris kifejezések segítségével kinyeri a lehetséges mutációs helyeket és ezeket az erre a célra létrehozott osztályban tárolja. Amennyiben a soronkénti egy mutációs optimalizáció be van kapcsolva, ha egy adott sorban már talált mutánst, nem tesz kísérletet további mutánsok felderítésére. Ha csak adott sorokon mutálandó a forráskód, a többi sort figyelmen kívül hagyja. A megtalált mutációs helyeket fájlonként listában gyűjti össze.

A lehetséges mutációk felderítése után a szoftver létrehozza a mutáns fájlokat az eltárolt lista alapján és ezeket sorszámozott, a mutáció adataival kommentezve fájlként menti a forráskód könyvtárába. Ezt követően minden létrejött mutánsra lefordítja, majd lefuttatja a tesztkészletet. A tesztek futása során keletkezett logfájlt feldolgozva megállapítja, hogy életben maradt-e a mutáns, a halott mutánst pedig vagy törli vagy megjelöli.

Futás közben statisztika készül a mutációs pontszám megállapítása érdekében, illetve – amennyiben igényeltük – részletes, táblázatkezelőbe importálható logot is kaphatunk a futási eredményekről.

```
GET sourceFiles                         //Konfigurációs fájlból

FOR file IN sourceFiles
    FOR mutOp IN operators
        IF mutOp is used
            CALL FindMutants(file)   //Mutációs helyek feltárása és mutáció
elvégzése
                                        //reguláris kifejezések segítségével
            CALL CreateMutants()        //Mutáns fájlok létrehozása

FOR file IN mutantFiles
    CompileFile                         //mutáns fájl lefordítása
    CompileTests                           //forráskód és tesztkód applikációba
fordítása
    RunTests                            //tesztapplikáció futtatása
    result <-- getResult                //eredmények kiértékelése
    IF result=PASSED                    //összes teszt átment
        liveMutants += file
    ELSE                                //volt bukott teszt
        killedMutants += file
    LogResult                          //részletes logolás (ha be van kapcsolva)

PRINT liveMutants                       //élő mutánsok fájlnevei
PRINT killedMutants/allMutants          //Mutációs pontszám
```

4. ábra – Az eszköz működésének pszeudokódos leírása

4. Kiértékelés

A következő fejezetben az irodalomban használt benchmark kódok használatával felmérem az elkészített eszköz más eszközökhöz viszonyított teljesítményét. Ugyanezen kódokon megvizsgálom az optimalizáció eredményekre tett hatását. Ezt követően elvégzem az ipari esettanulmányból származó tesztszoftver automatikus tesztgenerálásának mutációs teszteléssel való értékelését, majd a forráskód és a hozzátartozó komponenstesztek teljes mutációs tesztelését. Ennek eredményeiből következtetek a tipikusan előforduló tesztelési, illetve forráskódbeli hibákra, az elkészített eszköz hiányosságaira és a különböző operátorok hatékonyságára.

4.1. Kiértékelés megtervezése

4.1.1. Célkitűzések

Az elkészített eszköz kiértékelését először az irodalomban széles körben felhasznált benchmark kódokon végzem el. Ezek közül a hosszabb, statisztikailag szignifikánsabb mennyiségű mutánst generáló kódon megvizsgálom az általam használt operátorokon az optimalizáció hatását. Végül az ipari esettanulmányban használt tesztszoftver automatikus, kódlefedettség alapú tesztgenerálását minősítem az általam létrehozott mutációs eszköz segítségével. A célokhoz kapcsolódó konkrét kutatási kérdések (research question):

- **RQ1:** Hogyan teljesít a saját eszköz az irodalomban felhasznált benchmarkokon más mutációs tesztelő eszközökről publikált eredményekhez képest?
- **RQ2:** Mennyiben befolyásolja a tesztelés eredményét kódsoroként egyetlen mutáns létrehozása?
- **RQ3:** Hogyan teljesít a teljes kódlefedettséget megvalósító automatikus tesztgenerálás mutációs teszteléssel szemben?

A létrehozott eszköz kiértékelését követően felhasználom az ipari esettanulmányban szereplő forráskód mutációs tesztelésére. Az eredményeket a tesztelés és a forráskód hiányosságainak, hibáinak felmérésére használom fel, majd megvizsgálom a felderített hibák és az alkalmazott operátorok kapcsolatát.

- **RQ4:** Milyen mutánsok, milyen környezetben jöttek létre, illetve maradtak életben?
- **RQ5:** Elkülöníthető-e hatékonyabb vagy kevésbé hatékony, redundáns operátorok?
- **RQ6:** Milyen jellegzetes **a)** tesztelési, **b)** forráskódbeli, illetve **c)** eszközbeli hibákra vezethetőek vissza a mutációs tesztelés eredményei?

4.1.2. Felhasznált forráskódok

Az elkészített eszköz, az optimalizáció és az automatikus tesztelés vizsgálatára a SIR (Software-artifact Infrastructure Repository) [32] adatbázisból származó *tcas* illetve *space* kódokat, és a hozzájuk tartozó tesztkészleteket használtam fel (lásd 3. táblázat).

Név	LOC	Tesztesetek száma
tcas	137	1608
space	5905	13585

3. táblázat – felhasznált SIR programok, kódsorok és elérhető tesztesetek száma

Az ipari esettanulmányban vizsgált forráskód 15 komponensből áll, melyek egyenként 1-7, különböző hosszúságú fájlból állnak. Minden fájlhoz tartozik legalább 1 tesztkészlet, 1 tesztkészlethez pedig több fájl is tartozhatna, azonban a vizsgált kódban ilyen nem fordult elő (lásd 4. táblázat).

Név	Fájlok	LOC	Tesztkészletek	Tesztesetek
comp1	2	408/99	8	1/10/42/2/1/1/5/1
comp2	1	100	2	13/1
comp3	1	324	1	42
comp4	4	57/541/271/476	4	10/70/36/54
comp5	5	77/91/17/23/42	4	9/14/3/0/7
comp6	7	22/10/10/21/5/15/88	9	3/1/1/1/2/1/3/1/9
comp7	4	362/12/10/262	5	5/44/2/1/17
comp8	1	402	2	44/1
comp9	6	110/402/15/480/187/151/74	6	15/29/3/21/16/3
comp10	1	40	1	5
comp11	1	136	1	18
comp12	1	558	8	10/1/5/11/1/1/6/10
comp13	1	254	2	38/1
comp14	2	738/334	1	78
comp15	1	27	1	4

4. táblázat – az ipari kód komponensei, az egyes komponensekben található fájlok száma, azok hossza, a komponensekhez tartozó tesztkészletek és tesztek száma

4.1.3. A kiértékelés menete

Az elkészített mutációs tesztelő eszköz kiértékeléséhez először a *tcas* kódjának mutációs tesztelését végzem el minden operátor felhasználásával. A kód rövidségéből következő kisszámú mutánst kihasználva minden rendelkezésre álló tesztbementre kapott kimenetet ellenőrzök. Mivel nem áll rendelkezésemre információ a „helyes" kimenetre vonatkozólag, a tesztek kiértékeléséhez a mutáns által adott kimenetet hasonlítom össze az eredeti kód által generált kimenettel. Hasonlóképpen végzem el a *space* mutációs tesztelését is, azonban a mutánsok nagy száma miatt nem használom fel az összes rendelkezésre álló tesztet, csak az egyik, kódlefedettséget megvalósító, 152 tesztből álló tesztkészletet használom fel. Mindkét mérést megismétlem az optimalizációs opció bekapcsolásával is. A mutánsok Cantata-s tesztelésétől eltérő ellenőrzése miatt ezekhez a mérésekhez szükségessé vált a létrehozott eszköz mutánsok tesztelését végző forráskódjának kis mértékű átalakítása.

Ezen kísérletek eredményeit Andrews, Briand és Labiche mutációs tesztelés hibadetektálásban való alkalmazhatóságát vizsgáló tanulmányának [33] mérési eredményeivel hasonlítom össze. Mérésükben az általam fejlesztett eszközhöz hasonló, de bővebb operátorkészletet alkalmaztak 8 SIR-ból származó forráskódon, melyeken egyenként 5000 különböző, 100 véletlenszerűen kiválasztott tesztesetből álló tesztkészlettel végeztek mutációs tesztelést.

A tesztszoftver automatikus tesztelési funkciójának vizsgálatához mind a *tcas*, mind a *space* kódjaihoz Cantata projektet hoztam létre. Ezt követően a vizsgált ipari környezetben előforduló legszigorúbb, SIL 4-es, teljes feltétellefedettséget (condition coverage) megkövetelve elindítom a tesztgenerálást. A *tcas* kódjának bonyolultsága miatt nem készült a lefedettséget kielégítő tesztkészlet, így azt a továbbiakban nem tudtam vizsgálni. A *space* kódjának egy részéről készült a szabványnak megfelelő lefedettséget megvalósító, 92 tesztesetből álló tesztkészlet, azonban ebben az esetben a használt számítógép hardveres korlátjai miatt a Cantata nem tudta feldolgozni a teljes forráskódot, csak annak első 1158 sorát (kb. a teljes kód 20%-a). Így nem a kód egészén végeztem el a mutációs tesztelést, hanem az adott kódszakasz mutációját lehetővé tevő opció felhasználásával csak annak első 1158 során.

Az ipari kód vizsgálata során annak minden komponensén lefuttattam az eszközt minden rendelkezésre álló operátor felhasználásával. A rendelkezésre álló tesztek automatikus generálással, majd a kódlefedettség és követelmények teljesítéséhez kézi kiegészítéssel készültek Cantata-ban.

A különböző elrendezések futtatásához a konfigurációs fájl megfelelő beállítását követően parancssorból elindítottam a futtatáshoz szükséges scriptet. Esetenként előfordult, hogy bizonyos mutánsok hatására futás közben végtelen ciklusba került a program. Ennek áthidalására létrehoztam egy visszaállító scriptet, mely tetszőleges sorszámú mutánstól a config fájlnak megfelelően folytatja a mutánsok ellenőrzését és a log írását, azonban a futási idő mérését értelemszerűen nem tudta megvalósítani. Egy-egy lefutást követően a CSV formátumú logot Excel táblázatba importáltam, hogy elvégezzem a fu-

tások kiértékelését. Minden futtatást egyszer végeztem el, a futási idők összehasonlíthatósága érdekében ugyanazon számítógépen, melyet a mérések során másra nem használtam. A használt számítógép Microsoft Windows 10 Enterprise 10.0 19042-t futtat, Intel® Core™ i7-6700, 3.40GHz-es négymagos processzorral és 16 GB RAM-mal rendelkezik.

4.2. Laboratóriumi környezetben

A 4.1.3-es pontban részletezett irodalmi benchmark kódokat használó mérések elvégzését követően a 4.2.1-es pontban kiértékelem a létrehozott eszköz működését, 4.2.2-es pontban megvizsgálom az alkalmazott optimalizáció hatásait, végül a 4.2.3-as pontban ugyanazon irodalmi forrásokon vizsgálom az automatikusan generált tesztek mutációs teszteléssel szembeni viselkedését.

4.2.1. RQ1: A létrehozott eszköz értékelése

A kiértékelés első lépeseként a mutációs tesztelő eszközömet a *tcas* forráskódján és annak teljes tesztkészletén futtattam, mely az alábbi táblázatban részletezett eredményeket produkálta (5. táblázat). A viszonyítási alapul vett vizsgálat az általam létrehozott eszköztől részben eltérő, illetve bővebb mutációs operátorok felhasználásával ennél jóval nagyobb számú, 291 mutánst generált, majd azokat többszörösen ellenőrizte a 100 véletlenszerűen kiválasztott tesztesetből álló tesztkészletek segítségével. Ezen futtatások pontszámai 77% és 97% szórtak, 91%-os átlaggal, ezen eredményt egyértelműen befolyásolta a kiválasztott tesztesetek adott mutánsokkal szembeni hatékonysága. A saját eszközöm által eredményezett, átlagnál alacsonyabb pontszám több élő mutánst jelent a teljes tesztkészlettel szemben, de nem tűnik ki az irodalmi mérésekből. A feltételvizsgálatok mutációjánál megjelenő különösen alacsony teljesítmény egyrészt a nagy mérete ellenére feltételvizsgálatok tesztelésében igen alacsony hatékonyságú tesztkészletre utal, másrészt a létrehozott eszköz hatékonyságára világít rá ezzel a tesztelési hibával szemben.

	Összes mutáns	**Halott**	**Élő**	**Mutációs pontszám**
Return NULL	7	7	0	100,0%
Conditionals negation	15	11	4	73,3%
Mathematical	18	18	0	100,0%
Conditionals boundary	13	6	7	46,2%
Remove condition	21	21	0	100,0%
ALL	**74**	**63**	**11**	**84,15%**

5. táblázat – Futási eredmények *tcas* kódon és tesztjein

Második lépésként a *space* forráskódján és annak egyetlen, kódlefedettség elérésére célzott, 152 tesztesetből álló tesztkészletén futtattam az általam létrehozott eszközt, melynek eredményeképp az alább látható eredmények születtek (6. táblázat). Az irodalmi mérésben a mutációs pontszám a 11379 létrehozott mutánson és 100 véletlenszerűen kiválasztott tesztesetből álló tesztkészletek esetén 65% és 82% között szórt, 75%-os átlaggal, mely igen közel áll a saját eszközöm eredményeihez. Ezen mérés során már figyelembe kellett vennem az ekvivalens mutánsok létrejöttének problémáját a kapott eredmények

korrekciójához: a létrejött *Return NULL* mutánsok számából levontam a „return 0” megjelenéseinek számát a kódban. Kiemelendő még a *Conditionals boundary* operátor alkalmazására kapott feltűnően alacsony pontszám, mely tükrözi, hogy a kódlefedettséget célzó tesztelés gyakran nem fedi le megfelelően a feltételvizsgálatokat.

Operátor	Összes mutáns	Halott	Élő	Mutációs pontszám
Return NULL	393	207	186	52,7%
Conditionals negation	596	486	110	81,5%
Mathematical	285	171	114	60,0%
Conditionals boundary	129	25	104	19,4%
Remove condition	1398	1128	270	80,7%
Increment invert	33	30	3	90,9%
ALL	**2834**	**2047**	**787**	**72,23%**
Return NULL*	254	207	47	81,5%
Korrigált ALL	**2695**	**2047**	**648**	**75,96%**

6. táblázat – Futási eredmények *space* kódon és egy tesztkészletén, illetve makrókból eredő ekvivalens mutánsok korrigálását követően

RQ1: Az elkészített eszköz szűkebb operátorkészletéből kifolyólag kevesebb mutánst hoz létre, de a tesztelés során előállt pontszám hasonló a vizsgált irodalmi adatokhoz.

4.2.2. RQ2: Az optimalizáció hatásai

Vizsgálatom következő lépéseként a mutációs tesztelő eszközöm soronként egyetlen mutánst létrehozó optimalizációs funkcióját teszteltem és mértem fel annak hatásosságát. A *tcas* kódján futtatva az eredeti mutánsok 32%-a jött létre, a futási idő a teljes futtatás idejének 38%-ára esett vissza. A teljes futtatásban a kód minden sorára volt halott mutáns, az optimalizációs méréssel ez 92%-os egyezést mutatott. A kapott mutációs pontszám jóval magasabb, 92% volt. A *space* kódján futtatva az eredeti mutánsok 39%-a jött létre, a futási idő viszont a mindkét esetben előforduló végtelen ciklusba esések miatt nem volt mérhető. A teljes futtatás sorainak bukása 99,55%-os egyezést mutatott az optimalizáció eredményeivel, a mutációs pontszám szintén valamivel magasabb, 79,88% volt.

Az eredmények alapján az optimalizáció valamivel magasabb pontszámot eredményez, de kódsoronként igen hasonló eredményeket hoz. Jelentős mértékben lecsökkenti a létrehozott mutánsok számát és ezzel a futtatás idejét, azonban esetenként egyes operátorokat egyáltalán nem használ fel és így bizonyos hibákra semmilyen esetben sem világít rá. A mérés céljától függően így egyes esetekben indokolt lehet a használata, például ha csak átfogó képet szeretnénk kapni a tesztjeink minőségéről, vagy azok hiányosságainak kódbeli pozíciójáról, a teljes kiértékelés idejének töredéke alatt. Megfontolandó még a keresett hibák függvényében bizonyos operátorok kikapcsolása, hatékonyabbnak bizonyuló operátorok előnyben részesítése.

RQ2: Az optimalizáció a vizsgált esetekben valamivel magasabb mutációs pontszám eléréséhez vezetett, azonban kódsoronként igen nagy egyezést mutatott a teljes futtatás eredményeivel.

4.2.3. RQ3: Az automatizált tesztelés értékelése

Az irodalmi benchmark kódokat használó végső mérésemben az ipari esettanulmányban használt tesztszoftver automatikus tesztelési funkcióját vizsgáltam. A 4.1.3-as pontban leírt okokból ez a *space* forráskódjának egy részét lefedő mutációs tesztelést jelent, mely a lentebb látható eredményekkel zárult (7. táblázat).

Operátor	Összes mutáns	Halott	Élő	Mutációs pontszám
Increment invert	0	0	0	
Mathematical	23	9	14	39,13%
Conditionals boundary	2	2	0	100.00%
Conditionals negation	26	20	6	76,92%
Boolean invert	0	0	0	
Return NULL	61	38	23	62.30%
Remove condition	109	78	31	71,56%
ALL	**221**	**147**	**74**	**66,52%**
Return NULL*	38	38	0	100,00%
Korrigált ALL	**183**	**147**	**36**	**80,33%**

7. táblázat – *space*-ből származó kódrészlet és Cantata teljes feltétellefedettséget megvalósító automatikus tesztek mutációs tesztelésének eredményei

Látható, hogy a matematikai operátorok, illetve a feltételvizsgálatok mutációja gyengébb pontszámot eredményezett, ez feltehetőleg ciklusok, illetve függvények passzív hatásának nem megfelelő tesztelésére vezethető vissza. Ellenben a visszatérési értékek és az összehasonlítási határok mutációja minden esetben megbukott, mutatva, hogy mind a visszatérési értékek vizsgálata, mind a határértékekkel való tesztelés megfelelően megvalósított az automatikus tesztekben is.

RQ3: A teljes kódlefedettség automatikus tesztekkel való biztosítása általánosságban jól teljesít a tesztelt függvények aktív hatásait mutáló operátorokkal szemben, míg a passzív hatásokat érintő mutánsok tesztelése kifejezetten gyenge eredményeket hoz.

Említendő még, hogy ezen kísérlet során a létrehozott mutánsok kb. harmada futási hibát okozott, mivel a Cantata kódlefedettség mérése hibába futott. Ez minden esetben a *Remove condition* operátorcsoport alkalmazása során történt. Konkrétan ezen operátorok használatát a 2.4-es pontban vizsgált mutációs eszközök közül a dextool kivezette, nagy mennyiségű „szemét” képzésére hivatkozva. Mivel az operátorok egyetlen helyre 3 mutánst generálnak, az elkészült mutánsok nagy részét teszik ki és elnyomhatják más operátorok mutációs pontszámra tett hatását, miközben nagymértékben növelik a futási időt.

Emellett azokon a helyeken, ahol ilyen mutáns marad életben, az esetek nagy többségében valamilyen összehasonlító művelet áll, melynek más operátorok alkalmazásával létrejött mutációja szintén életben marad. Megfontolandó lehet az operátor kihagyása.

4.3. Ipari környezetben

Az irodalmi benchmark kódokat felhasználó vizsgálatokat követően áttértem az elkészített eszköz ipari alkalmazására. A következő pontokban ennek eredményeit és a levont következtetéseket tárgyalom. A futtatás során 3281 mutáns jött létre, ebből 2871 futott le, a kapott pontszámok és a típusaik eloszlása lentebb láthatóak (8. táblázat). A részletes futtatási eredmények a függelék részét képző táblázatokban találhatóak.

Operátor	**Mutációs pontszám**	**Mutánsok száma**	**aránya**
Increment invert	57,53%	91	3,17%
Mathematical	66,16%	351	12,23%
Conditionals boundary	40,33%	132	4,60%
Conditionals negation	89,81%	517	18,01%
Boolean invert	91,48%	281	9,79%
Return NULL	65,13%	299	10,41%
Remove condition	88,78%	1200	41,80%
ALL	**80,79%**	**2871**	

8. táblázat – az ipari komponenseken és tesztjeiken történő futtatás eredményei

4.3.1. RQ4: Tipikus életben maradt mutánsok vizsgálata

A teljes ipari kódkészleten való mutációs tesztelést követően manuálisan megvizsgáltam minden életben maradt mutáns forráskódját és az életben maradása mögött álló lehetséges tesztelési hibát. Világossá vált, hogy az ugyanazon operátorok használatával elkészített mutánsok életben maradása mögött általában ugyanazok a jellegzetes hibák álltak.

Increment invert operátorcsoport alkalmazásából származó élő mutánsok az esetek döntő többségében for ciklusok léptetései voltak. Ezen ciklusok gyakran töltenek fel tömböket vagy léptetnek pointereket, így akkor buktathatnának tesztesetet, ha azok a pointerek címét vagy az általuk mutatott értéket is ellenőriznék, nem csak a függvény lefutását.

Mathematical operátorcsoport által generált mutánsok több jellegzetes helyzetben is életben maradtak. Egyfelől && vagy || műveleteknél fordultak elő, ami a több logikai kifejezés kapcsolatából álló feltételvizsgálatok nem megfelelő tesztlefedettségére utal.

Másfelől életben maradtak függvényhívások paramétereibe ágyazott matematikai műveletek esetén is, ami a függvényhívások paramétereinek ellenőrzésének hiányára utal. Emellett bizonyos, a program által végzett számítások mutációi is életben maradtak, ez a tényleges számértékek hiányos ellenőrzésére mutat rá.

Conditionals boundary típusú mutációk mindig határértékekkel való tesztelés elmaradásának köszönhetően maradtak életben. Látható, hogy messze ez a mutáció eredményezte a legalacsonyabb pontszámot, ami annak köszönhető, hogy bár triviális helyeken (if-szerkezetek) ez a típusú tesztelés nem maradt el, de mind for, mind while ciklusok feltételvizsgálatainak ilyen jellegű mutációja szinte mindig életben maradt.

Conditionals negation alkalmazása esetén szintén ciklusok feltételvizsgálatainak mutált verziói maradtak életben, a fentiekhez hasonló okokból kifolyólag. Emellett jellemző volt még else-ággal nem rendelkező if-szerkezetek feltételeiben is, mely a kódlefedettség teljesülése mellett az elágazás nem megfelelő tesztlefedettségének egyértelmű indikátora.

Boolean invert operátorcsoport által képzett mutációk jellemzően akkor maradtak életben, ha valamilyen logikai változó kezdeti értékadását képezték, melyet a program később mindenképpen felülírt. Ezek a mutánsok tehát inkább a kódban található felesleges értékadásoknak köszönhetően maradtak életben.

A *Return NULL* operátor alkalmazása előre tudhatóan nagy mennyiségű ekvivalens mutánst generál, a vizsgált életben maradt mutánsok szinte minden esetben valamilyen 0-ra kiértékelődő makró helyén képződtek. Azokban az esetekben, ahol ez mégsem volt így, vagy olyan változó volt a visszatérési érték, mely minden esetben 0 értékű, vagy valamilyen függvényhívás, melynek a lehetséges visszatérési értékeit nem fedték le tesztesetek.

Remove condition mutáció alkalmazásakor a Cantata kódlefedettségi mérések gyakran hibába futottak, így nem minden mutáns viselkedését lehetett megvizsgálni. A vizsgált esetekben azonban megállapítható, hogy ezek a mutánsok olyan helyeken maradtak életben, ahol maga a feltételvizsgálat valamely mutációja egyébként is életben maradt, azonban a háromféle mutáns életben maradásának kombinációjából lehetett következtetni a tesztelési hiba jellegére. Például, ha csak az if (TRUE)-ra mutált kód maradt életben, az egyértelműen az elágazás nem megfelelő lefedettségének jele, míg ha mindhárom mutáns életben maradt valószínűsíthető, hogy az adott kódrészlet hatását semmilyen teszteset nem ellenőrzi, vagy maga a kód nincs hatással a program lefutásának kimenetelére.

RQ4: Minden operátorcsoport mutánsai néhány jellegzetes helyzetben, jellegzetes hibákra utalva maradtak életben:

- *Increment invert*: for ciklusok léptetése
- *Mathematical*: logikai kifejezések kapcsolata, függvényhívások paraméterei, számítások
- *Conditionals boundary*: ciklusok feltételei
- *Conditionals negation*: ciklusok feltételei, else-ág nélküli if-szerkezetek
- *Boolean invert*: felesleges változóinicializációk
- *Return NULL*: függvényhívás visszatérési értékként, ekvivalens mutánsok
- *Remove condition*: hiányosan tesztelt if-szerkezetek

4.3.2. RQ5: Operátorok hatékonysága

A különböző létrejött mutánsok számát és a hozzájuk tartozó pontszámokat vizsgálva egyértelműen látható, hogy messze a legalacsonyabb pontszám a *Conditionals boundary* operátorcsoport mutánsaihoz tartozik, azok igen kis száma mellett. Ennek ellenére hiba lenne hatékonyabbként kiemelni a többi mutáns közül, mert csak bizonyos típusú hibák felderítésére alkalmas.

A nagy számú ekvivalens mutáns miatt megfontolandó a *Return NULL* operátor használata, vagy annak átalakítása. Preprocesszált kódon történő alkalmazás esetén nem képezne ekvivalens mutánsokat, így csökkentve a felesleges futtatásokat.

Egyértelműen redundáns, instabil és emellett messze a legtöbb mutánst képző operátor a *Remove condition*. Minden esetben más operátorok által is felderített hibákra világít rá, csak a tesztelési hiányosság jellegéről ad újabb információt. Emiatt a továbbiakban csak akkor ajánlott az alkalmazása, ha megfelelő számítási kapacitás áll rendelkezésünkre vagy szükségünk van az általa biztosított további információra.

RQ5: Kiemelkedően hatékony operátor elkülönítése nem indokolt, a használt operátorok közül mindegyik más típusú tesztelési hibát azonosít sikeresen. Az ekvivalens mutációk számának csökkentése érdekében korlátozható a *Return NULL* operátor használata, míg a *Remove condition* operátorcsoport alkalmazása redundáns és igen labilis, ezért erősen megfontolandó.

4.3.3. RQ6: Jellegzetes hibák

Az elvégzett mérések tapasztalatait felhasználva elkülöníthetőek a tesztelésben, a forráskódban, illetve az elkészített eszközben megjelenő hibák, hiányosságok.

A tesztelésben általánosságban előforduló és többféle operátor alkalmazásából származó mutáns életben maradását okozó hiba, hogy *a tesztelt függvény passzív hatásai nincsenek megfelelően ellenőrizve*. A Cantata eszközkészlete lehetővé teszi, hogy csak a függvényben létező változókat is ellenőrizzünk, emellett pedig triviálisan is ellenőrizhető

egy-egy pointer címe vagy az általa mutatott érték, így ezek a hibák könnyen javíthatóak. Szintén gyakori hiba – mely a tesztszoftver által biztosított eszközökkel könnyen javítható – *a meghívott függvények paraméterezésének ellenőrzése*, illetve *az összes lehetséges visszatérési érték biztosítása* a tesztek során. Emellett ritkábban előforduló hiba volt még függvényhívások ellenőrzésének teljes elmaradása. Nehezebben megoldható problémát okoz a különböző *ciklusok megfelelő tesztelése*. Az *iterációk száma* egyértelműen nincs tesztelve, de ennek közvetlen ellenőrzése nem indokolt és nem is valósítható meg minden esetben. Az iterációk számának hatása azonban sokkal fontosabb a program működése szempontjából és emiatt mindenképp tesztelendő. Ezek már szintén az adott függvény passzív hatásai közé sorolható és a fent említett módszerekkel végrehajtható.

Néhány mutáns életben maradása nem a tesztek, hanem a forráskód hiányosságaira vagy logikailag helyes, de nem szabványos megoldásaira vezethető vissza. Ilyenek például az olyan értékadások, amelyeket azok felhasználása nélkül más értékadások követnek, így semmilyen hatással nincsenek a program futására. Hasonlóképpen nem okoz hibás működést, de a tesztelést, a kódlefedettség mérését és a kód fenntarthatóságát nehezítik a visszatérési értékként meghívott, illetve a makróban definiált függvények.

A mérések során előforduló futási hibákból fény derült a létrehozott eszköz néhány bővítési lehetőségére is. Többszörösen előforduló probléma volt a tesztelt program végtelen ciklusba kerülése, melyre részleges megoldást jelent az elkészített visszaállító script. Előfordult még NULL pointer címzését megakadályozó kódrészlet mutációja, ami futási hibát eredményez. A nem preprocesszált kód mutációjának velejárója a nagyobb számú ekvivalens mutáns keletkezése, de az feltételes fordítások részét képző inaktív kódrészletek mutációja is.

RQ6: a) A legjellemzőbb tesztelési hibák a tesztelt függvény passzív hatásainak elmaradó vizsgálata, a függvényhívások hiányos ellenőrzése, illetve a ciklusok és iterációik megfelelő tesztelésének elmaradása.
b) A forráskódban a felesleges értékadások és a szabványostól eltérő kódolás eredményezte némely mutáns életben maradását.
c) A létrehozott eszköznek a legnagyobb nehézséget a végtelen ciklusok, illetve a preprocesszor direktívák jelentették.

4.4. Eredmények összegzése és értékelése

A 4.4.1-es pontban a 4.2.1-es pontban bemutatott mérési eredményeket az irodalmi adatokkal összevetve értékelem a létrehozott eszköz teljesítményét, a 4.2.2-es pont eredményeit összehasonlítom az optimalizációra vonatkozó irodalmi eredményekkel, illetve megvizsgálom az irodalmi álláspont és a 4.3.2-es pontban leírt operátorok hatékonyságára vonatkozó tapasztalatok közti kapcsolatot. A 4.4.2-es pontban a generált tesztek mutációs tesztelésének 4.2.3-as pontban bemutatott eredményeit értékelem. Végül a 4.3-as pont tapasztalatait felhasználva a 4.4.3-as pontban javaslatot teszek a mutációs tesztelés továbbiakban történő alkalmazási módszereire és stratégiáira.

4.4.1. A saját eredmények összehasonlítása az irodalommal

A 4.2.1-es pontban bemutatott mérési eredményekből megállapítható, hogy a saját eszközöm használatával a hasonlítási alapul vett tanulmányhoz képest jelentősen kevesebb mutánst létrehozva kisebb kódon kismértékben eltérő, nagyobb kódon már közel azonos pontszámok születnek. A pontosabb összehasonlítás érdekében szükséges lenne nem csak a pontszám, hanem az életben maradt mutánsok által kimutatott tesztelési hiányosságok vizsgálata, azonban a tesztekre vonatkozó részletes információ hiányában erre nem volt lehetőségem. Emellett a tanulmány több, kisebb tesztkészleten végzett mérést összesít, ellentétben az általam végzett mérésekkel, amik egyetlen futtásból származnak. Ezért az eredményeim későbbi pontosítására lehetőség a tesztek vizsgálata, illetve egy hasonlóan mintavételezett tesztkészleteken elvégzett mérés. A megvalósított mutációs tesztelés hasonló mérési elrendezés hiányában nem tökéletesen összehasonlítható a vizsgált tanulmánnyal, de a valamivel szűkebb, hasonló operátorkészlettel elért hasonló pontszámok hasonló hatékonyságot engednek sejtetni.

A 4.2.2-es pontban mutattam be a Google-nél is alkalmazott optimalizációs eljárás saját eszközömmel történő végrehajtását és annak eredményeit. A kapott eredmények nem térnek el nagy mértékben, ám valamivel jobbak a Google szakemberei által mért értékeknél, így alátámasztják a soronként egy mutáns létrehozásának optimalizációban való használatának lehetőségét. A méréseim azonban csak igen kis mintáról származnak és véletlenszerűen kiválasztott operátorokat alkalmaztak, így inkább kísérleti és tesztelési célra szolgálnak, mintsem tényleges alátámasztásai az eredeti mérésnek.

A 4.3.2-es pontban megállapítottam, hogy a *Remove conditions* operátorcsoport használata alacsony hatékonysága és nagy számításigénye miatt erősen megfontolandó. Emellett problematikus volt a *Return NULL* operátor alkalmazása is, bár preprocesszált kódon alkalmazva a problémák kiküszöbölhetővé váltak. Az összes többi használt operátor hatékonynak bizonyult és más-más jellegű hibákra világított rá. Ezek az eredmények hasonlóságot mutatnak az 5 „hatékony” operátorcsoportot kiemelő irodalmi állásponttal, mivel az ezekbe a kategóriákba sorolható operátorok az általam elvégzett mérésekben is hatékonynak bizonyultak, míg az ide nem tartozóak használata nem feltétlenül gazdaságos.

4.4.2. Generált teszteken elért mutációs eredmények

A 4.2.3-as pontban mutattam be az automatikusan generált tesztek mutációs teszteléssel szembeni teljesítményét. Összességében megállapítható, hogy a legszigorúbb kódlefedettség sem biztosította a kód teljes funkcionalitásának ellenőrzöttségét, különös tekintettel egy-egy függvény passzív hatásaira. Ezek a hatások általában követelményekként mindenképpen tesztelendő részei a kódnak, ezért fontos a kódlefedettséget biztosító tesztek mellett a komponensszintű követelményeket ellenőrző tesztesetek készítése is.

4.4.3. Futtatási és konfigurálási stratégiák

Az operátorok hatékonyságára tett megállapításokból következik, hogy a *Remove conditions* operátor használata főleg nagyobb terjedelmű forráskód esetén nem hatékony és igen időigényes. A *Return NULL* operátor jól használható preprocesszált forráskódon,

ami egyben a makrókban definiált függvények vizsgálatára és a feltételes fordítások által inaktívvá váló kód problémájának kiküszöbölésére is alkalmas módszer, de az iparban gyakran előforduló nagy mennyiségű header fájl miatt túlságosan bonyolulttá és olvashatatlanná teheti a forráskódot.

A fejlesztési és tesztelési folyamatba történő integráció későbbi projektek során az alábbi módon lenne kivitelezhető:

1. Az elkészült kódhoz elkészül a tesztkészlet(ek) első verziója, melyen a forráskód maradéktalanul átmegy és eléri az adott biztonságintegritási szinthez szükséges teljes lefedettséget is.
2. Elvégezzük a teljes kód és tesztjeinek mutációs tesztelését.
3. Megvizsgáljuk az életben maradt mutánsokat.
4. Azokban az esetekben, ahol ezt indokoltnak tartjuk kiegészítjük az elkészült tesztkészlet(ek)et az élő mutánsok megölésére alkalmas tesztesetekkel.
5. Elvégezzük az érintett kódrészlet mutációs tesztelését.
6. A 3-5. lépéseket mindaddig ismételjük, amíg az összes (megölni kívánt) mutánst meg nem öltük.
7. Ellenőrzésképp ismét elvégezzük a teljes kód mutációs tesztelését.

5. Összefoglalás

A mutációs tesztelés elve már közel 50 éves múltra tekint vissza, ez idő alatt számtalan mérés, vizsgálat és kutatás tárgya volt. Ezen kutatások kezdetben az alkalmazás módszertanára irányultak, majd optimalizációs módszerekre létrehozását kutatták, a technológia – a számítási kapacitás és a programozási nyelvek – fejlődésével elindultak a gyakorlati alkalmazások felé irányuló kísérletek is. Az elmúlt évtizedben megindult a módszer ipari térnyerése is, azonban a módszer biztonságkritikus beágyazott rendszerek tesztelésében történő alkalmazásáról kevés eredmény áll rendelkezésre.

Munkámban létrehoztam egy olyan, mutációs tesztelést megvalósító eszközt, amely képes a vizsgált ipari esettanulmányban használt méretű és komplexitású C kód és a tesztelésére használt szoftver kezelésére. Az eszközt irodalmi benchmark kódokon futtatva, illetve funkcionalitását más eszközökével összehasonlítva megállapítottam, hogy annak teljesítménye összemérhető más hasonló eszközök teljesítményével, a fejlesztés során az ipari környezetben való működés érdekében hozott kompromisszumok nem befolyásolták negatívan az eszköz hatékonyságát.

A létrehozott eszközt először a kódlefedettséget biztosító, automatikusan generált tesztek minősítésére használtam fel, ezzel megmutatva, hogy a kód legszigorúbb szabályok szerinti lefedése sem biztosítja a működés teljes ellenőrzöttségét, és a funkcionalitást ellenőrző tesztek elkészítése mindenképpen szükséges. Ezt követően az ipari esettanulmányban szereplő forráskódon és annak automatikusan generált, majd manuálisan kiegészített tesztkészletén végeztem mutációs tesztelést. A kapott eredmények kiértékelését követően bemutattam a tanulmányban megjelenő jellegzetes tesztelési és forráskódbeli hibákat és javaslatot tettem a kijavításukra. Összehasonlítottam a különböző operátorokkal kapcsolatos tapasztalataimat az irodalomban szereplő adatokkal és a tapasztalatok felhasználásával javaslatot tettem az eszköz tesztelési folyamatokba való integrációjának optimális módjára. A kapott kísérleti eredmények jó kiegészítői az irodalomban elérhető mérési eredményeknek, és egy újabb, speciális környezetből származó ipari esettanulmány tapasztalataival bővítik a mutációs tesztelésről elérhető információkat.

A módszer és a létrehozott eszköz munkafolyamatokba történő integrációja előreláthatóan fejleszti a tesztelői és fejlesztői szemléletet, ezzel javul majd mind a forráskód, mind a hozzátartozó tesztek minősége. Az idővel felgyülemlő tapasztalatok pedig felhasználhatóak az eszköz funkcióinak és működésének továbbfejlesztésére, finomítására, ezzel az adott ipari környezetbe és folyamatokba beleilleszkedő módszert és eszközt létrehozva.

Irodalomjegyzék

[1] R. Ramler, T. Wetzlmaier és C. Klammer, „An empirical study on the application of mutation testing for a safety-critical industrial software system," in *Proceedings of the Symposium on Applied Computing*, Marrakech, 2017.

[2] R. Gopinath, C. Jensen és A. Groce, „Code coverage for suite evaluation by developers," in *36th International Conference on Software Engineering (ICSE 2014)*, New York, 2014.

[3] L. Inozemtsva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, New York, Egyesült Államok, 2014.

[4] P. S. Kochhar, F. Thung és D. Lo, „Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Montreal, 2015.

[5] IEC, "IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems". Patent IEC 61508, 04 2010.

[6] Z. Huang és R. Alexander, „Semantic Mutation Testing for Multi-agent Systems," in *Engineering Multi-Agent Systems*, Springer, Cham, 2015.

[7] T. A. Budd és F. Sayward, „Users guide to the Pilot mutation system," New Haven, Connecticut, 1977.

[8] R. G. Hamlet, „Testing programs with the aid of a compiler," *IEEE transactions on software engineering*, %1. kötet4, pp. 279-290, 1977.

[9] R. A. DeMillo, R. J. Lipton és F. G. Sayward, „Hints on test data selection: Help for the practicing programmer," *Computer*, %1. kötet11, %1. szám4, pp. 34-41, 1978.

[10] T. A. Budd, Mutation analysis of program test data, Yale University, 1980.

[11] A. J. Offutt, G. Rothermel és C. Zapf, „An experimental evaluation of selective mutation," in *Proceedings of 1993 15th international conference on software engineering*, Baltimore, 1993.

[12] A. J. Offut, „An experimental determination of sufficient mutation operators," *ACM Transactions on Software Engineering and Methodology,* %1. kötet5, %1. szám2, pp. 99-118, 1996.

[13] M. Sahinoglu és E. H. Spafford, „A sequential statistical procedure in mutation-based testing," in *Proceedings of the 28th Annual Spring Reliability Seminar,* 1990.

[14] A. Derezinska és M. Rudnik, „Evaluation of mutant sampling criteria in object-oriented mutation testing," in *2017 Federated Conference on Computer Science and Information Systems (FedCSIS),* 2017.

[15] G. Petrović, M. Ivanković, G. Fraser és R. Just, „Does Mutation Testing Improve Testing Practices?," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE),* Madrid, 2021.

[16] G. Petrović, M. Ivanković, G. Fraser és R. Just, „Practical Mutation Testing at Scale: A view from Google," *IEEE Transactions on Software Engineering,* %1. kötet48, %1. szám11, pp. 3900-3912, 2022.

[17] J. Offut, „A mutation carol: Past, present and future," *Information and Software Technology,* %1. kötet53, %1. szám10, pp. 1098-1107, 2011.

[18] Y. Jia és M. Harman, „An Analysis and Survey of Development of Mutation Testing," *IEEE Trans. Software Eng.,* %1. kötet37, pp. 649-678, 2011.

[19] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon és M. Harman, „Mutation Testing Advances: An Analysis and Survey," *Advances in Computers,* 2018.

[20] J. Brannstorm, „dextool," 2014. [Online]. Available: https://github.com/joakim-brannstrom/dextool.

[21] A. Denisov, „mull," 2016. [Online]. Available: https://github.com/mull-project/mull.

[22] N. Lohmann, „Mutate++," 2017. [Online]. Available: https://github.com/nlohmann/mutate_cpp.

[23] M. A. G. Silva, „Proteum," 2012. [Online]. Available: https://github.com/magsilva/proteum.

[24] Y. Jia, „Milu," 2011. [Online]. Available: https://github.com/yuejia/Milu.

[25] J. Offut, „muJava,” 2015. [Online]. Available: https://github.com/jeffoffutt/muJava/tree/master/src/mujava.

[26] H. Coles, „PIT,” 2010. [Online]. Available: https://github.com/hcoles/pitest.

[27] K. Halas, „mutPy,” 2011. [Online]. Available: https://github.com/mutpy/mutpy.

[28] E. Bendersky, „pycparser,” 2008. [Online]. Available: https://github.com/eliben/pycparser.

[29] T. Parr, *ANTLR v4*, The ANTLR Project, 2012.

[30] S. Harwell, „ANTLR v4 C grammar,” 2013. [Online]. Available: https://github.com/antlr/grammars-v4/tree/master/c.

[31] M. Hafiz, Y.-S. Ma és J. J. Offut, *MUTATION TESTING TOOL FOR JAVA - Table 1*, 2005.

[32] H. Do, S. G. Elbaum és G. Rothermel, „Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact,” *Empirical Software Engineering: An International Journal*, %1. kötet10, %1. szám4, pp. 405-435, 2005.

[33] J. H. Andrews, L. C. Briand és Y. Labiche, „Is Mutation an Appropriate Tool for Testing Experiments?,” in *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, 2005.

Függelék

Itt elérhető függelék tartalmazza a 4. fejezet mérési eredményeit:

- **space_suite0_allmuts.csv, space_suite0_allmuts.xlsx:** *space* kódjának és egy tesztkészletének teljes mutációjából származó log, illetve annak kiértékelése (4.2.1)
- **space_suite0_opt.csv, space_suite0_opt.xlsx:** *space* kódjának és egy tesztkészletének optimalizált mutációjából származó log, illetve annak kiértékelése (4.2.2)
- **tcas_universe_allmuts.csv, tcas_ universe _allmuts.xlsx:** *tcas* kódjának és összes tesztjének teljes mutációjából származó log, illetve annak kiértékelése (4.2.1)
- **tcas_ universe _opt.csv, tcas_ universe _opt.xlsx**: *tcas* kódjának és összes tesztjének optimalizált mutációjából származó log, illetve annak kiértékelése (4.2.2)
- **RESULTS.xlsx:** az ipari esettanulmányból származó kód teljes mutációjának kiértékelése, összesítő táblázatban, illetve komponensekre lebontva külön munkalapokon (4.3), az utolsó munkalapon a *space*-ből származó kódrészlet és generált tesztjeinek mutációs tesztelése (4.2.3)