Objektumorientált környezetben készült biztonságkritikus szoftverrendszerek verifikálása

BENYÓ BALÁZS

Széchenyi István Egyetem, Informatika Tanszék
benyo@sze.hu

Kulcsszavak: biztonságkritikus rendszer, szoftver verifikáció, iteratív tesztgenerálás, regressziós tesztelés

Annak érdekében, hogy modern, objektumorientált (OO) technológia előnyeit kihasználó szoftverfejlesztés lehetővé váljon OO környezetben alkalmazható szoftver verifikációs módszerekre és a módszerek egyszerű megvalósítását lehetővé tevő fejlesztői környezetre van szükség. A cikkben egy olyan szoftver verifikációs eljárásokat és az eljárásokat támogató, ill. megvalósító keretrendszer kerül bemutatásra, mely lehetővé teszi objektumorientált rendszerek alapos tesztelését és támogatja az auditáláshoz szükséges dokumentáció előállítását. A keretrendszer vasútirányító szoftverek auditálásához készült.

A biztonságkritikus rendszerek fejlesztésekor különösen nagy gondot kell fordítani a rendszerek szolgáltatásbiztonságának igazolására. Különböző szakterületeken (vasút, egészségügy, légi közlekedés stb.) szakterületi szabványok rögzítik annak szabályait, hogy milyen vizsgálatokat kell az adott területen alkalmazott rendszer használatbavétele előtt elvégezni. Ezek a szabványok definiálják azt a kritériumrendszert, mely alapján az adott területen használt rendszert a szolgáltatásbiztonság szempontjából minősíteni vagy auditálni lehet.

Az alkalmazott rendszerek méretének és bonyolultságának növekedésével ezek a szabványok egyre kevésbé alkalmasak a megfelelően részletes szabályozásra, így egyre kevésbé alkalmazhatóak közvetlenül a gyakorlatban. Míg az elsősorban mechanikus és elektronikus hardver elemeket tartalmazó rendszerek esetén viszonylag könnyen lehetett általános szabályokat definiálni a rendszerek szolgáltatásbiztonságának kimerítő vizsgálatára, úgy az elsősorban szoftver komponensekből álló rendszerek esetén ezek a szabványok megmaradnak az általánosságoknál.

A hiányos, gyakran túl általános szabályozás a fejlesztők felelősségévé tette a szoftverrendszerek megfelelő módszerekkel, megfelelő mértékben történő verifikálását és validálását. A gyakorlatban ez azt eredményezte, hogy a biztonságkritikus szoftverek előállítói – óvatos megközelítéssel – csak kiforrott, régóta használatos szoftverfejlesztési technológiát és környezetet alkalmaztak, melyek esetén rendelkeztek a megfelelő módszertani és technológiai háttérrel a rendszerek teszteléséhez. Így gyakran előfordul, hogy biztonságkritikus szoftverrendszereket még ma is strukturált megközelítésben, procedurális programnyelveken fejlesztenek.

1. Célkitűzések

Az OO rendszerek verifikálása és tesztelése során a legnagyobb problémát a rendszerek korlátozott megfigyelhetősége okozza [3,6,8]. Az OO rendszerek korlátozott megfigyelhetősége az OO nyelvek természetéből adódik. Az egységbezárás (encapsulation), illetve az adatrejtés (data hiding) a OO programozási paradigma alapelvei, melyek elősegítik a kód-újrafelhasználást és a hatékony programfejlesztést. Ezt a problémát ügyes, a tesztelés szempontjait is figyelemben vevő tervezéssel, illetve implementációval legfeljebb csökkenteni lehet, de elkerülni nem.

A cikkben bemutatott kutatás-fejlesztési munka célja olyan verifikációs módszerek kidolgozása volt, melyek lehetővé teszik nagyméretű és bonyolult OO szoftverrendszerek verifikálását és ugyanakkor alkalmazhatóak a gyakorlatban. Valós méretű problémák esetén történő alkalmazhatósága a módszereknek kitüntetett fontosságú volt, mert az irodalomban publikált módszerek jelentős része a gyakorlatban nem állta meg a helyét [1]. A cikkben ismertetett módszereket verifikációs keretrendszer formájában implementáltuk annak érdekében, hogy gyakorlatban is kipróbáljuk őket. A gyakorlatban történő alkalmazhatóságon felül a módszerek kidolgozásakor, illetve a keretrendszer fejlesztésekor a következő célokat tűztük ki:

- Tegye lehetővé nagyméretű és összetett OO rendszerek tesztelését.
- A rendszer verifikálásának támogatása a fejlesztés minden fázisában.
- A rendszer végső auditálásának támogatása.
- A rendszerfejlesztők és tesztelők munkájának felhasználóbarát támogatása.
- Különböző tesztelési módszerek támogatása függetlenül a tesztelt rendszer jellemzőitől.

2. Módszerek

Minden klasszikus tesztelési stratégiának (hierarchikus tesztelés: top-down, bottom-up tesztelés; izolációs tesztelés stb.) és tesztelési módszernek (határérték tesztelés, ekvivalencia partícionálás, úttesztelés stb.) van olyan előnyös tulajdonsága, mely az adott tesztelési

stratégiát vagy módszert optimálissá teszi egy-egy speciális felépítésű, vagy adott tulajdonságokkal rendelkező rendszer tesztelésekor [2,7,8]. A gyakorlatban sajnos a tesztelt alkalmazások nem homogén felépítésűek ebből a szempontból, vagyis az egyes módszerek csak a tesztelt rendszer egy jól meghatározható részére lesznek hatékonyak.

Felismerve ezt az inhomogén tulajdonságát a valós rendszereknek a tesztelési környezet magját úgy alakítottuk ki, hogy az egyes, környezet által támogatott módszerek opcionálisan választhatóak, azokat nem kötelező alkalmazni minden rendszerkomponens esetén. A tesztelési környezet ebből a szempontból nyitott, könnyen kiegészíthető bármilyen további módszert támogató komponenssel, valamint a különböző tesztelési módszerek kombinálhatóak egy adott részrendszer tesztelésekor.

A következőkben röviden ismertetjük azokat a tesztelő, illetve tesztelést támogató módszereket, melyek implementálásra kerültek a tesztelő keretrendszerben.

A. Objektumok becsomagolása (object wrapping)

OO rendszerek verifikálásakor kritikus kérdés, hogy hogyan lehet megoldani a rendszer viselkedésének megfigyelését. A problémát az okozza, hogy OO alkalmazásokban az osztályok és objektumok tagváltozói, illetve tagfüggvényeinek elérhetősége korlátozott, azokat csak a kód jól meghatározott részeiből lehet olvasni vagy meghívni. Az OO rendszereknek ezt a lehetőségét, illetve tulajdonságát a jól tervezett OO alkalmazások igen intenzíven használják, mert ez segíti a kódújrafelhasználást [1,13].

Annak érdekében, hogy ellensúlyozzuk az OO rendszerek ezen tesztelést megnehezítő tulajdonságát szükséges, hogy a tesztelő keretrendszer támogassa a tesztelt rendszerben lévő objektumok viselkedésének megfigyelését. Ennek egyik lehetséges módja olyan megfigyelő osztályok készítése, amelyek olyan viszonyban (például C++ esetén friend) vannak a megfigyelt osztállyal, mely megengedi a külvilág számára nem elérhető adatok elérését. Ez az út sajnos az esetek többségében nem járható, mert csak a megfigyelt rendszer megváltoztatásával valósítható meg.

Az általunk készített keretrendszerben az objektumok viselkedésének megfigyelését a klasszikus rendszerekben alkalmazott becsomagolás (wrapping) módszer OO rendszerek osztályainak megfigyelésére kidolgozott változatával támogattuk. A módszer működését az *1. ábra* szemlélteti.

A tesztelés megkezdésekor minden rendszerbeli osztályhoz egy úgynevezett csomagoló (wrapper) osztályt hozunk létre. Ezek a csomagoló osztályok a teszteléshez kapcsolódó feladatok támogatására szolgálnak. A tesztelő rendszer ezen csomagoló osztályokon keresztül fogja a tesztelt rendszer objektumait elérni. A csomagoló osztály rendelkezik minden olyan külvilág által elérhető függvénnyel, mint a megfigyelt rendszerben levő párja. Ezen függvények törzse két funkciót valósít meg:

– a függvényhívás tényének és paramétereinek elmentése a tesztelés eredményét rögzítő adat fájlba;
– az eredeti megfigyelt rendszerben levő osztály megfelelő függvényének meghívása.

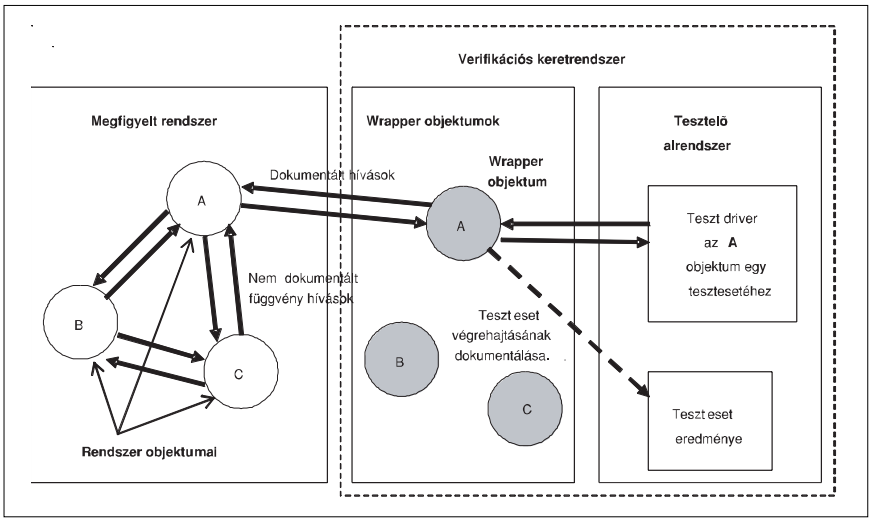
A rendszer működése ezek után egyszerű. A teszteset végrehajtásakor a tesztelő rendszer ezen csomagoló osztályokat fogja létrehozni és használni. A csomagoló osztályok fogják automatikusan instanciálni a megfigyelt rendszerben levő párjukat. Amikor a tesztelő környezet teszteli az egyes tagfüggvényeket, akkor a csomagoló osztály megfelelő függvényét fogja meghívni, ami automatikusan dokumentálja a hívást, és a paramétereket, majd aktivizálja az eredeti függvényt. A csomagoló függvény képes a tesztelt függvény visszatérési értékét is a teszteredmény fájlba elmenteni.

Természetesen ez a módszer nem fogja tudni a rendszeren belül történő hívásokat dokumentálni, azonban szisztematikus módszert ad az objektumok tesztelés során történő megfigyelésére. Az, hogy a rendszeren belül történő hívásokat nem tudjuk megfigyelni a hibadiagnosztikát megnehezíti. Ez a probléma azonban részben kiküszöbölhető az osztályok megfelelő sorrendben történő tesztelésével. Ha figyelembe vesszük az objektumok használati sorrendjét a rendszer természetes működése során, lehetséges olyan tesztelési stratégia kialakítása, mely esetén a egy adott teszteset végrehajtásakor az aktuálisan tesztelt hívás csak már tesztelt rendszeren belüli függvényhívásokat használ.

B. Tesztelés alaposságának mérése

Minden tesztelés során szükséges a tesztelés folyamatát megtervezni és irányítani, definiálni, hogy mely teszteseteket kívánjuk a tesztelés egy adott fázisában végrehajtani a tesztelt rendszeren. A tesztelés során ehhez általában valamilyen kvantitatív mérőszámot hasz-

1. ábra Csomagoló (wrapper) osztályok használata



nálunk, mely leggyakrabban egy arányszám, amely azt hivatott kifejezni, hogy a tesztelt rendszerünk mekkora részét teszteltük le tesztelés adott fázisában [7]. Számos ilyen arányszám létezik, azonban a gyakorlatban az utasítás lefedettség (statement coverage) mérőszám használható a legáltalánosabban. Az utasítás lefedettség használatának előnyei:

- Egyszerű az utasítás lefedettség mérése, és egyszerű az eredmény interpretálása.
- Összehasonlítva más mérőszámokkal, nem kötődik szorosan egyetlen tesztelési megközelítéshez, vagy módszerhez sem [4].

A tesztelési keretrendszerben az utasítás lefedettséget használtuk a tesztelés alaposságának jellemzésére.

C. Iteratív tesztelés

A modern OO rendszerfejlesztési metodikák alapján történő szoftverfejlesztés esetén az alkalmazásokat iteratív fejlesztési fázisok eredményeként állítjuk elő [13]. A rendszer által megvalósítandó funkciókat csomagokra osztjuk. A különböző csomagokban definiált funkciókat egymás után implementáljuk. Ennek megfelelően az egyes iterációk eredményeként előállított szoftver verziók az előző fázisokban előállított verziók kiterjesztett változatai lesznek [10].

A tesztelésnek illeszkedni kell ehhez az iteratív fejlesztési metodikához. Az iteratív fejlesztésnek két fontos következménye van a tesztelés szempontjából:

- Erősen támogatni kell a regressziós tesztelést, vagyis a tesztesetek ismételt végrehajtását, egy adott komponens újratesztelését.
- Támogatni kell a tesztesetek halmazának egyszerű bővítését.

A keretrendszer a tesztesetek halmazának egyszerű bővíthetőségét a következő alfejezetben ismertetésre kerülő hierarchikus teszteset azonosítókkal támogatja. A tesztesetek egyszerű megismételését a teszt driverek kódjának a forrásprogramként történő definiálásával oldottuk meg, mely definíció célszerűen a tesztelt rendszer fejlesztési környezetéhez illeszkedik. Ennek részleteit a következő alfejezetek tartalmazzák.

Az iteratív fejlesztés közvetlen támogatásán felül az iteratív fejlesztés gondolatát közvetlenül is alkalmaztuk a tesztesetek definiálásakor. Mivel a tesztelő környezet már fel volt készítve lépésenként növekvő teszthalmazok kezelésére és ismétlődő végrehajtására, kidolgoztuk az iteratív tesztelési módszerét, mely illeszkedik a klasszikus tesztelés gyakorlatához.

Az iteratív tesztelés alapötlete igen egyszerű. Egy adott funkcionalitású komponens tesztelését több fázisra bontjuk. Az egymást követő fázisokban a tesztelésnek egyre szigorúbb feltételeknek kell eleget tennie. Ennek megfelelően az egyes fázisokban használt tesztesetek halmazai – hasonlóan a szoftver verziókhoz – egyre bővülő halmazok, egymás kiterjesztett változatai lesznek.

Ennek a többfázisú tesztelésnek a potenciális előnye lehet a fejlesztés felgyorsítása. Elvileg egy adott fejlesztési fázis csak a korábbi fejlesztési fázis tesztelése után kezdődhet meg. A tesztelés azonban igen időigényes feladat, főleg biztonságkritikus rendszerek esetén, amikor a tesztelésnek igen szigorú követelményeknek kell eleget tennie. Iteratív tesztelés esetén elképzelhető, hogy a tesztelés valamelyik korai fázisának lezárása után elkezdődhet a termék következő fejlesztési fázisa. Ugyan a rendszerben potenciálisan maradnak még felderítetlen hibák, de azok száma viszonylag kicsi, így javításuk nem lesz olyan költséges a már részben továbbfejlesztett kódban, hogy ne érje meg ezt az árat megfizetni a fejlesztés lényeges felgyorsításáért.

A tesztelési keretrendszerben a tesztelésnek három iteratív fázisát definiáltuk:

– előzetes tesztelés,
– modul tesztelés,
– integrációs tesztelés.

Az egyes iterációk elnevezése nem véletlenül hasonló a klasszikus tesztelésben használt tesztelési fázisok elnevezéséhez. Az egyes tesztelési iterációk hasonló szerepet töltenek be az egyes fejlesztési fázisokban előállított szoftverek tesztelésekor, mint a lineáris, nem iteratív rendszerfejlesztés során alkalmazott tesztelési fázisok (modul tesztelés, integrációs tesztelés).

Definíciónk szerint tesztelő keretrendszerben a három iteratív tesztfázisban előállított teszthalmaznak a következő kritériumokat kell teljesíteni:

Előzetes tesztelés:

A szoftver legfontosabb, leggyakrabban használt funkcióit kell letesztelni. Nem szigorú kritérium a száz százalékos utasítás-lefedettség. A gyakorlatban 60-95%-os lefedettséget értek el az ebben a fázisban kidolgozott teszthalmazhoz tartozó tesztesetek.

Modul tesztelés:

A cél a tesztelt szoftver adott fejlesztési fázisban definiált komponenseinek önállóan történő alapos tesztelése. Követelmény a 100%-os utasítás lefedettség.

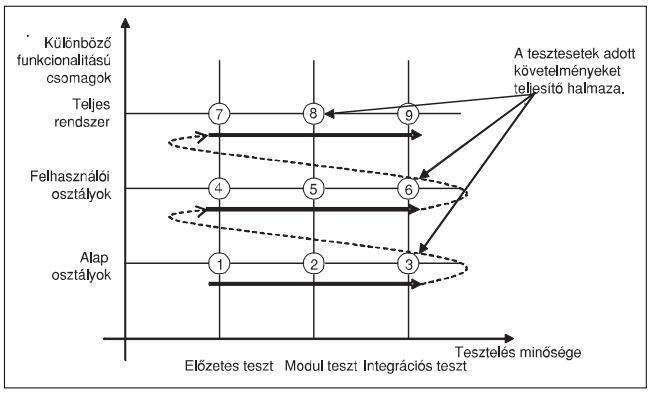
Integrációs tesztelés:

A cél a tesztelt szoftver adott fejlesztési fázisban definiált komponensei közötti kommunikáció alapos tesztelése. A komponenseket ebben a tesztelési fázisban egymással kommunikáló egységeknek tekintjük, és a tesztelés célja a köztük levő interfészek alapos tesztelése. Követelmény a 100%-os utasítás lefedettség.

A *2. ábra (a következő oldalon)* iteratív tesztelés módszerének alkalmazása esetén előállított tesztesethalmazokat mutat. Az ábrán a példaként vett szoftvernek három iteratív fejlesztési fázisban előállított verzióját láthatjuk: *Alap osztályok, Felhasználói osztályok, Teljes rendszer.* Minden egyes szoftver verziónál mind a három, fent definiált iteratív tesztelési fázisban előállított teszthalmazt szemléltettük. A teszthalmazokra írt számok, illetve a nyilak a teszthalmazok előállításának sorrendjét mutatják.

D. Tesztesetek hierarchikus számozása

A tesztesetek az adott rendszernek egy-egy jól definiált tulajdonságát verifikálják. A teszteseteket egyértelműen azonosítani kell a tesztelés során, az azonosí-



2. ábra
Különböző teszteset halmazok iteratív tesztelés esetén

tásra a teszteset azonosító (ID) szolgál. A teszteseteket elláthatjuk manuálisan ezzel az azonosítóval, azonban egy több ezer teszteset esetén már igen nehézkes. Egyszerűbb módja a teszteset azonosításnak, ha a teszt driver (teszt vezérlő) kódrészletek önmaguk generálják ezt az azonosítót a teszthalmazban levő elhelyezkedésük alapján.

Mivel a tesztesetek megvalósítását, a teszt driver (teszt vezérlő) kódrészleteket szekvenciálisan tároljuk, ahol célszerűen egymás után helyezkednek el egy adott osztály vagy komponens adott funkcióját verifikáló teszt driverek, nehézséget okozhat új tesztesetek beszúrása a teszthalmazba anélkül, hogy megváltozzon a beszúrt teszteset utáni tesztesetek azonosítója.

Ennek a problémának a megoldására vezettük be a hierarchikus teszteset azonosítók használatát. A teszteset azonosítók pontokkal aláosztott számsorozatok, pl. 1.15.24. Ebben az esetben, ha a teszthalmazba új tesztesetet kell beszúrni nem kell másra ügyelni, minthogy az újonnan beszúrt teszteset új aláosztást kezdjen. Tehát ha a 1.15.24-es teszteset után már volt 1.15.25-ös teszteset definiálva, akkor a kettő teszteset közé 1.15.24.1, 1.15.24.2 stb. számú teszteseteket szúrjuk be. Az azonosítók generálása automatikusan történt a tesztkörnyezet segítségével. Ha a teszteset definiálója új aláosztást akart kezdeni egy egyszerű deklarációt szúrt a teszteset teszt driverének első sora elé, és a rendszer automatikusan új azonosító hierarchiát kezdett.

E. Tesztesetek definiálása

A tesztesetek végrehajtását végző teszt driverek a tesztelt rendszer környezetéhez igazodva készültek. A teszt driverek forráskódját lefordítva egy olyan végrehajtható alkalmazást generáltunk, mely képes volt a teszteseteket végrehajtani. Mivel a teszt driverek tartalmazták a tesztesetek definícióját, generálták az azonosítóit, így – megfelelő paraméterezéssel – ez a program képes volt a teszteset leírásokat, specifikációkat is generálni. Természetesen paraméterezhető volt, hogy a tesztelő alkalmazás mennyire részletes tesztelési eredmény kimenetet készítsen: minden függvényhívás paraméterei szerepeljenek, vagy csak a tesztesetek eredménye. A teszt driverek tartalmazták a teszteset elfogadásának kritériumait, tehát maguk ellenőrizték a teszteset végrehajtásának hibás, illetve hibátlan voltát.

A tesztelő keretrendszer megvalósítása során egy C++ környezetben fejlesztett alkalmazást teszteltünk, így a keretrendszer is C++ környezetben készül el. A keretrendszer magja tartalmazza az összes olyan osztály, adatszerkezet definícióját, mely szükséges a tesztesetek egyszerű definiálásához.

Egy tipikus teszteset definíció három részből áll:

- Teszteset leírásából, specifikációjából.
  - TS_REQ: A tesztelt követelmény, illetve funkció leírása, mely a rendszer követelmény specifikációjában szerepelt.
  - TS_DESC: Annak a módszerének a leírása, amivel a követelményt teszteli a teszteset.
  - TS_EXPOUT: A kívánt, helyes működés során várt eredmény leírása.
- A tesztesetet végrehajtó kódrészletből, a teszt driverből.
- A teszteset végrehajtása után, az eredmény vizsgálatához szükséges ellenőrzések definíciójából. (TS_ASSERT).

A *3. ábrán* egy tipikus teszteset definíciót láthatunk. A TS_CASE_BEGIN(2) a tesztesethez tartozó teszt driver definíciójának elejét a TS_CASE_END a végét jelöli. A TS_CASE_BEGIN után a (2) azt jelöli, hogy a teszteset azonosítójában 2 aláosztás kell hogy szerepeljen.

```
TS_CASE_BEGIN(2);
TS_REQ("getNumTimersUsed: Returns the number of actual registered framework timers in the
        process.");
TS_DESC("Calling getNumTimersUsed(), no timers.\n"
          "Function returns the number of existing timers, i.e. 0.");
TS_EXPOUT( "Returns the number of actual registered framework timers in the process.");
TS_METHOD(TS_BOUND);
TS_CLASS_POS;
TS_TRY
{
        TS_ST_TimerControl          test_timerControl(timerControl);
        UInt32 testNumTimersUsed = test_timerControl.getNumTimersUsed();
        // check the number of existing timers
        TS_ASSERT( testNumTimersUsed == g_initedTimers );
}
TS_CATCH_ASSERT_FALSE;
TS_CASE_END;
```

3. ábra
Egy egyszerű teszteset definíciója

A 3. ábrán definiált teszteset végrehajtásakor generált teszt eredmény fájl adott tesztesethez tartozó részét a *4. ábrán* láthatjuk. A teszteset azonosítója 1.2.2, ami két aláosztást tartalmaz a 3. ábrán látható deklarációnak megfelelően. Az teszteset eredményének első részében láthatjuk a teszteset leírását a 3. ábrán definiált sorrendben. Ezután a teszteset végrehajtásának eredménye látható, ami ebben az esetben pozitív, tehát a tényleges eredmény azonos a várt eredménnyel.

```
TEST_CASE:              #1.2.2  (106)
TEST_REQUIREMENT:       getNumTimersUsed: Returns the number of actual
                        registered framework timers in the process.
TEST_DESCRIPTION:       Calling getNumTimersUsed(), no timers.
                        Function returns the number of existing timers, i.e. 0.
TEST_EXPECTED_OUTPUT:   Returns the number of actual registered framework
                        timers in the process.
TEST_METHOD:            Test case execution from boundary value analysis
TEST_CLASSIFICATION:    POSITIVE
-------------------
TEST_FUNCTION_CALL:     ST_TimerControl::getNumTimersUsed()
TEST_ASSERTED_TERM:     testNumTimersUsed == g_initedTimers
TEST_ASSERT_RESULT:     OK
TEST_RESULT:            PASSED
```

4. ábra
A 3. ábrán bemutatott teszteset végrehajtásakor keletkező kimenet

A tesztelő keretrendszer tartalmazza a az utasítás lefedettség mérésére és a tesztelési eredmény fájl elemzésére, statisztika készítésére szolgáló komponenseket is. Egy ilyen kiértékelés eredményét láthatunk az *5. ábrán*, mely az utasítás lefedettség mértékét mutatja. Az utasítás lefedettség a kódsorok végrehajtásának megfigyelésén alapult, mely információt a fejlesztői környezet szolgáltatta.

```
COVERAGE_REPORT_START
Total source files:          8
Total source lines:          372
Covered lines:               354     (95.2%)
Not covered lines:           18      (4.84%)
Not covered & not marked lines: 0    (0%)
Overall coverage:            372     (100%) <<<<
COVERAGE_REPORT_END
```

5. ábra
A tesztelés eredményének kiértékelése

3. Eredmények

Az ismertetett verifikációs módszereket, a módszereket implementáló tesztelési keretrendszert egy alkalmazásspecifikus operációs rendszer verifikálásánál alkalmaztuk. Az operációs rendszer on-line vasúti irányítórendszerek alkalmazásainak futtatására készült. A tesztelt alkalmazás jelenleg is valós környezetben működik.

A rendszer tesztelése során az iteratív tesztgenerálás módszerét alkalmaztuk minden funkcionális csomag esetén. A tesztelés során kidolgoztuk a teljes rendszerre a öndokumentáló, újra végrehajtható teszteseteket tartalmazó tesztelő rendszert. A tesztelés során teljes forráskódra nézve elértük a száz százalékos utasítás lefedettséget. Az operációs rendszer, a kidolgozott tesztelő rendszer segítségével sikeresen megfelelt a CENELEC szabványban [11] definiált hivatalos auditálási procedúrán.

4. Összefoglalás

A dolgozatban olyan rendszerverifikácós módszereket mutattunk be, melyek lehetőséget adnak nagyméretű és összetett objektumorientált rendszerek tesztelésére. Az ismertetett módszereket tesztelési keretrendszer formájában implementáltuk és sikeresen alkalmaztuk egy alkalmazás-specifikus operációs rendszer tesztelésénél és auditálásánál. A kidolgozott módszerek igen hatékonynak bizonyultak, segítségükkel a tesztek kidolgozására, illetve a tesztelésre fordított idő a legóvatosabb becslések alapján is kevesebb, mint a felére csökkent az eredetileg tervezett, e módszerek alkalmazása nélküli állapothoz képest.

Köszönetnyilvánítás

A tesztelési keretrendszer kialakításában történő közreműködésükért köszönetet mondok dr. Várady Péternek és Asztalos Attilának. A kutatómunkát támogatta az Országos Tudományos Kutatási Alap (OTKA-F046726), valamint a Gazdasági és Közlekedési Minisztérium (AKF-05-0093, AKF-05-0408, RET-04-2004).

Irodalom

[1] D. Ince: Software Testing in J. McDermin (ed.): Software Engineer's Reference Book, Butterworth-Heinemann Ltd., 1991.
[2] J.J. Chilenski, S.P. Miller: Applicability of Modified Condition Decision Coverage to Software Testing, Boeing Company and Rockwell International Co.,1993.
[3] P. Várady: Konzeption und Entwicklung einer Analysebibliothek zum Test des Verhaltens eingebetteter Software, Diploma Thesis in German, FZI-MRT Karlsruhe, 1997.
[4] H. Younessi: Object-Oriented Defect Management of Software, Prentice-Hall, Inc., USA, 2002.
[5] B. Benyó, J. Sziray: The Use of VHDL Models for Design Verification, IEEE European Test Workshop (ETW2000), Cascais, Portugal, May 23-26, 2000, ISBN 0-7695-0701-8
[6] P. Várady, B. Benyó: A systematic method for the behavioural test and analysis of embedded systems, INES 2000, 4th IEEE International Conference on Intelligent Engineering Systems 2000.
[7] IPL Information Processing Ltd: Advanced Coverage Metrics for Object-Oriented Software, White paper of IPL Information Proc. Ltd, 1999.
[8] J. Sziray, B. Benyó., I. Majzik, L. Kalotai, J. Góth, T. Heckenast: Quality Mangement and Verification of Software Systems, Research Report (in Hungarian), (KHVM 47/1998), Budapest, 2000.
[9] M. R. Woodward, D. Hedley, M. A. Hennell: Experience with Path Analysis and Testing of Programs, IEEE Transactions on Software Engineering, Vol. SE-6, No.3, pp.278–286., May 1980.
[10] David E. Avison, Hanifa U. Shah: The Information Systems Development Lifecycle: A First Course in Information Systems, McGraw-Hill Book Company, Great Britain, 1997.
[11] European Standard EN-50128, Final Draft, Railway Applications: Software for Railway Control and Protection Systems, CENELEC: European Committee for Electrotechnical Standardization, 1997.
[12] M. Dorman: C++ "It's Testing, Jim, But Not As We Know It", White paper of IPL Information Proc. Ltd, 1999.
[13] I. Jacobson, G. Booch, J. Rumbaugh: Unified Software Development Process, Addision-Wesley, USA, 1999.