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

Vasúti demonstrátor automatizált tesztelésének kialakítása

Szakdolgozat

Készítette
Verbőczy Kristóf

Konzulens
Dr. Micskei Zoltán

2016. november 14.

Tartalomjegyzék

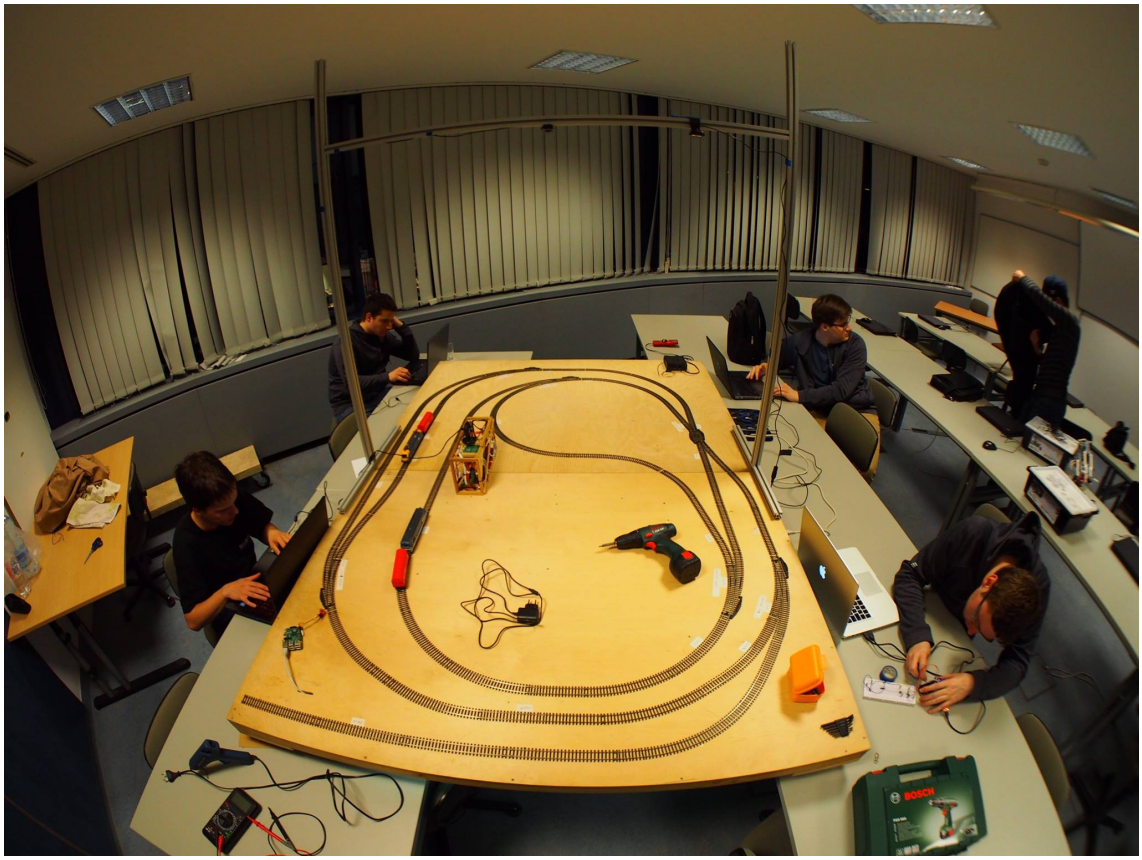
1. fejezet

Bevezetés

Vasúti demonstrátor automatizált tesztelésének kialakítása

1.1. MoDeS3

A MoDeS3 egy projekt elnevezése. A mozaikszó felbontása Model-based Demonstrator for Smart and Safe Systems, azaz modellalapú demonstrátor okos és biztonságos rendszerekhez. A projekt jelenlegi státuszában egy modellvasútból és egy robotból, melyeknek segítségével egy kritikus rendszerek tervezési, fejlesztési és ellenőrzési feladataihoz kapcsolódó módszereket és technológiákat lehet bemutatni.



1.1. ábra. a terepasztal

1.2. Terület

Ide mi lenne megfelelő?
biztonságkritikus rendszerek?
beágyazott rendszerek?

1.3. Cél

A szakdolgozat célja, egy olyan tesztelési módszer/keretrendszer kialakítása, amely felhasználásával meggyőződhetünk a MoDeS3 projekt komponensszintű biztonsági logikájának alapvető helyes működéséről és a projektbe való integrálhatóságáról. Ehhez szükség van különböző szinteken tesztelni a rendszert, aminek a hatékony kivitelezésére szolgál a módszer/keretrendszer.

1.4. Probléma

Az alapvető probléma az, hogy a rendszer architektúrája bonyolult. Több különböző hardver elem építi fel, továbbá ennél is több szoftver komponens. Ezen komponensek megértése időigényes, tesztelése körülményes. Még egy komponens (például a biztonsági logika) esetén is szükséges megérteni a többi működését, hiszen az együttműködés ellenőrzéséhez azokat is használni kell.

1.5. Feladat

A cél elérése érdekében a kiválasztott módszer a tesztelés. Tehát a feladatom az, hogy teszteljem a biztonsági logikát. Ahhoz, hogy megbizonyosodhassunk a rendszer biztonságos működéséről, ezt több szinten kell végrehajtani. Viszont fennállnak bizonyos problémák, amelyek bonyolultabbá teszik a több szintű tesztelés elvégzését. A feladat olyan módszer kitalálása és végrehajtása, amellyel ezen problémák mellett is hatékonyan lehet tesztelni a MoDeS3 projekt biztonsági logikáját különböző szinteken.

1.6. Főbb eredmények

Majd akkor, amikor már lesznek eredmények.

1.7. Dolgozat szerkezete

A dolgozat elején bemutatásra kerülnek a feladat elvégzéséhez szükséges háttérismeretek. Majd szó lesz kicsit részletesebben a tesztelés alatt álló rendszerről, azaz a MoDeS3-ról. Azután a tesztelés tervezése lesz olvasható. Ezt követi az implementáció, ami tulajdonképen a feladat végrehajtását írja le. A dolgozat végéhez közeledve szó lesz a tesztekről és a hozzájuk tartozó eredményekről, például mennyi hibát sikerült felfedni. Zárásképp az összefoglalás olvasható, amely tartalmazza a feladatom eredményét és a jövőbe tekint a továbbfejlesztési lehetőségekkel. Természetesen a felhasznált hivatkozások is össze lesznek gyűjtve.

2. fejezet

Háttérismeretek

A feladat elvégzéséhez szükség volt bizonyos ismeretekre, amelyek egy részével már képzés során különböző tárgyakban találkoztam, néhány pedig új volt, ezeket meg kellett ismernem. A háttérismeretek között van, ami elég általános, de van olyan is ami ennél specifikusabb, szorosabban köthető a projekthez.

2.1. V-modell

A V-modell egy életciklus modell a szoftverfejlesztésben. Felfogható a vízesés modell kibővítéseként. Ezt is a szekvenciális folyamat végrehajtás jellemzi, azaz minden fázist be kell fejezni mielőtt elkezdenénk a következőt. A vízesés modellhez képest a V-modellben az implementáció elkészítése utáni folyamatok nem lefele, hanem felfele helyezkednek el, ezzel kialakítva a jellegzetes V formát. Ez jól reprezentálja a fejlesztés adott fázisához tartozó tesztelési szintet. Ez látható a 2.1 ábrán, továbbá a fázisok összekapcsolódása.
A V-modell használata főleg a biztonságkritikus számítógéprendszerek fejlesztése esetében terjedt el. Alkalmazása kis és közepes méretű projektek esetén ajánlott. [11]

2.1.1. Fázisok

1. **Követelmények és specifikáció**
   Első lépésként felhasználói igények felmérése, dokumentumba foglalása történik ebben a fázisban. Ez a dokumentum lesz az alapja a végső validációnak. Ez alapján a mérnökök elkészítik a specifikációt. Továbbá dokumentációk készülnek a rendszertesztelés hez.

2. **Architekturális tervezés**
   A szoftver architektúrája készül el ebben a fázisban. Ide tartozik a modulok listája, a modulokhoz tartozó rövid leírás, interfészek leírásai, függőségek és architektúra diagramok.

3. **Részletes tervezés**
   A program specifikáció tartalmazza a modulok részletes logikáját, azaz a hozzá tartozó adatbázis táblákat, minden interfész részleteit, minden függőséget, hibaüzenetek listáját, a modul lehetséges ki- és bemeneteit.

4. **Implementáció**
   Ebben a fázisban készül el a programkód.

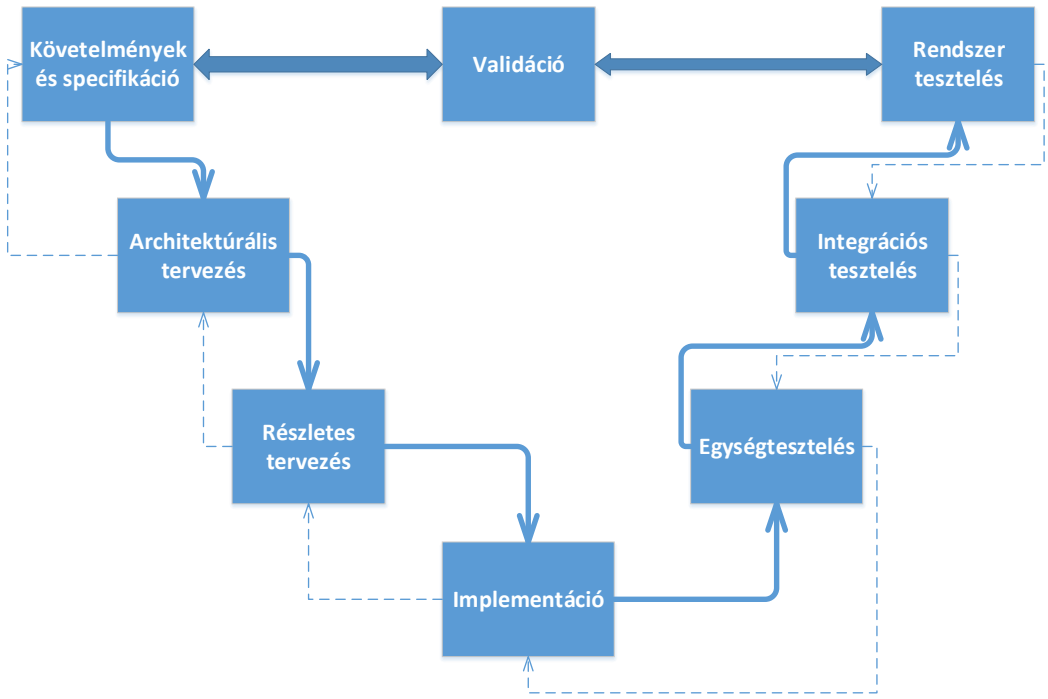
5. **Egységtesztelés**
   A részletes tervezés fázisában készülnek el az egységteszt tervek. Ezek a tesztek kerül-

nek végrehajtásra, hogy meggyőződjünk róla, hogy a rendszer legkisebb építőelemei helyesen funkcionálnak.

6. **Integrációs tesztelés**
   Az integrációs teszt tervek az architekturális tervezés fázisában készülnek. Ezek a tesztek azt hivatottak igazolni, hogy az egymástól függetlenül helyesen működő egységek képesek-e együttműködni, vagy kommunikálni.

7. **Rendszer tesztelés**
   Az egész alkalmazás tesztelésre kerül. Mind a funkcionális, mind a nem-funkcionális követelmények ellenőrzésre kerülnek. Ide tartozik például a teljesítmény tesztelés, stressz tesztelés, regresszió tesztelés.

8. **Validáció** Felhasználói elfogadási tesztek igazolják, hogy az elkészült alkalmazás megfelel-e a megrendelő igényeinek, és készen áll-e rá, hogy éles környezetben használják. [10]



2.1. ábra. V-modell [14]

A V-modellt számos kritikával illették, melyek szerint a valós szoftverfejlesztés esetén nem állja meg a helyét. Ezekre a dolgozat nem tér ki részletesebben.

2.2. Tesztelés

A szoftverek egyre nagyobb szerepet kapnak a mindennapi életben. Viszont a legtöbb ember már találkozott szoftverhibával. Ez jó esetben csak kis kellemetlenséggel járt. Viszont elképzelhető olyan eset is, amelynek során az illetőnek nagy kára keletkezik, vagy akár

emberélet is múlhat rajta. Ezek a hibák lehetnek emberi hibák következményei, de bekövetkezhetnek környezeti hatás miatt is. Utóbbira példa a sugárzás, mágnesség, elektromos mezők. A tesztelés segít csökkenteni a kockázatát annak, hogy hibás szoftver kerüljön használatra. Továbbá a szoftver minőségét is mérni lehet vele a talált hibákkal.

2.2.1. Fogalmak

Tesztelés

> "A folyamat, amely magába foglalja az összes életciklus tevékenységet, mind a statikusat mind a dinamikusat, ami kapcsolatban áll a szoftver és a hozzátartozó munkatermékek tervezésével, előkészületével és kiértékelésével. Mindezt a célból, hogy egyrészt eldönthető legyen az, hogy ezek kielégítik-e a velük szemben támasztott követelményeket. Másrészt, hogy demonstrálják azt, hogy megfelelőek az adott célra. Harmadrészt, hogy detektálják a hibákat, hiányosságokat." [8]

Teszt

> "Egy vagy több teszteset halmaza." [8]

Teszteset

> "Bemeneti értékek, végrehajtási előfeltételek,elvárt eredmények és végrehajtási posztkondíciók halmaza, bizonyos cél vagy tesztfeltétel elérése érdekében, mint például igazolni egy adott követelmény teljesítését." [8]

Tesztfeltétel

> "Egy komponens egy adata vagy eseménye, ami igazolható egy vagy több tesztesettel, például: egy függvény, funkció, tranzakció, minőség attribútum vagy strukturális elem." [8]

Tesztautomatizálás

> "Szoftver használata teszttevékenységek végrehajtására vagy segítésére. Például: tesztmenedzsment, teszttervezés, tesztvégrehajtás és eredmény ellenőrzés." [8]

Egységteszt

> "Egy különálló program vagy modul tesztelés, annak érdekében, hogy meggyőződjünk róla, hogy nem tartalmaz hibát." [4]

Integrációs teszt

> "Folyamatos linkelése és tesztelése a programoknak vagy moduloknak abból a célból, hogy megbizonyosodjunk a helyes működésről a teljes rendszerben." [4]

Rendszerteszt

> "Megtervezett tesztelés a teljes, integrált rendszeren, annak érdekében, hogy kiértékelésre kerüljön a rendszer megegyezősége a megadott követelményekkel." [3]

2.2.2. Alapelvek

Számos tesztelési alapelv került javaslatra az elmúlt 40 évben, amik általános irányelveket fogalmaznak meg, amelyek közösek az összes tesztelésben. [7]

1. alapelv - A tesztelés megmutatja a hibák jelenlétét
A tesztelés képes megmutatni egy hiba jelenlétét, viszont nem tudja bizonyítani azt, hogy nem létezik hiba. A tesztelés csökkenti a valószínűségét annak, hogy felfedezetlen hiba marad a szoftverben, de még ha nem is talál hibát, akkor sincs bizonyítás a helyességre.

2. alapelv - Kimerítő tesztelés nem lehetséges
Mindennek a tesztelése (a bemenet és az előfeltételek összes kombinációja) nem megvalósítható, kivételt képeznek a triviális esetek. A kimerítő tesztelés helyett kockázat analízis és prioritások használata javasolt a tesztelési erőfeszítések fókuszálására.

3. alapelv - Korai tesztelés
Ahhoz, hogy a hibákat már a korai fázisban megtaláljuk, arra van szükség, hogy a tesztelési tevékenységeket a lehető legkorábban elkezdjük.

4. alapelv - Hiba fürtözés
A tesztelési erőfeszítésnek arányosan kell megoszlania, a modulok várt, és későbbiekben megfigyelt hibasűrűségének alapján. Általában kis számú modul felelős a hibák nagy részéért.

5. alapelv - Rovarirtó paradoxon
Ha ugyanazokat a teszteket futtatjuk újból és újból, végül ez a halmaza a teszteseteknek nem fog a továbbiakban új hibákat találni. Ennek megoldására időnként felül kell vizsgálni a teszteseteket, módosítani őket, és újakat adni hozzájuk.

6. alapelv - A tesztelés kontextusfüggő
A tesztelés különbözően van végrehajtva különböző kontextusokban. Például egy biztonságkritikus rendszer máshogy van tesztelve, mint egy webalkalmazás.

7. alapelv - Hiba-hiány megtévesztés
Egy hiba megtalálása és kijavítása nem segít, ha a készített rendszer használhatatlan vagy nem elégíti ki a felhasználó igényeit, elvárásait.

2.3. Eclipse

Az Eclipse egy integrált fejlesztőkörnyezet, amit leggyakrabban a Java IDE miatt használnak. Vastagkliens alkalmazások fejlesztésére alkalmas, elsősorban Java nyelven, de más nyelveket is támogat (például: C, C++, Perl, PHP, Ruby, Erlang). A felépítésére az jellemző, hogy könnyen bővíthető plug-inekkel, ezáltal széles a lehetséges használati köre. Az

utóbbi tulajdonsága miatt került használatra a szakdolgozat során, hiszen az alap Java IDE-n kívül szüksége volt további plug-inekre (például: Yakindu, Xtext, Xtend). [2]

Jelen dokumentum írásakor a legfrissebb Eclipse verzió a Neon, de az implementáció elkészítéséhez az eggyel korábbi, a Mars került használatra. A 2.2. ábrán látható az Eclipse logója.

2.2. ábra. Eclipse logó [1]

2.4. Yakindu

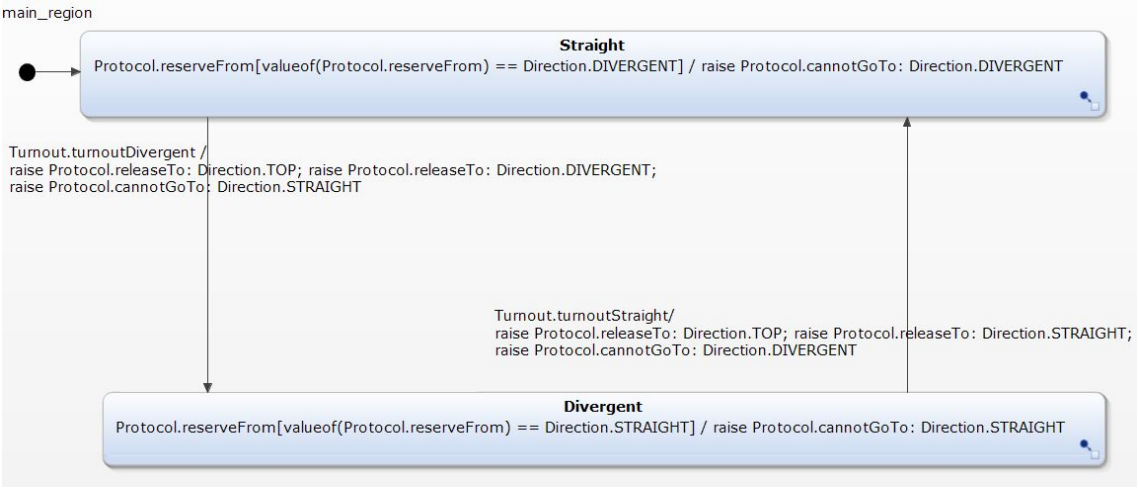
A Yakindu egy moduláris eszköztár beágyazott rendszerek modellalapú fejlesztéséhez. Az alapja a nyílt forráskódú Eclipse fejlesztő környezet, amiben plug-inként telepíthető. [6] Ezen eszköz segítségével állapotgépekkel történő modellezésre van lehetőség. Használata viszonylag könnyen elsajátítható. [5]
YAKINDU Statechart Tools az alábbi eszközöket biztosítja az állapotgép diagramok kezeléséhez:

- **Állapotgép diagram szerkesztő**, hogy grafikusan lehessen létrehozni és szerkeszteni az állapotgépeket.
- **Állapotgép szimulátor**, hogy lehessen szimulálni az állapotgépek viselkedését.
- **Kódgenerálás Java, C, C++ nyelvekhez**, hogy az állapotgép kóddá alakítható legyen.
- **Egyedi generátor projektek**, hogy könnyen készíthető legyen modell-szöveg transzformáció Xtenddel vagy Javaval.
- **Integrált validátor**, ami ellenőrzi az állapotgép modell szintaktikai és szemantikai problémáit.

A 2.3 ábrán egy példa látható arra, hogy hogyan néz ki egy Yakinduban készített állapotgép. Ez az állapotgép a biztonsági logikának a váltó komponensét hivatott modellezni.

2.5. MQTT

Az MQTT (Message Queuing Telemetry Transport) egy pehelysúlyú protokoll a TCP felett. Publisher-subscriber elven működik. Ez azt jelenti, hogy fel lehet iratkozni különböző témákra, amelyeken üzenetszórás jelleggel terjed az információ. Tehát az adott témára feliratkozottak mind megkapják az üzenetet. Ennek az az előnye, hogy adott komponens csak a számára érdekes témában kap vagy küld üzenetek.



2.3. ábra. Yakindu állapotgép példa

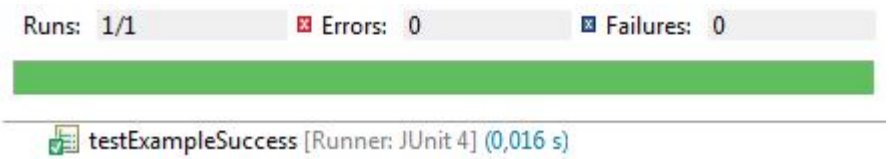
A MoDeS3 projekt támogatja ezt a kommunikációs protokollt, például a váltóknak megfelelő, mikrokontrollereken futó logikák képesek ezzel a módszerrel üzeneteket küldeni egymásnak.

2.6. JUnit

A JUnit egy egyszerű keretrendszer megismételhető tesztek írásához. Ez egy példánya az xUnit architektúrának egységtesztelő keretrendszerekhez. [9]
JUnit segítségével Java programozási nyelven írt kódhoz lehet egységteszteket készíteni. Ezekben a tesztkódokban különböző annotációkkal kell jelezni, hogy az adott metódusnak mi a funkcionalitása.

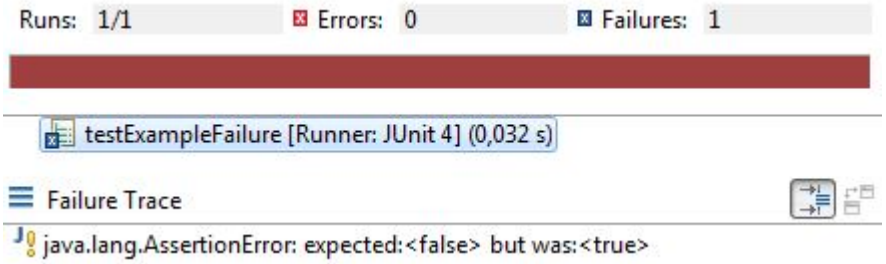
Gyakori annotációk
@Before - minden teszt metódus előtt végrehajtandó kód, tulajdonképpen egy kezdő állapotba visszaállítás, hogy az összes teszt metódus ugyanabból az állapotból induljon.
@BeforeClass - első teszt metódus előtt végrehajtandó kód, általában inicializálás történik az így megjelölt metódusokban.
@Test - valaminek a tesztelésére szolgaló metódus feletti annotáció, a teszt ítéletének a meghozásához az *assert* függvény valamely változata használható, amelynek meg kell adni egy várt eredményt, és a ténylegeset.
A JUnit tesztek háromféle eredményt adhatnak. Ezek az alábbiak:

- *siker*: abban az esetben, ha az elvvárt és a tényleges eredmények megegyeznek, ilyenkor a tesztünk zöld,



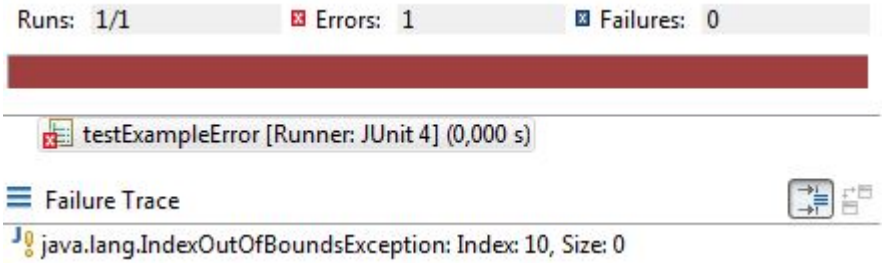
2.4. ábra. sikeres teszt

- *sikertelen*: abban az esetben, ha az elvárt és tényleges eredmények nem egyeznek meg, ilyenkor a tesztünk kék,



2.5. ábra. sikertelen teszt

- *hiba*: abban az esetben, ha nem sikerült a teszt futtatása, például egy nem várt Exception-t kapunk, ilyenkor piros a teszt.



2.6. ábra. hibás teszt

2.7. Xtext

Az Xtext programnyelvek és domén-specifikus nyelvek fejlesztésére szolgáló keretrendszer. Az Xtext-tel a saját nyelv definiálása egy erős nyelvtan segítségével történik. Eredményképpen egy teljes infrastruktúrát kapunk, beleértve az értelmezőt, a linkert, a típus ellenőrzőt, a fordítót, valamint szerkesztő támogatást Eclipse-hez, IntelliJ-hez, és a webböngészőhöz. [13]

2.8. Xtend

Az Xtend egy rugalmas és kifejező dialektusa a Java programozási nyelvnek, ami olvasható Java 5 kompatibilis forráskóddá fordul. Bármely Java könyvtárral probléma nélkül használható. A lefordított kimenet olvasható és szépen nyomtatott, és hajlamos ugyanolyan gyors futásra, mint a vele ekvivalens Java kód. [12]
Több oka van annak, hogy az Xtendre esett a választás. Egyrészt a projektben a kód nagy részét Xtendben írták, ennek az oka, hogy jól együttműködik az Eclipse-szel és a Javaval is, de egyszerűbb benne programozni. Másrészt alkalmas kódgenerálásra és illeszkedik az Xtexttel, így minden adott a saját nyelv alapján történő különböző tesztesetek generálásához.

3. fejezet

Tesztelendő rendszer

3.1. Domain

Biztonságkritikus beágyazott rendszerekről dolgok? MoDeS3-ról általános dolgok?

3.2. Architektúra

A nyári átírása annak megfelelően, hogy mik változtak. KÉP az architektúráról

3.3. Szoftver komponensek

Transport Messaging Leapmotion? Dashboard? Safetylogic (nem cl, cl)

4. fejezet

Tervezés

4.1. Tesztelés tervezése

TODO 1 váltó
Több váltó
táblázatok

4.2. DSL nyelv

Általános írás a nyelvről. Mit tud egy DSL nyelv. Milyen DSL nyelvet hoztam létre. A konkrét megvalósítás a következő fejezetben.

5. fejezet

Implementáció

5.1. XText nyelvtan

Kis általános, ide jöhetne pl itlaboros elmélet, kis példával.
Hogy épül fel a konkrét nyelvtanom, miért úgy. Példa (az egész, vagy csak részlet? (ha csak részlet, akkor függelékbe az egész?))

5.2. XTend generálás

Ábra, hogy a DSL-ben leírtból hányféle kód generálódik. Kép az Eclipseből, Generate gomb, ennek a képnek a leírása.

Szintek különválasztása. Esetleg kis példakód.

5.3. Illesztés

Itt leírnám, hogy kellett hozzá létrehozni valamiféle csonkot a tesztelési szintekhez, hogy a működést tesztelni lehessen. Ezeket a szinteket külön alfejezetbe tenném.

6. fejezet

Tesztek és eredmények

6.1. Tesztesetek

Érdekesebb tesztesetek kicsit részletesebb leírása. Számadatok, pl melyik szinthez hány teszteset, különböző számú váltóhoz hány teszteset, akár egy szép táblázatba.

6.2. Futtatási eredmények

Kiértékelés. Mennyi futott le helyesen, mennyi nem. Táblázat/diagram

6.3. Hibák

A talált hibák. Már le lehetne írni az MQTT-s projektben találtat. (?)

7. fejezet

Összefoglalás

7.1. Eredmények

7.2. Továbbfejlesztés

Irodalomjegyzék

[1] Eclipse. https://eclipse.org/org/.

[2] Eclipse(ide). https://en.wikipedia.org/wiki/Eclipse_(software).

[3] Ieee standard for system and software verification and validation, 2012. 05.

[4] Iso/iec 2382-20:1990, information technology–vocabulary–part 20: System development, 2015. 05.

[5] ITEMIS. *Yakindu 5 minutes tutorial*. https://www.itemis.com/en/yakindu/statechart-tools/documentation/tutorials/#oss_five-minutes-tutorial.

[6] ITEMIS. *Yakindu User-guide*. https://www.itemis.com/en/yakindu/statechart-tools/documentation/user-guide/#YAKINDUStatechartToolsReference.

[7] Matthias Hamburg Judy McKay: *Certified Tester - Foundation Level Syllabus*. ISTQB, 2011. 03.

[8] Matthias Hamburg Judy McKay: *Standard Glossary of Terms Used in Software Testing*. ISTQB Glossary Working Group, 2016. 03. http://www.istqb.org/downloads/category/20-istqb-glossary.html.

[9] Junit. http://junit.org/junit4/, 2016.

[10] V-model (software development). https://en.wikipedia.org/wiki/V-Model_(software_development), 2016.

[11] What is v-model- advantages, disadvantages and when to use it? http://istqbexamcertification.com/what-is-v-model-advantages-disadvantages-and-when-to-use-it/.

[12] Xtend. http://www.eclipse.org/xtend/.

[13] Xtext. http://www.eclipse.org/Xtext/.

[14] Dr. László Zoltán: Szoftver technology, 2014. Software process.