Szegedi Tudományegyetem
Informatikai Tanszékcsoport

Lefedettség-mérés beágyazott rendszereken

Diplomamunka

Készítette:
Bognár Szabolcs
programtervező
matematikus hallgató

Témavezető:
Dr. Gergely Tamás
egyetemi adjunktus

Szeged
2013

Feladatkiírás

Kutassa fel a beágyazott rendszerek tesztelésének eddigi módszereit, lehetőségeit és eredményit. Kutassa fel a funkcionális és strukturális tesztelési módszerek eddigi lehetőségeit, az általuk nyerhető információkat és azok felhasználási lehetőségeit. Kutassa fel a funkcionális és strukturális tesztelési módszerek együttműködési lehetőségeit. Tervezzen és valósítson meg egy módszertant, mely beágyazott rendszerek funkcionális tesztelését segíti és annak minőségét javítja strukturális tesztelés során nyert adatok segítségével.

Tartalmi összefoglaló

- ***A téma megnevezése:***

Beágyazott rendszerek metódus szintű kódlefedettség-mérése.

- ***A megadott feladat megfogalmazása:***

Android operációs rendszerű, digitális televízióadások vételére alkalmas beltéri egység (SetTopBox) beágyazott rendszerének tesztelési módszertanainak felkutatása, funkcionális és strukturális tesztelésének együttműködési lehetőségeinek felderítése, a rendszer metódusszintű kódlefedettség-mérése, társított tulajdonságok meghatározása.

- ***A megoldási mód:***

Kutatás: tesztelési módszerek megismerése, a különböző tesztelési módszerekkel gyűjthető információk megismerése, a különböző tesztelési módszerek együttműködési lehetőségeinek megismerése, általánosan a beágyazott rendszerek megismerése, a konkrét eszköz megismerése, az eszközt tesztelő rendszer megismerése.

Tervezés: a megfelelő tesztelési és adatgyűjtési módszerek kiválasztása, a megfelelő kinyerhető adathalmaz összeállítása, a kommunikáció csatornájának kiválasztása, a kommunikációs eszközök és protokoll megtervezése.

Fejlesztés: tesztelő modul (plug-in) fejlesztése, adatgyűjtő szerver fejlesztése, kommunikációs rend kiépítése, próbaalkalmazás (dummy) készítése, tesztesetek gyártása, mért/gyűjtött adatokat kiértékelő program készítése.

Egyéb: próba-alkalmazás, tesztelés, dokumentálás, kiértékelés.

- ***Alkalmazott eszközök, módszerek:***

Visual Studio 2008 és 2010, RT-Executor, Eclipse, Notepad++, Javaassist, Android sdk

BlackBox testing, WhiteBox testing, GreyBox testing, coverage collection, trace path collection, traceability calculation, runtime calculation, Java binary instrumentation

- ***Elért eredmények:***

Bármilyen Android programot tudunk instrumentálni, majd a megfelelő service alkalmazással ellátott berendezésen futtatni, és róla metódus-szintű lefedettségi információkat gyűjteni. Ezekből az információkból további jellemzőket tudunk számolni, melyek alapján egy meglévő funkcionális teszteset-halmazt tudunk priorizálni, belőle valamely attribútum alapján a legmegfelelőbbeket kiválasztani.

- ***Kulcsszavak:***

Beágyazott rendszerek, Android, funkcionális tesztelés, strukturális tesztelés, lefedettség mérés.

Tartalomjegyzék

Feladatkiírás 2

Tartalmi összefoglaló 3

Tartalomjegyzék 4

BEVEZETÉS 5

1. ÁLTALÁNOS ÁTTEKINTÉS 6

1.1. A beágyazott rendszerek 6

1.2. Az Android operációs rendszer 7

1.3. A kiválasztott berendezés 8

2. A TESZTELÉS 10

2.1. A tesztelés fontossága 10

2.2. A tesztelés fajtái 11

2.2.1. Funkcionális, azaz BlackBox tesztelés 11

2.2.2. Strukturális, azaz WhiteBox tesztelés 12

2.2.3. A Gray-box tesztelés 13

3. BEÁGYAZOTT RENDSZEREK TESZTELÉSI TECHNOLÓGIÁI 14

4. LEFEDETTSÉG-MÉRÉS BEÁGYAZOTT RENDSZEREKEN 17

4.1. Kutatás 17

4.3. Tervezés 24

4.4. Fejlesztés 32

4.5. Kiértékelés / Eredmények 36

5. AZ EREDMÉNYEK UTÓÉLETE: ÚJ PROJEKTEK AZONOS ALAPOKON 37

Irodalomjegyzék 38

Nyilatkozat 41

Köszönetnyilvánítás 42

Mellékletek 43

BEVEZETÉS

Dolgozatom alapját a CIRENE projektben [16],[17] végzett kutatási és fejlesztési munkám jelenti. A dolgozat általános részeit a [4] jelzetű felmérés kutatási társszerzőjeként gyűjtött információk alapján írtam. A korábbi munkám, és így a dolgozatomban leírt eredmények lényegi részét ebből a kutatásból szerzett összegzések és új ismeretek, valamint azok hasznosításának lehetőségei, tervei teszik ki.

A CIRENE projektben célul tűztük ki egy beágyazott rendszerek teszteléséről szóló általános módszertan definiálását, mivel ilyen eddig nem létezett a beágyazott rendszerek sokfélesége miatt, és amely módszertant egyszerűen lehet specializálni az eltérő rendszerekhez. Ebben a módszertanban különböző tesztelési módszereket ötvöztünk a tesztelési folyamat hatékonyságának növelésére, és az így nyert információkat felhasználtuk a kódot jellemző adatok meghatározására, valamint a kísérleti berendezés tesztelésére használt tesztbázis kibővítésére és szűrésére. A projekt keretein belül elkészült egy létező teszteszközt (RT-Executor [14]) kibővítő beépülő modul (plug-in) és egy általánosan felhasználható adatgyűjtő alkalmazás, amely megvalósítja a kidolgozott általános módszertan egy Android operációs rendszert futtató digitális beltéri egységre (Set-Top Box) specializált verzióját.

Dolgozatomban helyet kap a CIRENE projektben végtett kutatási, tervezési és fejlesztési munka leírása és eredménye, valamint a projekt utóéletének tekinthető későbbi projektek ismertetése, melyek a korábban elért eredményekre és kifejlesztett eszközökre támaszkodva új eredményeket hoztak, és a korábban kifejlesztett módszertan és eszközök egy megfelelően specializált változatát alkalmazták.

1. ÁLTALÁNOS ÁTTEKINTÉS

1.1. A beágyazott rendszerek

A beágyazott rendszerek meghatározására számos intuitív és hivatalos definíció található. A legáltalánosabb és a laikusok számára talán a legegyszerűbben megérthető intuitív definíció, hogy *minden olyan berendezés, amelyet számítógép vezérel vagy működésében segít, de ezt első ránézésre nem mondanánk meg róla, az beágyazott rendszernek tekinthető*. Egy formálisabb megfogalmazásban ez így hangzik: *„Beágyazott rendszernek nevezzük a számítógépes hardver és szoftver egy olyan kombinációját (szükség szerint kiegészítve egyéb mechanikai és más elemekkel), amelyet egy kijelölt feladat elvégzésére terveztek.”* [13]. Ez a formális definíció jól mutatja a beágyazott rendszerek lényegét: egy bizonyos feladat elvégzésére tervezik és fejlesztik ki őket, így másra nem is alkalmasak. Ugyanezen okból rendkívül változatosak mind hardver, mind szoftver terén.

Számos példa sorolható fel beágyazott rendszerekre: ipari megmunkáló és feldolgozó berendezések, közlekedési lámpák, hordozható zenelejátszók, videojátékok, mosógépek, sőt szinte az összes „modern” háztartási gép alapvetően beágyazott rendszernek tekinthető, de a felsőbb kategóriás gépjárművek is nagyban függenek a beépített érzékelőktől, melyeket beágyazott rendszerek vezérelnek bennük.

Beágyazott rendszerekkel manapság már a mindennapi életben számos helyen találkozunk, még ha nincs is róla tudomásunk. Legmeghatározóbb színterük az iparban van: megmunkálógépek, vezérlőrendszerek, automatizáló berendezések, mind-mind számítógép-vezérlésű gépek, melyek beágyazott rendszerekkel vannak felszerelve, kontrollálva. A köztudatba mégsem a legmeghatározóbb, hanem a leglátványosabb és a legnagyobb kényelmi szolgáltatásokat nyújtó berendezések vonultak be, mint beágyazott rendszerek: ilyenek például a szórakoztatóelektronikai termékek és a háztartási gépek, valamint a gépjármű-ipar által nyújtott extra felszereltségek. A hétköznapi ember nem is sejti, de a csuklóján hordott, sokfunkciós és nem is feltétlenül túl drága karóra is már beágyazott rendszer. Egyértelmű példa erre a sokak által kedvelt digitális karórák családja, melyek közül a legegyszerűbbek csak az időt és a dátumot mutatják, és egy ébresztési időpont is beállítható rajtuk. Ezt is beágyazott rendszer

működteti, amely speciálisan erre a néhány funkcióra lett kifejlesztve, így olcsó és könnyen sokszorosítható. A kevésbé elterjedt, de mégis többek által használt vagy vágyott órák a szintén beágyazott rendszerrel vezérelt atomórák, és a közéjük tartozó (akár analóg) karórák. Ezekben nem is a legbutább beágyazott rendszerek találhatóak, mert egy központi szerverrel való szinkronizálás után önkorrekciót kell végezniük, ami igen összetett folyamat.

Amikor számítógépekről beszélünk, akkor a legtöbb embernek a hétköznapokból jól ismert asztali számítógépek jutnak eszébe monitorral, billentyűzettel, ám nagy különbségek vannak ezek és a beágyazott rendszerek között. Alapvető különbség a hagyományos, általános célú számítógépek (mint például a személyi számítógépek) és a beágyazott rendszerek között, hogy míg az általános célú gépek (ahogyan a nevük is mutatja) a hétköznapi élet minden területére kiterjedő funkcionalitások ellátására képesek (munka, szórakozás, stb.), addig a beágyazott rendszerek feladata már a tervezésük kezdeténél meghatározott; speciális céllal, speciális körülmények közé készülnek, így tervezésük, kialakításuk, méretük és működésük az adott feladatra optimalizált. Gyakran valós idejű berendezések, azaz egy jel érzékelése, fogadása és a rá adott reakció között nagyon kis időnek szabad csak eltelnie, így a rajtuk futó szoftvereknek szigorú szabályoknak kell megfelelniük és alapos tesztelésen kell átesniük forgalomba kerülésük előtt.

1.2. Az Android operációs rendszer

A szórakoztató- és telekommunikációs ipar multimédiás termékeinek robbanásszerű fejlődése maga után vonta a termékeket megfelelően vezérelni képes és lenyűgöző felhasználói élményt nyújtó operációs rendszerek fejlesztését.

Az Android mobil operációs rendszert ezen új fejlesztések egyike, melyet az Android Inc. kezdte tervezni és fejleszteni a Google támogatásával, majd a Google fel is vásárolta a vállalatot, és az Open Handset Alliance társaságában fejlesztette tovább [1]. A legfőbb motiváció egy könnyen kezelhető, bárki által könnyen bővíthető, nyílt forráskódú, menedzselt kódú mobil operációs rendszer kifejlesztése volt. Ehhez nyílt Linux Kernel alapra építkeztek, Java programozási nyelven, és létrehoztak egy virtuális

gépet, mely optimalizált a memória és a hardver-erőforrások kezelésében: a Dalvik Virtuális Gépet [30].

A rendszer nem tesz különbséget az alap gyári alkalmazások és a harmadik féltől származó fejlesztések között: minden Androidra írt program egyforma jogosultságokkal rendelkezik, a berendezés bármely erőforrását elérheti, használhatja. Közreadtak továbbá temérdek eszközt és könyvtárat a fejlesztések megkönnyítésére és a lehető legszínesebb és leggazdagabb alkalmazások fejlesztésének támogatására [2],[3].

Ma már hatalmas fejlesztőgárda áll az Android mögött, beleértve a szabadúszó programozókat is; rövid idő alatt a közösségi fejlesztés egyik sikerstoryjává vált. Ráadásul a rendszer optimális erőforrás-kezelése miatt egyre elterjedtebb a specializálása és alkalmazása a beágyazott rendszerek más, különbözőbb fajtáiban is.

Az Android rendszer egy másik sajátossága, hogy mindaddig, amíg a harmadik fél által elkészített program nincs aláírva egy a Google-nél regisztrált kulccsal, vagy egy általánosa elfogadott privát-publikus kulcspárral, addig nem lehetséges a telepítése semmilyen Android rendszert futtató berendezésre. Így biztosítja a Google a programok nyomon követhetőségét és az ártalmas szoftverek készítőinek kiszűrését.

1.3. A kiválasztott berendezés

A beágyazott rendszerek vezérlőberendezései kompakt számítógépek, speciális feladatok ellátására tervezve és megvalósítva, így természetesen a rajtuk futó szoftverek is speciálisak. Az Android operációs rendszert alapvetően okostelefonokra fejlesztették ki, melyek korunk egyik legösszetettebb beágyazott rendszereinek tekinthetők. A gyártók, a hardverek és az igények sokszínűsége miatt az Android rendkívül rugalmas rendszer, bármilyen platformon elfut és a berendezés teljes erőforrás-kínálatát ki tudja használni, így az okostelefonokhoz nagyban hasonlatos táblagépek (tablet-PC) újra a tervezők és a kivitelezők figyelmének középpontjába kerültek és felvirágoztak. Ebből kiindulva az elmúlt időszakban megindult egy kezdeményezés, melyben Android operációs rendszert telepítettek egy digitális tévéadás vételére alkalmas beltéri egységre (Set-Top Box). A kutatáshoz tartozó kísérleti alkalmazásokat egy ilyen, még kísérleti stádiumban lévő, Android operációs rendszert futtató Set-Top Box-hoz fejlesztettük ki.

A dolgozatom alapját képező kutatások óta a Set-Top Box-okon futó Android rendszerek nagy változáson mentek keresztül, és már kereskedelmi forgalomba is kerültek bizonyos gyártók termékei, melyek megfelelően specializált Android rendszert futtatnak [5][6].

A Set-Top Box-okon kívül már megjelentek headless Android rendszert futtató pénztárgépek, melyek tesztelését szintén az itt bemutatott eredményeket felhasználásával segítették elő. A headless jelző azt jelenti, hogy az operációs rendszer nem képes grafikus elemek megjelenítésére, sőt, ezek az elemek le is vannak tiltva a rendszerben, és alkalmazásuk hibát és programleállást eredményez. Ezekről a rendszerekről és a tesztelésükről később, a dolgozatom témájával kapcsolatosan bővebben írok.

2. A TESZTELÉS

2.1. A tesztelés fontossága

A szoftverek fejlesztési életciklusának jelentős és meghatározó részét képezi a tesztelés, mely során a szoftverben előforduló szintaktikai, szemantikai és működésbeli hibákra igyekszünk fényt deríteni. A szoftver kiadása előtt felfedezett és kijavított hibák nagy mértékben javítják a szoftver minőségét és nagyságrendekkel több karbantartási költségtől kímélik meg a szoftver kibocsátóit, mint ha tesztelés nélkül adták volna ki, vagy ha csak a fejlesztők tesztelték volna a saját kódjaikat. Másfelől ha a megrendelő sok hibát talál a programban, a fejlesztőcég tekintélye nagy mértékben csökken a szemében, és akár a vásárlástól is elállhat.

A tesztelés másik fontos előnye, hogy segítségével különféle viselkedési és minőségi attribútumokat tudunk mérni a kódon. Megfelelő előkészítés (például instrumentálás, melyet később bemutatok) után a tesztesetek nem csak a számított kimenetet biztosítják, melyet összehasonlítva egy elvárt kimenettel az egyes tesztesetekre pass/fail címkét ragaszthatunk, hanem egyéb információval is szolgálnak, mint például lefedettségi adatok, vagy futásidő. Ezeket az információkat felhasználva beszédes metrikákat tudunk felállítani, vagy további származtatott attribútumokat tudunk meghatározni, melyek jól jellemzik a kódunkat. Ilyen metrikák és jellemzők például a McCabe komplexitás (azaz Ciklomatikus komplexitás; a programban található ciklusok legnagyobb egymásba ágyazási mélységével kapcsolatos metrika) [7], az Adatfolyam-komplexitás, a Döntési komplexitás (azaz a programban található elágazások, döntési pontok legnagyobb egymásba ágyazási mélységével és összetettségével kapcsolatos metrika); a Metódus szintű lefedettség, az Utasítás szintű lefedettség, az Elágazás szintű lefedettség, azaz, hogy a programban található metódusokból, utasításokból, elágazásokból mennyit érintettünk a program egy futtatása során, vagy egy tesztelési ciklus alkalmával.

Az itt felsorolt metrikák és jellemzők csak a töredékét képezik az iparban található és használatos metrikák halmazának.

2.2. A tesztelés fajtái

A tesztelési technikákat és módszertanokat sok szempont alapján lehet kategorizálni. Egy ilyen szempont például, hogy futtatjuk-e a kódunkat a teszteléshez.

Statikusnak nevezünk egy tesztelést, ha a program kódjának végrehajtása nélkül végez vizsgálatot. Ilyen módon bármiféle írott, a szoftverhez tartozó terméket lehet vizsgálni: követelményspecifikációt, terveket, dokumentációkat, és magát a forráskódot is. Az ilyen vizsgálat lehet kézi vagy automatizált is. Automatikusan leggyakrabban a forráskódokat szokták elemezni a szintaktikai szabályozottságuk miatt. Természetesen ilyen vizsgálatok során nem derülnek ki működési, szemantikai problémák.

Dinamikusnak azokat a tesztelési módszereket nevezzük, melyek a program végrehajtásával végzik vizsgálataikat. Ezek a folyamatok könnyen automatizálhatóak megfelelő segédalkalmazásokkal, melyekben definiálni tudjuk a vizsgált program által megtenni kívánt lépéseket, megadhatjuk a szükséges bemeneti értékeket, elkaphatjuk a kimeneti értékeket, és összehasonlíthatjuk azokat elvárt kimeneti eredményekkel.

Számomra ez utóbbi fajták szolgáltak hasznos információkkal, kutatásaimban velük foglalkoztam. A következő alfejezetekben a legismertebb dinamikus tesztelési módszereket mutatom be röviden, melyekkel én is dolgoztam.

2.2.1. Funkcionális, azaz BlackBox tesztelés

Ezt a kategóriát tekinthetjük a legalapvetőbb és talán legkezdetibb (mégsem elavult!) fajtának. Funkcionális tesztelés során nincs betekintésünk a program működésébe, csak a bemeneteket és a kimeneteket látjuk. Ezért is nevezik BlackBox, azaz fekete doboz technikának; a program maga egy átlátszatlan, fekete dobozként működik, amelybe inputokat pakolunk, és outputokat gyárt. A tesztelés folyamata pontosan ez: ismerjük a program követelményit, tehát hogy milyen adatokon kell műveleteket végeznie, milyen műveleteket kell végrehajtania, és így milyen kimeneteket kell produkálnia adott bemenetek esetén. A követelményekből fel tudunk állítani egy tesztadat-bázist, melyek bemenet-kimenet párokat tartalmaznak. Egy teszteset egy ilyen pár felhasználása a következő módon: a pár bemenet tagját odaadjuk a programnak és megvárjuk a kimenet előállítását. A pár kimenet tagját, mint elvárt kimenetet, referenciaként használva összehasonlítjuk a program által előállított kimenettel. Bizonyos hibaszázalékot

megengedve vizsgáljuk a különbséget. Ha a hibahatárokon belül vagyunk, a program megfelelt az adott teszteseten. Ha a program minden teszteseten megfelelt, vagy ha szükséges, természetesen itt is felállíthatunk egy határértéket, hogy mennyi tesztesetnek kell sikeresen lezajlania, hogy elfogadjuk a végső eredményt, és ha ezt a határértéket elértük, akkor a funkcionális tesztelést befejezettnek tekinthetjük. Ha a tesztadat-bázis elég körültekintően volt összeállítva és figyelembe vette a megfelelő szélső- és kirívó eseteket, akkor a programot felkészültnek tekinthetjük éles alkalmazásra.
Természetesen ebben az esetben nem kapunk információt a program hatékonyságáról, a benne megvalósított algoritmusok helyességéről, pusztán arról, hogy megfelelően végzi a feladatát.

2.2.2. Strukturális, azaz WhiteBox tesztelés

Ezt a tesztelési fajtát tekinthetjük az előző fejezetben említett tesztelési fajta ellentettjének is. Tesztelés közben figyelemmel kísérjük a kódban az adatok útját, a kimeneti értékek képződését, az egyes programrészek futás közbeni viselkedését, stb. Mint ha a rendszer egy átlátszó (fehér) doboz (azaz WhiteBox) lenne, és mi a működését szemlélve értékelnénk ki azt. A bemeneti és kimeneti értékek nem játszanak lényeges szerepet, pusztán az a céljuk, hogy a rendszert meghajtsák. Pontosabban: a bemenet és a kimenet értéke olyan módon nem lényeges, mint a fenti kategória esetén. Olyan szempontból viszont lényeges, hogy a program számára valid, azaz kiértékelhető legyen a bemenet, ha a számítási algoritmusokat akarjuk vizsgálni, és fals, azaz ne legyen kiértékelhető, ha a program hibás bemenetre adott válaszát és viselkedését akarjuk vizsgálni.
Strukturális tesztelés során tehát magáról a programkódról szeretnénk valami információt szerezni: arról, hogy hogyan éri el a célját, hogyan hajtja végre a szükséges műveleteket, hogyan állítja elő a kimenetét. Ennek érdekében ez a tesztelési módszertan a lehető legkimerítőbben igyekszik bejárni a programkódot valamely aspektus szerint és közben adatok és jellemzők gyűjteni a kódról. Ilyen aspektus például, hogy érintsük az összes elágazást a kódban, vagy hogy járjuk be az összes útvonalat a programban, amit az adat bejárhat. A lehető legszigorúbb aspektusokban az az elvárás, hogy érintsük az összes utasítást vagy az összes kódsort a programban.

A strukturális tesztek során gyűjthető leglényegesebb és legalapvetőbb információ a lefedettségi metrika. Ennek is több fajtája van, minden aspektushoz legalább egy, de a legfőbb két metrika a kódlefedettség és az elágazás-lefedettség. Kutatásomban és munkámban a kódlefedettséget találtam legmegfelelőbb információnak a célom eléréséhez.

2.2.3. A Gray-box tesztelés

Mint a fenti két kategóriának, ennek is igen beszédes neve van. A GrayBox, azaz szürke doboz technika ugyan még nem egy teljesen kiforrott, egységes módszer, de minden elképzelése és változata egyezik abban, hogy a GreyBox nem más, mint a Funkcionális és a Strukturális tesztelési módszerek vegyítésével, azok együttműködésével alkotott tesztelési módszertan. Alapvetően értelmezhető úgy, mint egy „kibővített” strukturális tesztelés, ahol a bemenetek és kimenetek értékeit is figyelembe vesszük úgy, mint a funkcionális teszteknél. Bizonyos szempontból így is van: a GreyBox technikák lényege, hogy az egyik tesztelési módszer kimeneteit felhasználjuk a másik tesztelési módszer minőségének fejlesztésére. Funkcionális teszteléssel magáról a programról, a szoftver funkcionalitásáról tudunk minőségi jellemzőket mondani, míg Strukturális teszteléssel inkább a végrehajtott teszthalmaz minőségét tudjuk jellemezni. Ezért általában a strukturális tesztelés eredményeit szokták felhasználni a funkcionális teszteléshez, hiszen a strukturális tesztelés több információt tud szolgáltatni, és azokból az információkból sok egyéb, származtatott információt tudunk számolni, melyekkel a funkcionális tesztesetek halmaza bővíthető, csökkenthető, azon fontossági sorrend definiálható, stb.

A jelen dolgozat alapját képező kutatásom és fejlesztésem során a strukturális teszteléssel nyerhető kódlefedettségi információt használtam fel a funkcionális tesztesetek jellemzésére.

3. BEÁGYAZOTT RENDSZEREK TESZTELÉSI TECHNOLÓGIÁI

A beágyazott rendszerek egyik legfőbb jellemzője a változatosságuk, mind hardver, mind szoftver tekintetében. Speciális feladatok elvégzésére tervezték őket és az előállításuk során olyan hardverelemeket és elméleti, szoftveres megvalósításokat használtak fel, melyekkel az adott feladat optimálisan végezhető el, s így a legkülönfélébb platformokkal rendelkeznek. Ez a tesztelésüket megnehezíti, hiszen a különböző platformok teszteléséhez különböző külső eszközökre van szükség. Nem mellesleg speciális mivoltuk miatt a rájuk írt program egy lefordított, futtatható, úgynevezett bináris változata található csak meg, a forráskódot, ha lehet, titokban tartják. Gyakran a berendezésen található bináris kód is valamilyen fajta védelem alatt áll, hogy ne lehessen kiolvasni, kimásolni az eszköz memóriájából. Mégis a beágyazott rendszerben lévő bináris kódból is lehet információkat kinyerni, de sokkal nehezebb módon, és speciális eszközökre van szükség hozzá. Ebből a sokrétű speciálisságból ered, hogy tesztelésük nehéz és hosszadalmas lehet [15].

Másik lényeges szempont beágyazott rendszerek tesztelésénél, hogy legelterjedtebb fajtái valós idejű feladatokat végeznek, tehát az akció és a reakció közt eltelt idő nagyon rövid kell legyen. Ilyenkor a funkcionális tesztelés sikerességét nem csak az jelzi, hogy megkaptuk-e a várt és helyes eredményeket, hanem az is, hogy időben kaptuk-e meg őket.

Mindezen tulajdonságok és jellemzők azt sugallják, hogy a beágyazott rendszereket csak funkcionálisan lehet tesztelni. Nos, semmi sem állhatna távolabb a valóságtól.

A beágyazott rendszerek speciális volta és szigorú feltételeknek való megfelelési kötelezettsége miatt rendkívül nagy szükség van a mindenre kiterjedő és alapos tesztelésre. Ezekhez kifejlesztették a megfelelő eszközöket (és minden új beágyazott rendszer készítése közben elkészítik a hozzá tartozó ilyen eszközt), melyek alkalmasak a beágyazott rendszereken futó alkalmazások vagy akár a teljes rendszer monitorozására, a rendszerre új alkalmazás feltöltésére, a rendszeren futó alkalmazások módosítására, akár futás közben is (persze csak korlátozott mértékben és szabályozott keretek között). Ezek a debuggerek [18].

Léteznek hardveres és szoftveres debugger eszközök. Mindkét eszközcsalád lényege, hogy betekintést enged a beágyazott rendszer tárolóegységébe, memóriájába, így figyelemmel kísérhetjük a programok működését. Természetesen itt is nehézségeket okoz az, hogy az eszközön a szoftvernek csak a binárisa található meg. A fejlettebb hardveres debuggerek rendelkeznek olyan belső logikával, amely képes a bináris kódot egy ember számára könnyebben érthető, ám még mindig alacsony szintű kóddá alakítani. Ezek a kódok assembly-utasításokra emlékeztetnek a legjobban, ami még mindig nehezen olvasható és értelmezhető a tesztelők számára, de már nagyságrendekkel jobb, mint ha bináris kóddal kellene dolgozniuk. A szoftveres debuggerek általában a fejlesztőkörnyezet részét képezik és a fejlesztés alatti hibakeresést teszik lehetővé. Ezek látják a forráskódot és csatlakoznak az eszközre, hogy nyomon kövessék a program futását. Jól strukturált kezelőfelülettel rendelkeznek, melyen folyamatosan láthatjuk, hogy a programunk melyik részénél tart a futásban. Töréspontokat helyezhetünk el vele a kódban, melyeknél a futás megakad, és ahonnan akár soronként léptethetjük az alkalmazást, hogy teljes betekintést nyerjünk a működés egyes lépéseibe és a kiváltott mellékhatásokba. egyes szoftveres debuggerek és programozási nyelvek lehetővé teszik, hogy futás közben módosítsunk a programunkon, így tesztelve az egyes működési elméleteket vagy javítási ötleteket. Ezzel a módszerrel a fejlesztés közben rengeteg nem triviális hiba is felderíthető és javítható [19].

A beágyazott rendszerek jellemzője, hogy sokféle fizikai csatolófelülettel, inteface-szel rendelkeznek, melyek alapvetően megegyezhetnek az általános számítógépeken találhatóakkal, de funkcionalitásukban kibővített lehetőségekkel rendelkezhetnek. Például található lehet USB csatlakozási felület egy adott beágyazott rendszeren, melyre ha adathordozót vagy beviteli-kiviteli eszközt csatlakoztatunk, akkor úgy viselkedik, mint egy hagyományos USB-port, megfelelő csomagkezeléssel írható-olvasható rajta az adat, de ha egy debuggert csatlakoztatunk rá, akkor egy folyamatosan élő, szinkron vagy aszinkron csatornává változik, melyen keresztül a rendszeren futó szoftvert érhetjük el, monitorozhatjuk, szerencsésebb esetben bele is írhatunk a kódba. A leggyakoribb ilyen debugolásra is használt interface beágyazott rendszereken a soros port, de a gyártók gyakran kifejlesztenek maguknak saját, egyedi debug portokat, egyrészt a biztonság érdekében, másrészt hogy a gyártó saját debug eszközét kelljen

beszerezniük a fejlesztőknek [20],[21]. Az ilyen csatlakozási interface-ek biztosítanak lehetőséget a beágyazott rendszerek strukturális tesztelésére.

Ezek mellett léteznek a beágyazott rendszerek tesztelésére olyan eszközök, amelyek a fejlesztés korábbi fázisaiban is lehetővé teszik a teljes rendszer majdani működés és viselkedés modellezését, vizsgálatát: ezek az emulátorok [22]. Ezek a szoftver- és hardvereszközök olyan berendezések, amelyek képesek szimulálni, leutánozni bizonyos más berendezések működését, így lehetőséget biztosítanak például a személyi számítógépen modellezni egy még tervezés alatt álló mikrovezérlőt, amely egy későbbi beágyazott rendszer központi eleme lesz, vagy akár képesek modellezni egy teljes beágyazott rendszert, minden kimenetével és bemenetével együtt, így a beágyazott rendszerre készülő szoftvert már a legkorábbi fázisától kezdve tesztelni tudják. Továbbá léteznek ezeknek az eszközöknek olyan fajtái, amelyek képesek szoftver- és hardverkörnyezetek szimulálására is, így nem csak a fejlesztés alatt álló eszközt tudják modellezni, de például az eszköz egy elkészült prototípusát el tudják helyezni egy olyan szimulált környezetben, amely a későbbi felhasználás helyével megegyezik. Ez az eszközcsalád kiemelkedő lehetőségeket biztosít a fejlesztésekre és a tesztelésekre. Noha az emulátorok működése általában lassabb a valós rendszereknél, szinte minden esetben alkalmazzák őket.

4. LEFEDETTSÉG-MÉRÉS BEÁGYAZOTT RENDSZEREKEN

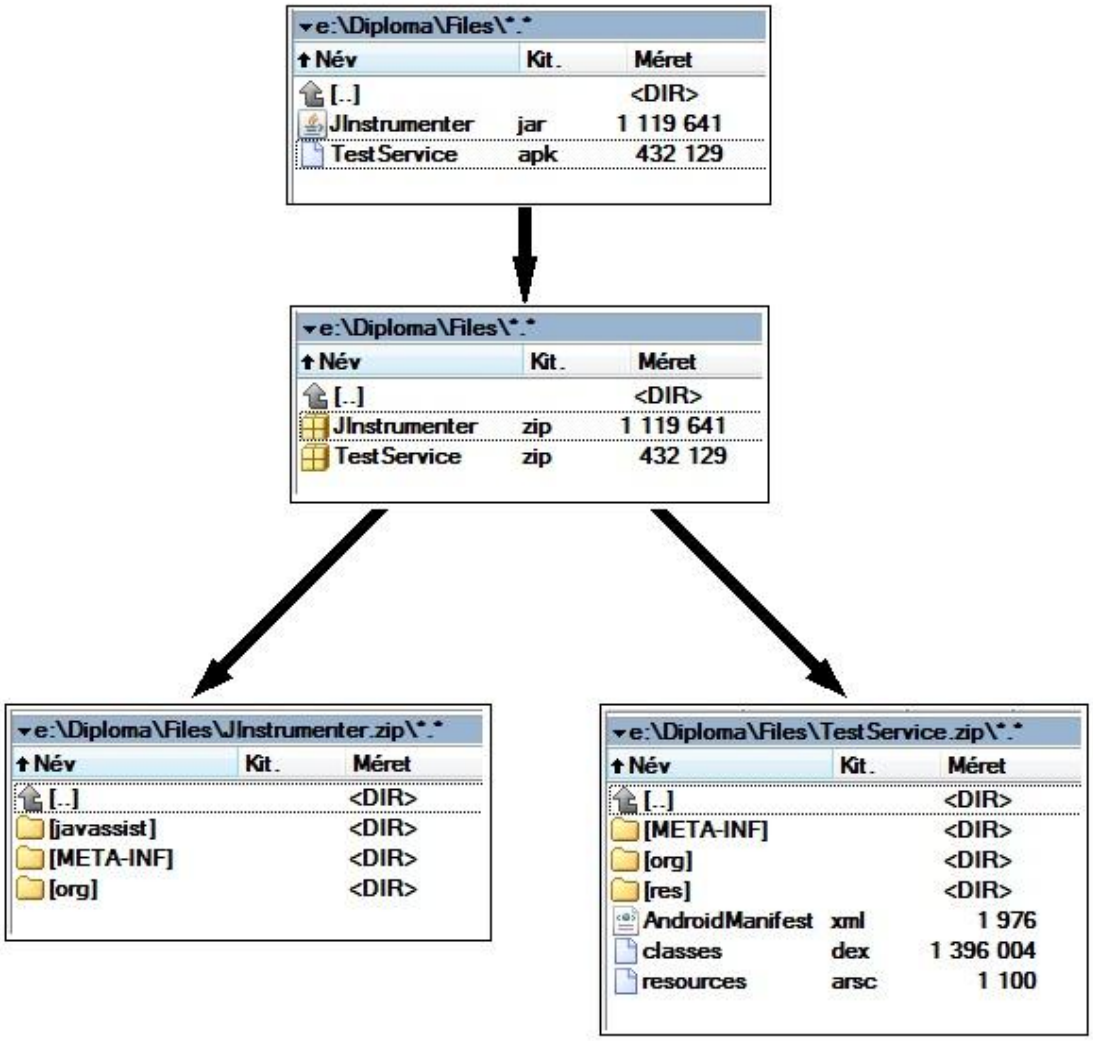
4.1. Kutatás

A feladat lehet, hogy első ránézésre nem tűnik túl bonyolultnak, mégis sok nehézséget rejt magában. Első lépésként szükséges volt felmérni és kiértékelni a beágyazott rendszerek tesztelésének legkorszerűbb módszereit és lehetőségeit, különös tekintettel a BlackBox és a WhiteBox módszerek kombinált alkalmazására. Ebből a felmérésből elkészült egy technikai riport [4], melyben kifejtettük megállapításunkat, miszerint a beágyazott rendszerek tesztelésének területén nincs nagy előrehaladás a kombinált technikák alkalmazásában, részint a 3. fejezetben kifejtett technikai nehézségek miatt: nincsenek megfelelő interface-ek és eszközök a feladat hatékony elvégzésére. Ráadásul a beágyazott rendszereken futó szoftverek nagyon speciálisak, ezért érthető módon a gyártók, fejlesztők előszeretettel nem adják ki a forrást a berendezéssel együtt. Ám ez a kisebb nehézség, hisz léteznek olyan alacsony szintű technikák, melyek nem igényelnek forráskódot, csak annak egy alacsonyabb szintű reprezentációját, például a gép által már értelmezhető, lefordított, bináris kódot. Technikailag ez a bináris kód található meg a berendezések memóriájában, ám itt jön a következő nehézség. Ez a bináris nem minden esetben olvasható ki a készülékből, vagy mert a gyártó levédte, nehogy azt jogtalanul felhasználják más berendezésekben, vagy mert a berendezés nem rendelkezik megfelelő szoftveres inteface-szel, perifériával vagy csatlakozási felülettel, hogy egy általános célú számítógépet rácsatlakoztatva hozzáférjünk a tárolt elemekhez. Találhatóak ugyan speciális kiegészítő eszközök, a korábban említett debuggerek, melyekkel képesek vagyunk olvasni a beágyazott rendszerek memóriáját, de ezek sem alkalmasak a kiolvasott kód lemásolására vagy egy módosított kód memóriába való visszaírására.

Kutatásom első lépéseként tehát meg kellett ismerkednem a beágyazott rendszerek tesztelésének jelenlegi állásával. Fel kellett kutatnom, hogy lehetséges-e a beágyazott rendszereken strukturális teszteket futtatni és rajtuk lefedettséget mérni. Arra az álláspontra jutottunk többek között a [23],[24],[25],[26],[27],[28],[29] jelzetű tanulmányok és cikkek alapján, hogy a vállalkozás nem lehetetlen: némi segédprogram segítségével a nem védett memóriájú berendezésekből kinyerhető a bináris kód, viszont

speciális, nem rögzített programú berendezésre lenne szükség ahhoz, hogy a vizsgálatra, mérésre előkészített változatot vissza is tudjuk írni, vagy pedig szükségünk lenne egy testboard-ra, amelynek vezérlőrendszere hasonlatos a tesztelni kívánt berendezéshez, és amelyre feltölthetjük a kinyert binárist, és azon esetleges kimenet-szimulációs kiegészítésekkel működésre is tudjuk bírni azokat.

A hagyományos beágyazott rendszerek tesztelésének kutatása végén az eredményeket látva a CIRENE projektben a Szegedi Tudományegyetemmel együtt dolgozó társszervezet, az Újvidéki Egyetem felajánlotta egyik kísérleti berendezésük használatát, mely a hagyományos Set-Top Box-októl eltérően nem egy speciális célszoftvert, hanem Android operációs rendszert futtat. Ez rendkívüli mértékben megkönnyítette a dolgunkat, ugyanis, mint ahogy azt a korábbi fejezetben említettem, az Android egy Java alapú, nyílt forrású rendszer, melyhez akárki fejleszthet alkalmazásokat, és minden alkalmazásnak rendelkezésre áll a bináris kódja a program ’apk’ kiterjesztésű telepítőfájljaként. Ezeket a binárisokat kezelni tudjuk hagyományos Java-s eszközökkel, mint például a Javassist [8], amely egy bytecode-manipulációt segítő osztálykönyvtár, és amely lehetővé teszi Java osztályok akár futás közbeni és JVM-be (Java Virtual Machine, Java Virtuális Gép; ez futtatja a Java binárisokat, nem konkrétan a fizikai gép) töltés közbeni módosítását, bővítését. De mivel az apk fájlok nem általános Java bináris csomagok, hanem Androidra optimalizált, és az Androidon üzemelő Dalvik Virtuális Gépre átalakított csomagok, így azokat előbb át kell alakítani standard Java csomagokká. Szerencsére az Android és a Java fejlesztői hamar belátták ennek a műveletnek a szükségességét és előnyeit, és közreadtak egy apktool [9] nevű eszközt, amely az Androidos apk fájlok ki- és becsomagolását teszi lehetővé, és egy dex2jar [12] nevű eszközt, amely az apk fájlokban található bináris csomagot, a dex kiterjesztésű fájlt Java-csomagokká tudja konvertálni és vissza. Egy standard Java csomag, azaz egy JAR fájl nem más, mint egy speciális tömörített állomány, amely az adott Java alkalmazás futásához szükséges binárisokat és meta-információkat tartalmazza. Erről úgy is megbizonyosodhatunk, ha a JAR fájl kiterjesztését átírjuk például zip-re, akkor bármilyen fájlkezelő meg tudja nyitni, és láthatjuk a konkrét tartalmát. Az apk fájl ehhez teljesen hasonlatos: ugyanúgy tartalmazza a binárisokat és a meta-információkat, és az is zip-re átnevezve megnyitható és megtekinthető. Az alábbi 4.1.1. ábra ezt szemlélteti:



4.1.1. ábra: A JAR és az apk fájlok kibonthatósága

A különbség annyi, hogy az Androidos binárisok már módosítottak, Dalvik gépre optimalizáltak. De az apktool segítségével ezek a dex kiterjesztésű fájlok visszaalakíthatóak hagyományos JAR-fájlokká, és a továbbiakban úgy kezelhetőek, mint bármely más Java binárisok. Ugyanez az eszköz használható a módosított JAR csomagok visszaalakítására dex-ekké. Ez a módszer igen kézenfekvőnek és egyszerűnek mutatkozott a kutatás által feltárt további megoldásokhoz képest.

Ez után jött a kérdés, miszerint hogyan határozhatunk meg különböző jellemzőket és metrikákat a kiválasztott beágyazott rendszeren futtatott tesztesetekről. Az eddig tárgyaltakból világosan látszik, hogy strukturális tesztelést kell végeznünk a berendezésen, hogy információhoz jussunk. Ám a tesztek végrehajtása nem elégséges, hisz azok önmagukban nem képesek a rendszerre vagy épp önmagukra jellemző

adatokat szolgáltatni. Valamilyen előkészítő folyamatra van szükség, melyben meghatározzuk a szolgáltatott információkat és azok továbbítási módját. A kutatások során számos lehetséges megoldásra bukkantam, melyek közül néhány lehetséges módszert alaposabban meg is vizsgáltam. Ezek közé tartozik a futtatókörnyezet módosítása: magába a futtatókörnyezetbe, jelen esetben a virtuális gépbe építenénk be az információgyűjtést és a kommunikációt. Ez elég hatékony módszer, ipari keretek között, ahol nap mint nap számtalan eszközt kell tesztelni. A CIRENE projektben elkészítendő prototípushoz ez túl nagy energia befektetés lett volna, és nem is állt rendelkezésünkre megfelelő erőforrás a művelethez. A lehetséges módszerek között szerepelt a korábban tárgyal debugolás is. Ebben az esetben a debugger szoftvert kell kiegészíteni, vagy egy középréteget (middleware) kell beépíteni a debugger és a berendezés közé, amely naplózza a kommunikációt, a kódot, vagy bármilyen adatot, amely keresztülfolyik rajta. Ez szintén költséges és nehézkes feladat. Nem ezen az úton folytattuk a gondolatmenetet. Ám már a fent említett, beágyazott rendszerek tesztelésére vonatkozó felmérés közben is és az új kérdésben végzett kutatás során sokszor megjelent az instrumentálás kifejezés, mint az információgyűjtést előkészítő lépés (például [27]). Az instrumentálás folyamata nem más, mint az eredeti forráskódok kibővítése olyan utasításokkal, amelyek futás közben információt tudnak szolgáltatni a rendszerről, annak állapotáról, vagy egy összetettebb folyamat részeként valamely jellemző metrika kiszámításában játszanak szerepet. Ettől kezdve erre a lehetőségre irányítottuk a figyelmünket. A cikkek és tanulmányok mellett segítséget nyújtott, hogy a Szegedi Tudományegyetem Szoftverfejlesztés Tanszékén nagy hagyománya van az instrumentálásnak, így nem csak írásokkal, de konkrét eszközökkel is rendelkeztünk, melyek közül egyet kissé átalakítva már alkalmazni is lehetett a projektben. Instrumentálni viszont több mindent lehet; itt is szükséges volt némi kutatással megelőzni a tervezési és kivitelezési folyamatokat. Többek között lehetséges a Java Virtuális Gép, vagyis annak egy cserélhető, módosítható része, a hozzá tartozó middleware instrumentálása. Ez egy jó módszer, mert nem a teljes virtuális gépet kell módosítani, csak annak egy kis részét. Sajnos ez sem tartozik a legegyszerűbb feladatok közé, hisz az interneten fellelhető nyílt forráskódú middleware-ek nagyon összetettek, bonyolultak, és sokszor nem is hibátlanok. Ezek módosítása még költségesebb lett volna, mint egy saját middleware kifejlesztése, ami rengeteg időt és erőforrást

felemésztett volna. Nem volt rá lehetőség. Mégsem kellett az instrumentálás ötletét elvetnünk, mert egy másik elem, amit instrumentálni lehet, az maga a tesztelendő alkalmazás forráskódja. Ez egy viszonylag alacsony költségű, gyors folyamat, mely után a program saját magáról tud információkat gyűjteni. Ennek is van hátránya, de nem akkora, mint a korábbi módszereké: minden egyes tesztelni kívánt alkalmazást instrumentálni kell.

Így pedig, hogy a berendezés rendelkezésre állt, az információgyűjtés módszereit felsorakoztattuk, a következő feladat az információ átadásának meghatározása volt. Itt a kutatási fázist megelőzte egy ötletelés, mert az adattovábbítás módjainak összegyűjtése nem ördöngös feladat. A kutatás akkor következett, amikor már a konkrét kiválasztott kommunikációs csatorna használatának lehetőségeit és módjait kellett megismerni. Az ötletelés során előkerültek, mint használható adattovábbítási csatornák, a fájlok. Használatuk kézenfekvő és egyszerű. Egy megegyezés szerinti vagy egy általánosan elterjedt belső formátum használatával a feldolgozásuk bárki számára gördülékeny feladattá válik. A letárolt adatok bármikor elérhetőek, feldolgozásuk a tesztelési folyamattól különálló, így nem lassítja azt. Hátulütői is vannak a fájlok használatának: ha a tesztelt berendezésen akarjuk tárolni őket, akkor nagy körültekintéssel kell lennünk, hisz a beágyazott rendszerek kevés erőforrással rendelkeznek, és hamar kifuthatunk a memóriából, ráadásul ismét speciális eszközökre lehet szükség, hogy a tárolt adatokhoz hozzáférjünk. Ha leválasztható adathordozón rögzítjük az információt. akkor a mozgatásával nincs igazán problémánk. Ebben az esetben viszont a külső tároló elérésére és írására fordítandó idő az, ami a szűk keresztmetszetet adhatja. Jó és áthidaló megoldásnak tekinthető egy felcsatolt mappa használata, amely fizikailag egy távoli gépen található, de megfelelő protokollok használatával és hálózati kapcsolat esetén a beágyazott berendezés fájlrendszerében is elérhető, mint ha lokális mappa lenne. Így az adatok letárolása nem jár plusz költségekkel, és azok azonnal elérhetővé válnak a külső gépen is, a feldolgozásuk megkezdődhet az adott tesztvégrehajtás alatt. Egyetlen, ám elég súlyosnak, költségesnek tűnő momentum ebben a technikában, hogy folyamatosan élő hálózati kapcsolatot kell fenntartani a külső géppel, és a külső gépnek is biztosítani kell a tesztelés alatti folyamatos és hibamentes működését. Ha folytatjuk ezt a gondolatmenetet, miszerint egy külső szervert építünk be a rendszerbe, amelyhez hálózati kommunikáció segítségével jutnak el az adatok, és azokat már a szerveren futó

alkalmazások kezelik, de nem szeretnénk folyamatosan élő hálózati kapcsolatot fenntartani, akkor csak a hálózati kommunikációs protokollok betartására és a hálózaton küldhető csomagok felépítésére, valamint azok adott időközönkénti elküldésére kell figyelmet szentelnie a beágyazott rendszernek, ami kevesebb időt vehet igénybe, mint a külső meghajtó írása, és kevesebb hibalehetőséget rejthet magában, mint a folyamatos kapcsolat fenntartása. Itt viszont figyelemmel kell lennünk, hogy ne terheljük túl a hálózatot vagy a szervert a nagyon sűrűn küldött adatokkal, mert elveszhet egy részük, vagy a szerver összeomlásával akár a teljes addigi munkánk eredménye. Ám mivel nem minden beágyazott rendszer rendelkezik standard hálózati csatolófelülettel, így bizonyos esetekben további külső eszközök bevonására is szükség lehet, amelyek egy, a berendezésen megtalálható csatolófelületre kapcsolódni tudnak és az azon érkező adatokat továbbítani tudják a hálózatra. Felmerülhet a kérdés, hogy ha ilyen köztes berendezések léteznek, és egy ilyen bonyolultabb rendszert ki tudunk építeni, akkor miért nem egyszerűsítjük le a dolgot, és kötjük össze közvetlenül a beágyazott rendszert a külső géppel? Természetesen erre is találhatóak megoldások, melyek nagyban leegyszerűsítik a rendszert, és kevesebb költséggel járnak, mint az általános hálózati kapcsolat és kommunikáció. A hagyományos beágyazott rendszerek legtöbbjén megtalálható a soros kommunikációs port, amely még régebbi asztali számítógépeken is általános. Modernebb berendezések és asztali gépek már rendelkeznek USB-s csatlakozási lehetőséggel, amely még egy nagyságrenddel könnyebbé teszi a kapcsolódást és a kommunikációt. Esetleg vezeték nélküli kommunikációs felületekkel is rendelkezhetnek, mint például Bluetooth vagy Infra port. Ám ha közös csatolófelület mégsem található, ismét szerephez jutnak a külső átalakító berendezések.

További fejtörést okoz, ha az adatokat nagy részletességgel, finom felbontásban szeretnénk tárolni, akkor rengeteg helyet felemészt, viszont ha csak a legfőbb információkat tároljuk, akkor a későbbiekben nem biztos, hogy ki tudunk számítani minden származtatott metrikát, jellemzőt. Ha a letárolási és feldolgozási folyamat az tesztvégrehajtástól különálló folyamat, akkor tesztelés közben nem érhetőek el a feldolgozott adatok, így ha azok alapján valami módosítást szeretnénk végezni a tesztvégrehajtásban vagy a tesztbázisban, akkor azt csak két tesztelési fázis között tudjuk elvégezni, ezzel kissé lelassítva és redundánssá téve a tesztelési folyamatot.

Mivel a fizikai lehetőségeink igen korlátozottak, nagy hangsúlyt kellett fektetni a begyűjtött információkra. Mivel a fő cél lefedettségi információt gyűjteni, és ennek segítségével a funkcionális tesztesetek bázisát szűrni és kiegészíteni, így azt kellett megtalálni, hogyan tudunk a legegyszerűbben, direkt vagy származtatott módon lefedettségi adatokhoz jutni. Egy közismert módszer ehhez a nyomon követés (tracing) [31]. Ez egy speciális logolási módszer, mellyel követni tudjuk a tesztelt program futását. Ha nem vagyunk speciálisabb dolgokra kíváncsiak, és a beágyazott rendszer szoftvere jól lett implementálva, akkor rendelkezik általános és debug logokkal, melyeket egyébként is folyamatosan szolgáltat a rendszer futása alatt. Ezekből a logokból csak bizonyos speciálisabb blokkok érintését tudjuk kinyerni. Ha más szintű vagy bővebb információra vagyunk kíváncsiak, akkor elő kell készítenünk a szoftvert, például instrumentálással, hogy az igényeinknek megfelelően viselkedjen. A következő tracing szintek a legelterjedtebbek: útvonal szintű tracing esetén a programban az adatok által bejárható útvonalak nyomon követése a cél. Ez viszonylag magas szintű metrika, nem ad a kód struktúrájáról, lefedettségéről megfelelő információt. Hasonlóan magas szintű az elágazások érintésének naplózása. Ebből az információból már jobban tudunk dolgozni, de még ennél a metrikánál is igaz, hogy a teljes kódra nem enged rálátást. Az alapvető blokkok, azaz a kapocszárójelek közötti részek figyelése már megközelíti a szükséges szintet, de még mindig nem kapunk információt az egyes blokkokon belüli metódusokról, utasításokról. Tovább haladva lefelé a szinteken eljutunk a metódusokhoz. A tracing ezen a szinten már elég részletes adatokkal szolgál a programkódról, hisz bizonyos paradigmák a metódusokat tartják a programozás alapegységeinek, és ha egy program a futása során érinti az összes benne található metódust, akkor azt optimálisan kivitelezettnek tekinti. Sőt, vannak paradigmák, melyek egy szinttel magasabbra, az osztályok és objektumok szintjére helyezik azt. Osztályokat is lehet nyomon követni, de az a metódusok nyomon követésénél kevesebb információval rendelkezik, ráadásul a beágyazott rendszerek világában nem jellemző az objektum-orientált szemléletmód. Legalacsonyabb szinten van a lehető legtöbb információval szolgáló, kódsor szintű tracing. Ez azt jelenti, hogy minden egyes kódsor meg van jelölve, és végrehajtása során a jele bekerül a logfájlba. Mivel minden sor futásához hozzá tartozik egy fájlírás vagy egy adatküldés, ezért ez a szint nagyban lassítja a rendszer működését. Csak a legszükségesebb esetben alkalmazzák.

A tracing által szolgáltatott lefedettségi információt sokféle módon tudjuk hasznosítani. Önmagában is elegendő a tesztesetek halmazának kibővítésére, hisz ha összevetjük az eredeti forrással, akkor megvilágíthatjuk annak olyan területeit, amelyeket eddig nem érintett egyetlen teszteset sem. Alkalmas még a tesztesetek szűrésére, sorba rendezésére is, hiszen ha tesztesetenként külön-külön tároljuk el a lefedettség mértékét, akkor ki tudjuk válogatni a tesztesetek közül azokat, amelyek a legnagyobb részét fedték a kódnak, vagy épp sorrendet tudunk felállítani ez az információ alapján. Ám ha a lefedettség-mérést ügyesen valósítjuk meg, akkor nem pusztán annyi információnk lesz, hogy mennyi programkód-elemet érintettünk, hanem az is, hogy az egyes programkód-elemeket hányszor érintettük. Így egyetlen méréssel kétféle információt nyertük: hogy mekkora részét fedtük a kódnak (vagy a választott szintnek megfelelő programelemnek), és hogy az egyes tesztesetek milyen hosszú utat jártak be a programkódban futásuk során. Még egy kis kiegészítéssel a nyomvonalat is rögzíteni tudjuk, ha szükségünk van rá. Így már különböző adatok alapján, vagy akár a kombinációjukból létrehozott új adatok alapján tudunk műveleteket végezni a teszthalmazon, vagy ezeket az információkat felhasználhatjuk a tesztelt kód jellemzésére, további metrikák és jellemzők kiszámítására. Például fontos származtatott jellemzői a kódnak, melyek mind a fejlesztőknek, mind a tesztelőknek nagy segítséget nyújthatnak, a traceability linkek [32]. A traceability a program fejlesztésének különböző fázisaiban létrejött programelemek és -szintek összekapcsolását jelenti. Segítségével megmutatható például a rendszer specifikációjában megfogalmazott követelmények, funkcionalitások és a programkód egyes részei közötti összefüggés.
A CIRENE projekt keretein belül nem volt szükség további ismeretanyag vagy terület felkutatására, a tervezett elemek megvalósításához megfelelő mennyiségű és minőségű információval rendelkeztünk eddigre.

4.3. Tervezés

Az előző fejezetben részletesen kifejtettem a kutatási területeket és az ott tapasztalt eredményeket, lehetőségeket. Ezek alapján továbbléphettünk a projekt következő fázisába, a tervezésbe. Mint ismert, a CIRENE projekt beágyazott rendszerek tesztelési minőségének fejlesztését tűzte ki célul, s ezen a területen eredményeket is ért el. A

kezdeti felmérés eredményéből jól látszott, hogy nem létezik még egy jól megtervezett, általános módszertan a beágyazott rendszerek tesztelésére, mely önmagában is megállja a helyét az elméleti területeken, és szükség esetén megfelelő specializációval a gyakorlatban is. Továbbá kiderült, hogy a beágyazott rendszerek tesztelésében nem jellemző a kombinált eszközök használata, mint amilyen a GreyBox is. A CIRENE projektben eltökéltük, hogy ezen változtatunk, és nekiveselkedtünk, hogy egy bármiféle beágyazott rendszeren alkalmazható kombinált tesztelési módszertant megtervezzünk, kivitelezzünk, majd egy alkalmas eszközön kísérleti céllal meg is valósítsunk. Ebben a fejezetben az általános módszertan megtervezésének lépéseit, nehézségeit és eredményeit fogom részletezni. A módszertan elméleti diagramját a 4.3.1. ábrán szemlélhetik meg.

A kezdeti felmérés a beágyazott rendszerek tesztelési módszereiről, szokásairól, módszertanairól szólt, így ezekből össze sikerült gyűjteni (kis kiegészítő kutatás segítségével) az összes olyan elméleti és gyakorlati momentumot, amely a beágyazott rendszerek tesztelésében kulcsfontosságú szerepet játszik. Ezek szükségszerűsége lett az alap a tervezés kezdetén. Mivel a tapasztalatok azt mutatták, hogy a célrendszert többnyire csak funkcionálisan tesztelik, ezért összegyűjtöttük a tervezett módszertan számára fontos elemeket, amelyeket a funkcionális tesztelés szolgáltathat.

Egyik ilyen elem a tesztesetek generálása, ugyanis a berendezés tesztelésére szolgáló teszteset-halmazt a berendezés fejlesztése során készült dokumentációkból, legjobb esetben a fejlesztés első lépéseként létrejött követelményspecifikációból állítják elő. A generálás lehet manuális vagy automatikus. Manuális generálást legtöbbször szakképzett tesztelők végzik a kiadott dokumentációk alapján. Közepes mértékben hosszú és költséges folyamat, de elengedhetetlen, főleg a beágyazott rendszerek területén. Automatikus teszteset-generáláshoz szükséges a követelményspecifikáció valamilyen, számítógép által értelmezhető reprezentációja, vagy ha a tesztelést később, a berendezés vagy a szoftver elkészülte után kezdik meg, akkor a termék egy absztrakt reprezentációja. Ez leggyakrabban valamilyen gráf vagy fastruktúra, amelyet az automatikus teszteset-generáló be tud járni, és fel tudja építeni a teszteset lépéseit. A bemeneti és az elvárt értékek halmazát is meg kell adni, így a generált tesztesetek a lehető legnagyobb mértékben képesek fedni a rendszer funkcionalitását, a bementi adatok szélső- és kritikus értékeire külön figyelmet fordítva. Az így előállított

teszteseteket a generáló definíciójától függően manuálisan vagy automatikusan is végre lehet hajtani.

A másik elem, amit a funkcionális tesztelés szolgáltat, az a funkcionális lefedettség. A követelményspecifikációból kinyerhető egy funkcionalitás-lista, amit a fejlesztett rendszernek meg kell valósítania, és amely alapjául szolgál a rendszer tesztelésének. A funkcionális tesztelés célja ugyanis leellenőrizni, hogy a rendszerben megvalósított algoritmusok megfelelnek-e a követelményspecifikációban leírt működéseknek. Ehhez minden egyes elemhez a funkcionalitás-listán készítenek egy vagy több tesztesetet, és végrehajtják azokat. Ha az összes teszteset, vagy legalább is minden funkcionalitáshoz elegendő számú teszteset sikeresen lefut, akkor a rendszert késznek tekintik. Ez a módszer viszont semmilyen információval nem szolgál a rendszer működéséről, belső felépítéséről, hatékonyságáról. Itt jön a képbe a WhiteBox módszer, amely pótolni képes ezeket a hiányosságokat, és meg tudja mutatni, ha a rendszerben strukturális hibák vagy nem megfelelő, nem optimális megvalósítások vannak. A strukturális tesztelés által a módszertanunkhoz adható elemek bemutatása következik.

A legalapvetőbb elem a lefedettségi információ. Ez a magja minden lehetséges módosításnak, hiszen ez adja meg az egyes tesztesetek jellemző metrikáit, amelyek alapján rangsorolni tudjuk őket, vagy ki tudunk választani belőlük egy szükséges és elégséges részhalmazt. Az általános módszertanban figyelembe kellett venni minden lehetséges technikát, amellyel meg lehet szerezni ezt az információt. Ezeket a dolgozatom korábbi részeiben már bemutattam, most felsorolom őket előnyeikkel és hátrányaikkal, valamint néhány kiegészítő információval.

A kód-instrumentálás egy egyszerű, alacsony szintű megoldás, amely nem igényel nagy erőforrásokat. Különböző szintű lefedettségi adatokat lehet vele kinyerni a futtatott rendszerből attól függően, hogy mire van szükségünk. Különböző megoldásai vannak: legegyszerűbb a forráskód-instrumentálás, amikor a fejlesztés alatt álló vagy már elkészült program forráskódját bővítjük ki a megfelelő utasításokkal. ez nem igényel speciális eszközöket, az eddig használt fejlesztőkörnyezet tökéletesen alkalmas a végrehajtására, de automatizált szkripteket is készíthetünk, melyek pillanatok alatt végigfutnak a forrásfájlon és a megfelelő helyeket mintaillesztéssel felismerve elhelyezik a szükséges utasításokat. Hátránya ennek a módszernek, hogy szükséges hozzá a program forráskódja, amit harmadik féltől származó szoftver esetén kevés

eséllyel tudhatunk magunkénak. Másik megoldása a bináris instrumentálás. Ez nehezebb, de beágyazott rendszereknél és forráskód nélküli szoftvereknél ez a használható módszer. A bináris kód a szoftver forráskódjának lefordított, gépi kódú változata, amely futtatható a kívánt berendezésen vagy emulátorban. Akadnak eszközök, melyek képesek értelmezni ezt a bináris kódot és felismerik bennük az egyes programkód-elemeket, mint például a metódusok fejléceit vagy a változódefiníciókat. Ezek az eszközök alkalmazhatóak a bináris instrumentáláshoz.

Következő szinten szerepel a tesztelőrendszer és a berendezés közé építhető középréteg, middleware, amely úgy alakítható ki (vagy ha a tesztelőrendszerhez tartozik middleware, akkor úgy módosítható), hogy az szolgáltasson megfelelő információkat, hiszen azon minden adat átfolyik a tesztelés közben. Beleintegrálva egy binárist értelmező eszközt és kiegészítve a szükséges szelektálási és letárolási metódusokkal könnyedén adatnyerővé alakíthatjuk. Előnye, hogy egyszer kell elkészíteni, egyetlen elemet kell módosítani, és korlátlanul használható. Viszont egy middleware megtervezése, kivitelezése, de még a módosítása is gyakorlott fejlesztőket és rengeteg erőforrást kíván.

Egyel magasabb szintű adatnyerési megvalósítás a futtatókörnyezet módosítása. Ilyen futtatókörnyezet például a Java Virtuális Gép, amely bármilyen berendezésen működhet, és ami a Java nyelven írt programok futtatását végzi, megvalósítva ezzel egy bizonyos mértékű platformfüggetlenséget. Módosítani egy ilyen rendszert nagy vállalkozás, bár a végeredménye igen kifizetődő lehet. Egy módosított futtatókörnyezetet bármilyen rendszerre telepítve számtalan féle alkalmazásból nyerhetnénk ki adatokat. Természetesen, mint ahogy a middleware esetén is, egy virtuális gép nagyon összetett rendszer, módosítása nagy szakmai gyakorlatot kíván ezen a speciális területen. A CIRENE projekt keretein belül nem volt sem szükség, sem lehetőség ilyen nagy volumenű beruházásra, vállalkozásra.

Korábban említettem a debugolás lehetőségeit, mind hardveres, mind szoftveres úton. Program fejlesztése során a szoftveres debug a jellemzőbb, hisz az szinte minden fejlesztőkörnyezetben beépített funkció, így alkalmazása nem okoz nehézséget, akár már fejlesztés közben is. A szoftveres debuggerek szintén látják a teljes kódot, mind forráskód, mind bináris szintjén, így kézenfekvő megoldás lehet ezen szoftverek kiegészítése, esetleg beépülő modul (plug-in) fejlesztése hozzájuk, mely lehetővé teszi

adatok kinyerését debug folyamatok közben. Ez nem túl költséges megoldás, és nem szükséges hozzá túl nagy szakmai tapasztalat sem. Viszonylag könnyen kivitelezhető, csak meg kell ismerni az adott rendszer fejlesztési nyelvéhez tartozó általánosan használt és nyílt forráskódú, vagy csak bővíthető fejlesztőkörnyezetet, és el kell végezni a szükséges módosításokat. Hátránya inkább az időigényében van, mert megismerni egy teljes, összetett fejlesztőkörnyezetet, annak paradigmáit és szokásait igen hosszadalmas és fáradságos.

Végül, de nem utolsó sorban a hardveres debuggerek beágyazott rendszereknél való elterjedt alkalmazása szintén kézenfekvővé teszi annak felhasználását információ kinyerésére. A hardveres debugger olyan berendezés, amely kapcsolódik a beágyazott rendszerre és a személyi számítógépre egy időben. Saját szoftverkörnyezettel is rendelkezhet, amely segítségével meg tudja jeleníteni a berendezésen futó folyamatokat, azok állapotát, bináris kódját. Ezzel a szoftverkörnyezettel bizonyos esetekben bele is tudunk nyúlni a programok futásába, töréspontokat tudunk elhelyezni bennük, léptetni tudjuk a folyamatokat és nyomon tudjuk követni azokat lépésről lépésre. Ennek a szoftverkörnyezetnek a kiegészítése, vagy egy másik hardvereszköz beiktatása lehetővé tenné a szükséges adatok fellelését, kinyerését és letárolását. Ám ezek a szoftverkörnyezetek biztonsági okokból igen zártak, semmi féle módosítást vagy kiegészítést nem tesznek lehetővé a gyártó szellemi tulajdonának védelme érdekében.

A fentiek alapján a módszertan elméleti folyamatában a funkcionális és strukturális elemeknél kapcsolódási pontokat lehetett felfedezni és definiálni, melyek egymás erősítésére váltak. Az egyik ilyen kapcsolódási pont a lefedettségi adatok manuális meghatározása és rögzítése. Ez főleg a funkcionális lefedettség kiszámításánál lehetséges, mert ott nincs akkora elembázis, amit figyelemmel kellene kísérni. Apróbb kódok esetén a metódus- és magasabb szintű lefedettségek is meghatározhatóak manuálisan. Gyakoribb eljárás azonban a manuális visszaellenőrzés, amikor is a futás során előállt lefedettségi adatokat kézzel vetjük össze a kóddal, és így keresünk nem érintett kódrészleteket. A funkcionális kimenetek eredményét is figyelembe véve a kódban könnyen behatárolhatjuk a hibák hollétét. Ehhez szorosan kapcsolódik a másik közös pont, a traceability, vagyis a programok fejlesztése során előálló különböző szintű termékek közötti kapcsolatok meghatározássa. Ezzel tudjuk összekapcsolni a teszteseteket az általuk érintett forráskód-részletekkel, a követelményspecifikációban

szereplő kívánalmakat a tesztesetekkel, és így a kívánalmakat az őket megvalósító programkód-elemekkel. A traceability egy nagyon hasznos és fontos tulajdonság. Segítségével a fejlesztőknek pontos adatokkal tudunk szolgálni a hibák helyéről és jellegéről, valamint a hozzá kapcsolódó követelményekről és a hibát kiváltó felhasználási lépéssorozatokról (ami maga a teszteset).

Egy fontos és előremutató kapcsolódási pont a funkcionális és a strukturális elemek között az irányított random tesztelés. A random tesztelés egy speciális teszteset-gyártási módszer, amely véletlenszerűen választja mind a metódusokat, mind a bemeneti értékeket a tesztesethez. Így viszonylag hamar fényt tudunk deríteni a legsúlyosabb működési hibákra, viszont ára is van neki. Egy idő után nagy lesz a tesztesetek átfedése, vagyis nem ellenőriznek eltérő részeket a forrásból. Ez kiküszöbölhető egy irányító elem bevezetésével, amely figyelemmel kíséri a tesztesetek gyártását és futtatását, és ha a módszer egy redundáns tesztesetet készít, azt elveti, mielőtt lefuttatná. Ez az irányító elem visszajelzéseket is küld a tesztelőknek a futások eredményei alapján, mert a hibával záródó tesztesetek által érintett kódrészletek között nagy a hiba valószínűsége.

Végül talán a legfontosabb elem, amely a két módszer összevegyítéséből született: a priorizálás és a szelekció lehetősége. Az adatgyűjtési folyamatok által szolgáltatott információkat és a belőlük készített származtatott adatokat hozzá tudjuk rendelni az egyes tesztesetekhez, ezzel rájuk jellemző metrikákkal ruházva fel őket, amelyek alapján sorrendet tudunk felállítani közöttük, és ki tudjuk közülük válogatni azt a minimális elemszámú teszteset-halmazt, amely szükséges és elégséges egy adott kritérium kielégítéséhez. A teljesítendő kritériumok meghatározzák, hogy milyen jellemző adat alapján válogatjuk vagy rendezzük sorba a teszteseteket.

A tervezés eddigi elemei elégnek bizonyultak az általános módszertan felállítására. A módszertan diagramként ábrázolva megtekinthető a 4.3.1. ábrán a mellékletek között, valamint a [33] jelzetű dokumentumban. Az ábra egy tömör magyarázatát itt is megejtem; részletes magyarázat található a fent említett dokumentum 15. oldalán. Az ábrán szereplő téglalap alakú dobozok folyamatokat, a hullám alakú dobozok adathalmazt jelölnek. A nyilak az adatáramlás irányát jelölik. Az ábrán szereplő folyamatok:

Test case generation: a specifikációból automatikusan teszteseteket generáló algoritmust jelöli. Kiegészítő bemeneti adatként megkapja a nem fedett kódrészletek listáját és a funkcionális lefedettségi adatokat, hogy szükség szerint új teszteseteket készíthessen.

Prioritization/Selection: a már többször említett sorba rendezési és kiválogatási algoritmust jelöli. Bemenetként a lefedettségi metrikák és a korábbi futások eredményei adottak, melyekből definiálható a műveletekhez kritérium.

Test execution: a tesztesetek végrehajtásáért felelős rendszert jelöli. Automatikus tesztvégrehajtás és információkinyerés a feladata.

Not covered code calculator: a forráskódból és a lefedettségi adatokból meghatározza az adott tesztkör által nem érintett kódrészleteket.

Traceability computation: azt a folyamatot jelöli, amely rögzíti a tesztesetekhez a futásuk során érintett útvonalat. Továbbá képes előre meghatározni egy elvárt útvonalat az adott tesztesethez, ha adott a másik két bemenete, a funkcionalitás és a kód közötti, valamint a funkcionalitás és a teszteset közötti kapcsolat. A számított és a rögzített útvonal különbségét jellemző adatként rögzíti.

Functional coverage computation: ez a folyamat képes meghatározni a lefedettségi és a traceability információkból a teszteset funkcionális lefedettségét.

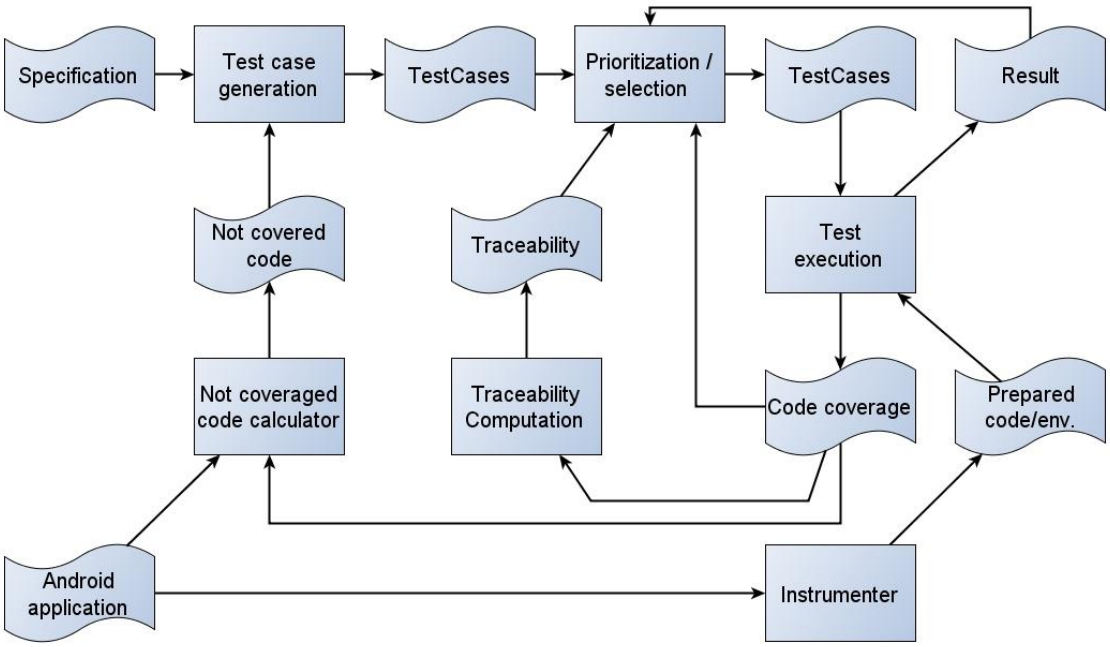
Instrumenter: a forráskódot, a binárist, a middleware-t vagy a futtatókörnyezetet módosító eszköz. Előkészíti a választott szintű elemet az információgyűjtésre.

Reliability computation: a tesztelt rendszer megbízhatósági metrikáját számítja ki lefedettségi és egyéb adatok alapján.

Az általános módszertan kifejlesztéséhez szükséges kutatások és tervezések eredményeit, valamint a kész módszertant magát a [33],[34] jelzetű dokumentumok tartalmazzák.

A CIRENE projektben a Szegedi Tudományegyetem Szoftverfejlesztés Tanszékének társintézménye volt az Újvidéki Egyetem Műszaki Tudományok Kara, ahol jelentős mértékű beágyazott rendszer tervezés, fejlesztés és tesztelés folyik. Az ő javaslatukra és felajánlásukra döntöttünk úgy, hogy a kísérleti megvalósítást egy kísérleti berendezésen, egy Android rendszert futtató Set-Top Box-on valósítjuk meg. Elkészítettük hozzá a

specializált módszertant, melyről részletes beszámolót olvashatnak a [35] jelzetű dokumentumban. A specializáció során nem került megvalósításra az általános módszertan néhány eleme, és a fentebb felsorolt számos lehetőségek közül kiválasztásra kerültek az akkori helyzethez mérten legmegfelelőbbek. Az alábbi 4.3.2. ábrán látható a specializált módszertan diagramja. A jelölések és az folyamatok megegyeznek a 4.3.1. ábránál leírtakkal.



4.3.2. ábra: Az Android rendszert futtató Set-Top Box-ra specializált módszertan

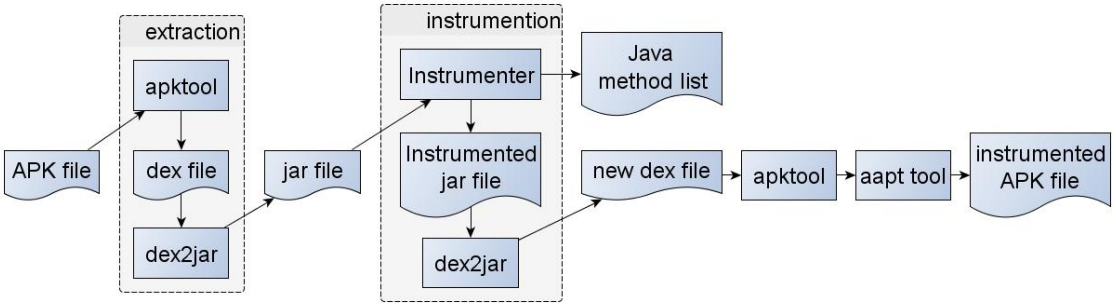
Kihagytuk az általános módszertanból a megbízhatósági metrika kiszámítását (reliability), mert nem állt rendelkezésre minden szükséges információ, és a kiszámítás bonyolultsága elhúzta volna a fejlesztést. Ráadásul jelen esetben nem kapott mérvadó jelentőséget az információtartalma. Nem került megvalósításra továbbá a funkcionális lefedettség mérése, mert azt ezen kívüli eszközzel is egyszerűen meg tudták határozni. A funkcionalitások és a kód, valamint a funkcionalitások és a tesztesetek kapcsolatát szintén külső eszközzel határozták meg, így a rendszerbe nne mvolt szükség ezek megvalósítására sem.

A következő fejezetben részletezem a megvalósított elemeket és megvalósításaik módját.

4.4. Fejlesztés

A fejlesztésben elsődleges fontosságú volt meghatározni a korábbi fejezetekben bemutatott számos lehetőség közül a jelen rendszerhez és lehetőségekhez legjobban illőket, felkutatni és megvalósítani a hozzájuk szükséges alkotóelemeket. Ezeket mutatom be alább.

Adatgyűjtési módszernek a bináris kód instrumentálására esett a választás, mert ahhoz egyrészt volt dokumentáció és szabadon felhasználható programvázlat a tanszéken belül, másrészt az Android rendszer nyílt forrású és szabadon bővíthető volta miatt a rajta futó programoknak elérhető a binárisa. Elhanyagolható többletköltségnek tekinthető az, hogy minden tesztelni kívánt alkalmazást egyenként kell instrumentálni. Az instrumentálás folyamatát a 4.4.1. ábra szemlélteti:



4.4.1. ábra: Az instrumentálás folyamata

Az Androidos alkalmazások apk kiterjesztésű bináris fájljai és a Java nyelven írt, futtatható, bináris programok jar kiterjesztésű csomagjai a korábban bemutatott módon hasonlatosak egymáshoz, és az Android fejlesztőkörnyezete lehetőséget is biztosít a csomagok közötti átjárásra a kiterjedtebb fejlesztői lehetőségekért. Ez a lehetőség biztosította ebben a fejlesztésben is az instrumentálást. Az apk fájlokat kicsomagolta az apktool [9], a benne található dex kiterjesztésű binárist, amely a Dalvik Virtuális Gépre optimalizált, futtatható bináris csomag, a dex2jar [12] Java csomaggá alakította, melyet a Javassist [8] segítségével létrehozott instrumentáló alkalmazás ellátott a megfelelő kiegészítésekkel. Ezután a dex2 jar eszköz jar2dex metódusának segítségével létrejött az új, instrumentált binárisokat tartalmazó dex fájl, melyet az apktool ismét Android számára felismerhető, arra telepíthető formára hozott. Szükség volt továbbá az alkalmazás újbóli aláírására, mert a kicsomagolás során az aláírás elvész. Erre szolgálnak az aapt [10] és a Jarsigner [11] nevű eszközök. Az aapt lehetőséget ad a

„ZIP-kompatibilis” bináris csomagok módosítására, a Jarsigner pedig egy adott kulcsfájl segítségével aláírást helyez el a fájlban.

Instrumentálás során az alkalmazás kigyűjti a tesztelt programban található összes metódus nevét egy kimeneti fájlba későbbi felhasználás céljából, és ellátja őket egy egyedi azonosítóval, konkrétan egy sorszámmal (*MethodId*). A program kódja kibővül egy CoverageCollector nevű osztállyal melynek egyetlen adattagja van, egy egészeket tartalmazó tömb, melynek annyi eleme van, ahány metódusa a tesztelt programnak van (*MethodNumber*):

```
static int[] coverage = new int[MethodNumber];
```

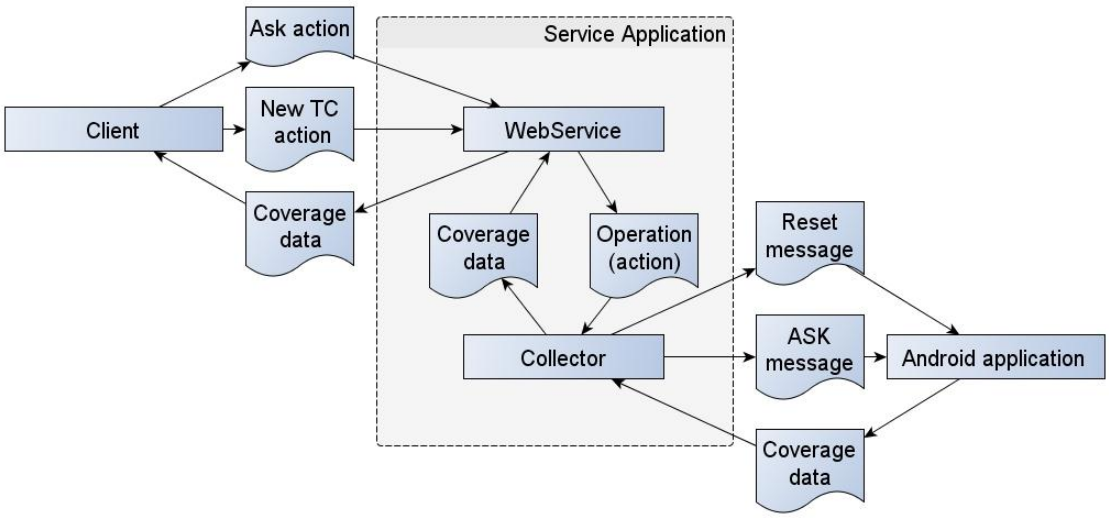
Ebben a tömbben történik a metódusok hívásainak számlálása úgy, hogy a metódusok első utasításaként bekerül egy inkrementáló lépés:

```
CoverageCollector.coverage[MethodId]++;
```

Az egyszerűség kedvéért a *MethodId*-k a gyűjtőtömb egyes indexei. Így meg tudjuk határozni tesztesetenként, hogy melyik metódusok futottak le, és hányszor. Ebből az információból a teszteset futásának végén kiszámítottuk a teszteset által bejárt útvonal hosszát, az általa lefedett kódrészlet százalékos mértékét, a lefedett és a nem fedett metódusok listáját. Az instrumentálás nem okozott észlelhető lassulást a programok működésében.

Így hát tesztesetenként rendelkezésre álltak az adatok. Következő lépésben azok továbbítását kellett megoldani, mert mint látható, a tesztelt programok magukban tárolják a lefedettségi adatokat, melyek megszűnnek a tesztelt program kilépésekor. Az Android rendszer jogosultságokkal dolgozik, így akár ha a berendezéshez tartozó külső tárolóra akarunk írni, akár a hálózaton szeretnénk kommunikációt végezni, engedélyekre van szükségünk. Nem lett volna nehéz még egy módosító alkalmazást létrehozni, amely megadja a szükséges engedélyeket az instrumentált alkalmazásoknak, és azokban egy kommunikációs osztályt is elhelyez, ám arra a következtetésre jutottunk, hogy a kommunikáció lehetőségét folyamatosan fenn kell tartani, így célszerű egyetlen, mindig futó alkalmazást készíteni, amely a háttérben figyeli a kommunikációs csatornát és szükség esetén üzenetváltásokat tesz, valamint tartja a kapcsolatot az instrumetnált alkalmazásokkal és szükség esetén begyűjti tőlük és elküldi a kliensek felé az adatokat.

Szerencsére az Android rendszer lehetőséget biztosít a háttérben folyamatosan futó alkalmazások létrehozására. Ezek az úgynevezett service alkalmazások.



4.4.2. ábra: A CollectorService alkalmazás felépítése és működése

A CollectorService alkalmazás pont a fentiekben leírt működést valósítja meg.

A kommunikációhoz egy egyszerű, ám platformfüggetlen protokollra volt szükség, így több lehetőség megvizsgálása után a JSON [37] rendszerre esett a választás. A JSON egy rendkívül egyszerű, szöveges alapú üzenetpárokra épülő kommunikációs protokoll, melyben az üzenetek formátuma kötött:

{„TAG” : „DATA” }

A *TAG* helyén megegyezés szerinti akcióüzenet állhat, amely meghatározza, hogy a *DATA* mezőben szereplő adattal mi történjen. A választást nagyban megkönnyítette, hogy a Java egész osztálykönyvtárat hozott létre JSON objektumok létrehozására kezelésére. Jelen implementációban a következőkben ismertetett üzeneteket definiáltuk.

Az instrumentált alkalmazások irányába küldött üzenetek és jelentésük:

Ask: az instrumentált alkalmazás elküldi az addig összegyűjtött adatot a service-nek.

Reset:az instrumentált alkalmazás kiüríti a gyűjtőtömböt, benne minden értéket nullára állít.

A kliensek irányából fogadott üzenetek:

NEWTC: egy új teszteset indulását jelenti, hatására a service elküldi a Reset jelet a tesztelt alkalmazásoknak.

ASK: a teszteset végét jelenti, hatására a service elküldi az Ask jelet a tesztelt alkalmazásoknak.

A service alkalmazás válaszai a klienseknek:

Összetett JSON objektumok, melyek tartalma a NEWTC jel esetén:

{ „timestamp” : „TIME” }

ASK jel esetén:

{ „timestamp” : „TIME”, „data” : { „NAME1” : [LIST1], „NAME2” : [LIST2], ... } }

Mindkét esetben a *TIME* adattag tartalmazza az aktuális rendszeridőt, a NAME# tagok jelölik az egyes tesztelt alkalmazásokat, a [LIST#] tagok pedig a hozzájuk tartozó adatokat. Így nem csak a fent említett adatok számíthatóak ki az egyes tesztesetekhez, de a tesztesetek végrehajtási ideje is, mint egy újabb jellemző metriak.

A rendszer a hálózati kommunikáción kívül lokális fájlokban tárolja az instrumentálás során kigyűjtött metódusneveket az azonosítóikkal és a tesztelési ciklus végén a teljes tesztelési fázisra vonatkozó, tesztesetenként és alkalmazásonként lebontott eredménytáblázatot az összes jellemző adattal.

A tesztesetek végrehajtására szintén az Újvidéki Egyetemtől kaptunk eszközt, egy automatikus funkcionális tesztelő eszközt, az RT-Executort [14]. Rendkívül egyszerű szöveges formátumban tárolja a teszteseteket, melyek így a programon kívülről is egyszerűen módosíthatóak. Ehhez az eszközhöz készítettünk egy beépülő modult, amely kommunikál a CollectorServicce-szel, jelzi a tesztesetek kezdetét és végét, valamint feldolgozza a kapott adatokat, és letárolja a végső kimeneti fájlban.

Az általunk szolgáltatott információk felhasználásával az Újvidéki Egyetem Műszaki Tudományok Karának dolgozói elkészítettek egy priorizáló és szelektáló eszközt, mely a specializált módszertan ábráján is megtalálható. Ezt az eszközt és módszertant azóta is használják munkájukban az automatikusan előállított regressziós teszthalmazaik méretének csökkentésére.

4.5. Kiértékelés / Eredmények

A dolgozatom alapját képező CIRENE projektben végzett kutatás és fejlesztés eredménye egy beágyazott rendszereken általánosan használható, funkcionális és strukturális elemeket kombináló tesztelési módszertan.

A témából egy felmérés, több beszámoló és cikk készült, valamint egy hosszú távú tervezet, melyben diploma- és doktori témák is szerepelnek.

A projekt egy kiemelt szeletéből, a priorizáló és szelektáló részből az Újvidéki Egyetem egy mester szakos hallgatója elkészítette diplomadolgozatát, melyet sikeresen meg is védett.

A projekt keretei között elkészült egy kísérleti megvalósítás Android rendszert futtató Set-Top Box-on.

Az általunk szolgáltatott információk felhasználásával az Újvidéki Egyetem Műszaki Tudományok Karának dolgozói elkészítettek egy priorizáló és szelektáló eszközt, melyet éles ipari környezetben alkalmaznak.

5. AZ EREDMÉNYEK UTÓÉLETE: ÚJ PROJEKTEK AZONOS ALAPOKON

A módszer általánossága és az Android rendszer kiterjedt használata lehetővé teszi az itt elért eredmények továbbfejlesztését és átalakítását, finomhangolását más rendszerekre.

Egy ilyen finomhangolásra és átalakításra volt szükség egy nemrég zárult projektben, ahol egy Android rendszert futtató pénztárgép fejlesztésének elősegítésére kellett a lefedettség-mérőt felkészíteni és alkalmazni. A pénztárgépen úgynevezett headless Android rendszer fut, ami azt jelenti, hogy semmilyen grafikus megjelenítésre nem képes, ráadásul ha egy program ezen a rendszeren grafikus elemet próbál használni, akkor az a rendszerben hibát eredményez és a program futása megszakad. Továbbá kérés volt, hogy a pénztárgép belső memóriáját használjuk az adatok tárolására, ne a korábban megvalósított szerver-kliens rendszert.

Másik, teljesen hasonló projekt, ami szintén a fentebb említett pénztárgép típusába tartozó eszközhöz kapcsolódik, egy nyomon követési feladat, miszerint egy távoli szerverre bizonyos időközönként el kell küldeni a futtatott metódusok azonosítóját és futásaik számát, ezzel elősegíteni egy heurisztika felállítását az egységnyi idő alatt futtatott metódusok mennyiségéről, esetleg egyes metódusok futtatási gyakoriságáról.

Irodalomjegyzék

[1] *Android* wikiedia page
http://en.wikipedia.org/wiki/Android_(operating_system) (2013 Május)

[2] *Open Handset Alliance*'s Android page
http://www.openhandsetalliance.com/android_overview.html (2013 Május)

[3] *Android* homepage
https://www.android.com (2013 Május)

[4] Árpád Beszédes, Tamás Gergely, István Papp, Vladimir Marinkovic, Vladimir Zlokalica: *Survey on Testing Embedded Systems*. Technical report, Department of Software Engineering, University of Szeged and Faculty of Technical Science, University of Novi Sad (2012)

[5] *EXOON* Android TV Set Top Box
http://www.xbmcandroidtv.com/shop/category-layout/power-pro-detail.html (2013 November)

[6] *PRWeb* about EXOON's Android TV
http://www.prweb.com/releases/Android-TV-ROM-Challenge/08-02-2013/prweb10991071.htm (2013 November)

[7] Thomas J. McCabe: *A Complexity Measure*
IEEE Transactions on Software Engineering, Vol. Se-2, No.4, p.308–320 (December 1976)

[8] *Javassist* homepage
http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/ (2013 November)

[9] *apktool* homepage
https://code.google.com/p/android-apktool/ (2013 November)

[10] *aapt* tool's homepage
http://elinux.org/Android_aapt (2013 November)

[11] *Jarsiner* homepage
http://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html (2013 December)

[12] *dex2jar* homepage
https://code.google.com/p/dex2jar/ (2013 December)

[13] Michael Barr: *Embedded Systems Glossary*. Netrino Technical Library
http://www.barrgroup.com/Embedded-Systems/Glossary-E (2013 December)

[14] RT-RK Institute: *RT-Executor* homepage
http://bbt.rt-rk.com/software/rt-executor (2013 December)

[15] *Debugging* wikiedia page
http://en.wikipedia.org/wiki/Debugging (2013 December)

[16] *CIRENE* homepage
http://cirene.sed.hu/ (2013 December)

[17] Sandra Kukolj, Vladimir Marinkovic., Miroslav Popovic., Szabolcs Bognár: *Selection and prioritization of test cases by combining white-box and black-box testing methods*.
In: Proceedings of the 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems (2013)

[18] *Debugging tools* wikiedia page
http://en.wikipedia.org/wiki/Debugger (2013 December)

[19] Jonathan B. Rosenberg: *How Debuggers Work: Algorithms, Data Structures, and Architecture* (1996)

[20] Advanced RISC Machines Inc (ARM): *Debugging Embedded Systems* (2001)

[21] *Debugging tools for embedded systems*
http://www.electronicproducts.com/Software/Debugging_tools_for_embedded_systems.aspx (2013 December)

[22] *Emulator* wikiedia page
http://en.wikipedia.org/wiki/Emulator (2013 December)

[23] Joachim Wegener, Kerstin Buhr, Hartmut Pohlheim: *Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing* (2002)

[24] Daniel Sundmark: *Structural System-Level Testing of Embedded Real-Time Systems.* p.19-34 (2008)

[25] S.J. Cunning and J.W. Rozenblit: *Automatic test case generation from requirements specications for real-time embedded systems*
In: Systems, Man, and Cybernetics (1999)

[26] I. K. El-Far and J. A. Whittaker. *Model-based software testing*
In: Encyclopedia of Software Engineering, p.1-22. (2001)

[27] Kim Hazelwood and Artur Klauser: *A dynamic binary instrumentation engine for the ARM architecture*.
In: Proceedings of CASES '06, p.261-270 (2006)

[28] S. Kandl, R. Kirner, and P. Puschner: *Automated formal veri cation and testing of c programs for embedded systems.*
In: Object and Component-Oriented Real-Time Distributed Computing, p.373-381 (2007)

[29] Tingting Yu, Ahyoung Sung, W. Srisa-an, and G. Rothermel: *Using property-based oracles when testing embedded system applications*.
In: Software Testing, Verification and Validation, p.100-109 (2011)

[30] *Dalvik Virtual Machine* wikiedia page
http://en.wikipedia.org/wiki/Dalvik_(software) (2013 December)

[31] *Tracing* wikiedia page
http://en.wikipedia.org/wiki/Tracing_(software) (2013 December)

[32] *Traceability* wikiedia page
http://en.wikipedia.org/wiki/Traceability (2013 December)

[33] *Embedded systems functional testing methodology*.
Cirene project technical report. (2012)

[34] *Embedded system software and hardware analysis methodology*.
CIRENE project technical report. (2012)

[35] *Specialized Methodology*.
CIRENE project technical report. (2012)

[36] *Test System Architecture*.
CIRENE project technical report. (2012)

[37] *JSON*
http://www.json.org/ (2013 December)

Nyilatkozat

Alulírott *Bognár Szabolcs Programtervező Matematikus* szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Tanszékcsoport *Szoftverfejlesztés* Tanszékén készítettem, *Programtervező Matematikus* diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy diplomamunkámat a Szegedi Tudományegyetem Informatikai Tanszékcsoport könyvtárában, a helyben olvasható könyvek között helyezik el.

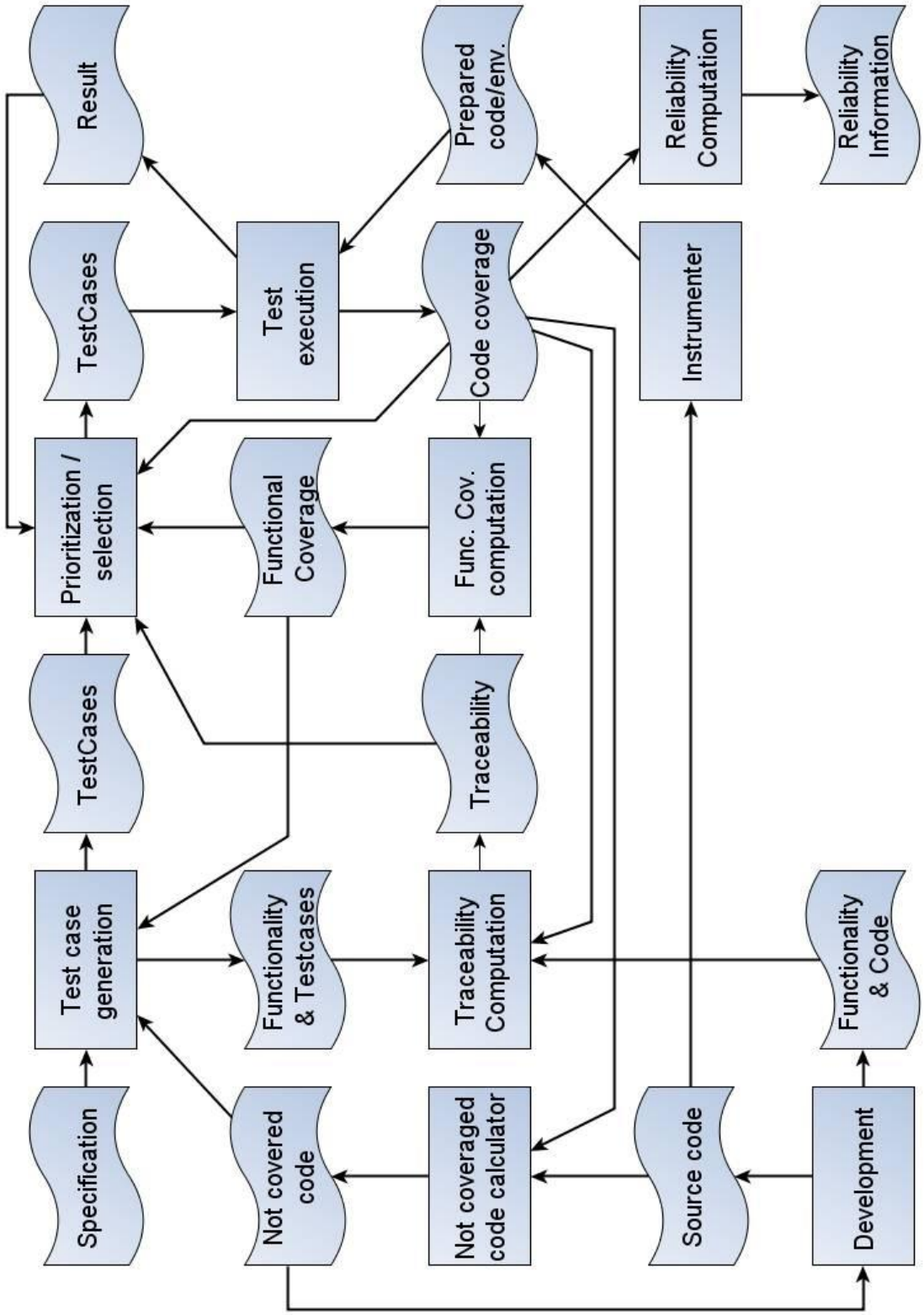
2013. december 07.

Aláírás

Köszönetnyilvánítás

Köszönöm mindenkinek, aki segített dolgozatom elkészítésében, aki lehetőséget adott, hogy dolgozzam és a munkámból megírjam a dolgozatomat, aki segített a munkámban, és aki segített a mindennapjaimban.

Mellékletek



4.3.1. ábra: Általános módszertan beágyazott rendszerek teszteléséhez (Készült a [33] jelzetű dokumentumoz)