AZ EKFCSLINT HOLTPONT KERESŐ ESZKÖZ

THE EKFCSLINT DEADLOCK CHECKER TOOL

Kusper Gábor, Király Roland, Kovács Emőd, Baranyi Ádám, Csintalan Ádám, Gréczi László, Pekk Roland, Pető György
Eszterházy Károly Főiskola

Összefoglaló

Ez a cikk az Eszterházy Károly Főiskolán az evosoft Hungary Kft. megrendelésére elkészült C# holtpont kereső, az EKFCSLint, működő prototípusát mutatja be. Holtpont (deadlock) akkor következhet be, ha két, vagy több folyamat egymás elől kölcsönösen lefoglalja az erőforrásokat, vagy több mint két folyamat körbe hivatkozva vár az erőforrásokra. A holtpont keresők nagy szerepet töltenek be olyan kritikus szoftverek ellenőrzésében, mint például az egészségügyi vagy banki szoftverek. Beszélhetünk dinamikus és statikus holtpont keresőkről. Az előbbiek futtatják a kódot, az utóbbiak elemzik. Az EKFCSLint statikus holtpont kereső. Speciálisan nem a C# forráskóddal, hanem a köztes bájtkóddal dolgozik, így a nagy értékű forráskód felfedése nélkül képes holtpontot keresni. A köztes kódból felépíti a foglalási (lock) gráfot, ahol a gráf csúcsai a különböző erőforrások, az élek pedig az őket lefoglaló metódusok. A foglalások *Monitor.Enter* és *Monitor.Exit* közt találhatók, ezeket kell megkeresni a köztes kódban. Ezután a foglalási gráfban kört kell keresni. Minden kör egy-egy lehetséges holtponti helyzetet jelent, amit még egy programozónak ellenőriznie kell. A szoftverhez készült egy vizuális nyomkövető, ami a köztes kód elemzését segíti, illetve foglalási gráf megjelenítő. Ez utóbbival eddig egyetlen C# holtpont kereső se rendelkezett.

Kulcsszavak

Holtpont, Holtpont kereső, CSLint, C#

Abstract

This article introduce the working prototype of a C# deadlock checker, the EKFCSLint, which was developed at the Eszterházy Károly College and which was ordered by the evosoft Hungary Kft. Deadlock might occur if two or more processes lock mutually from each other researches or two or more processes wait for researches in a circle, each waiting the next to raise the lock. The deadlock checkers play an important role in critical software, like health or banking software. There are dynamic and statical deadlock checkers. The first ones run the code, the second ones analyze the code. The EKFCSLint is a tactical deadlock checker. It is a special one, because it works not on the C# source code, but on the intermediate bytecode, so there is no need to discover the valuable source code to be able to detect deadlock. From the intermediate code we build the lock graph, in which the vertices are the researches and the edges are the methods which lock them. The locks are between *Monitor.Enter* and *Monitor.Exit*, so we look for these in the intermediate code. After this we try to find loops in the graph. Each loop means a possible deadlock, which has to be reviewed by a programmer. The software consists of also a visual debugger, which helps to analyze the intermediate code, and a lock graph visualizer. This is a feature which is not included in any other C# deadlock checker yet.

Keywords

Deadlock, Deadlock Checker, CSLint, C#

1. Bevezetés

Ebben a cikkben az Eszterházy Károly Főiskola Matematikai és Informatikai Intézet egyik projekt laborjában folyó kutatás eredményeiről számolunk be. A kutatás azzal a kérdéssel foglalkozik, hogyan lehet statikus módszerrel C# [1] programozási-nyelven írt kódból nagy biztonsággal kiszűrni az esetleges inkonzisztenciákat, holtpontokat (Deadlock) a program futtatása, és a forráskód közvetlen elemzése nélkül. A megvalósítás Windows, valamint Linux operációs rendszerre készül, lehetőleg az aktuálisan legfrissebb .NET keretrendszer lehetőségeit kihasználva. A legfontosabb célunk az volt, hogy készítsünk egy prototípust, valamint egy megvalósíthatósági tanulmányt, ami további fejlesztések, esetleg termék előállításának alapjául szolgálhat.

Elsőként értelmezzük a holtpont fogalmát. Általánosságban azt mondhatjuk, hogy akkor következik be holtpont szituáció, amikor két vagy több versenyző folyamat arra vár, hogy a másik befejeződjön, s ez egyik esetben sem következik be. Hasonló ez ahhoz, hogy eldöntsük mi volt előbb a tojás vagy a tyúk. Találó megközelítés még az is, amikor párhuzamos vasúti vágányokon közlekedő szerelvények egy időben érnek a kereszteződéshez. [2] Számítástechnikában a fogalom definícióját a következőképpen adhatjuk meg.

Holtpont (Deadlock): *akkor következhet be, ha két, vagy több folyamat egymás elől kölcsönösen lefoglalja az erőforrásokat, vagy több mint két folyamat körbe hivatkozva vár az erőforrásokra.*

A holtpont probléma gyakran előkerül elosztott rendszerek esetében [3-6]. Szokás megkülönböztetni a szoftveres (soft lock) és a hardveres (hard lock) eseteket is. Ebben a cikkben elsősorban a C# kód futtása során fellépő holtpontokkal foglalkozunk.

Gyakori eset, mikor két szál várakozik olyan erőforrásra, amit a másik szál foglalt le, így egyik sem futhat tovább. A probléma azért jelenthet különösen nagy veszélyt, mert csak futás időben derül ki, ha van holtpont és nem is minden futásnál, hiszen a párhuzamosan futó szálak egymáshoz viszonyított futása nem determinisztikus. Különösen nagy veszélyt jelent ez a bankszférában, vagy az egészségügyben használt rendszerek esetében, ezért a holtpont keresőkre nagy igény mutatkozik.

Az elkészített elemző rendszer továbbfejlesztése lehetséges, sőt szükséges is ahhoz, hogy valós környezetben megállja a helyét. Vannak még apróbb bizonytalanságok, de ezek éles környezetben felderíthetőek, javíthatóak. A holtpont kereső prototípusa, nem csak tudományos, de ipari szempontból is igen jelentős. Ezt bizonyítja az a tény, hogy az Eszterházy Károly Főiskolán kutatócsoport jött létre a holtpont kereső eszköz készítésére az evosoft Hungary Kft. támogatásával. A prototípust átadtuk tesztelésre, s más cégek is érdeklődtek a fejlesztés iránt. A következőkben megpróbáljuk röviden bemutatni a prototípus megvalósításának problémáit a modellalkotástól kezdve az implementáción át a tesztelésig. Valamint elemezzük a lehetséges felhasználási módokat.

2. A statikus holtpont kereső módszer

A kitűzött cél megvalósítását azzal kezdtük, hogy feltérképeztük a holtpont helyzetek kiszűrésének lehetséges technikáit. A felderítésre két alapvető módszer létezik: statikus analízis, vagy dinamikus elemzés.

A **statikus módszer** forráskód vagy köztes kód elemzésével valósítható meg, nem kell futtatni a forráskódot. A **dinamikus módszer** a programot futás közben ellenőrzi, valós időben figyeli a szál-műveleteket. Sajnos ez a módszer nem tudja teljes körűen tesztelni a programot. A megoldás kimerül abban, hogy elég nagyszámú „problémamentes" futtatás esetén helyesnek nyilvánítja a programot.

A fentiek azt sugallják, hogy a statikus módszer talán célravezetőbb. Ez mindenképpen igaz a gyorsaságot tekintve, de nem a pontosságot. A statikus módszerek általában csak a holtpont lehetőségét tárják fel. A figyelmeztetést még meg kell vizsgálnia egy programozónak, hogy a jelzett helyen ténylegesen lehet-e holtpont.

A statikus elemzés módszerei három fő irányvonalra bonthatóak:

- *Kódátalakítás alapú módszer*: A forráskódot automatikusan átalakítjuk olyan formára, mely megfelelő valamely létező model-checker program számára [7], majd az ellenőrzés eredményét feldolgozva, azt elemezhető formára alakítjuk.
- *Absztrakción alapuló módszer*: A forráskód alapján megalkotjuk a program egyszerűsített logikai modelljét, és a modell tulajdonságait vizsgáljuk.
- *Bájtkód elemzés*: A virtuális futtatókörnyezetet használ a lefordított köztes kód ellenőrzéséhez. Ennél a módszernél a forráskód nem szükséges az elemzéshez. Ennek számos előnye van, de talán a legfontosabb ezek közül az, hogy külső elemzők bevonása során a programkód nem kerül nyilvánosságra.

A kutatócsoport a *bájtkód elemzést* választotta, mivel a szükséges virtuális gép előnyeit jól ki tudtuk használni. Ezeket az előnyöket az alábbi néhány pontban foglaltuk össze:

- A virtuális gép segítségével a teljes programot elemezni lehet, nincs szükség átalakításra vagy magasabb absztrakciós szint bevezetésére.
- A program összes lehetséges állapotát feltérképezi, majd szűkíti a lehetőségeket és lehetséges holtpont helyzeteket, kivételeket keres.
- Gyors módszer hogy inkonzisztenciákat, és szerializációs problémákat is fel tudjunk deríteni.
- Nem igényel kódátalakítást a kódban.

A módszernek jelenleg számos megvalósítása létezik, de egységes, minden probléma felderítésére alkalmas rendszert nem találtunk. A kutatás során elemeztük a rendelkezésünkre álló eredményeket.

Jlint egy Java nyelvre írt modell checker, ami alkalmas többek közt deadlock-ok felderítésére is, egyszerű, és gyors modelchecker [8].

- A Java PathFinder a NASA [13] által fejlesztett deadlock checker.
- Az MMC mono Linux [10],[11] platformra fejlesztett model checker. Képes többek között deadlock felismerésére is.
- MoonWalker (régi nevén: MMC) .NET alkalmazásokra [9].
- A CsLint [12] ami C# kódot ellenőrző eszköz, deadlock felderítésére alkalmas.

Kutatócsoportunk a megvalósításhoz a nyílt forráskódú CsLint-et [12] választotta. Ez az eszköz még a .NET 1.0-hoz készült, és sajnos ennél újabb keretrendszer ellenőrzésére nem

alkalmas. A cikk írásakor legfrissebb a .NET 3.5-ös keretrendszer, így egyértelmű, hogy a CsLint eszköz kiinduló pontnak megfelelő, de mindenképpen továbbfejlesztésre szorul.

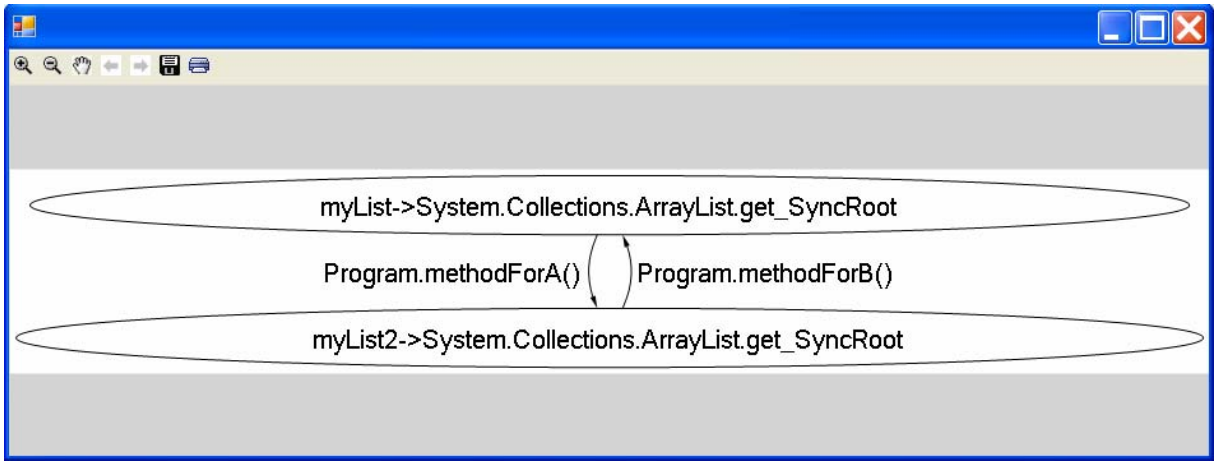
3. A megvalósítás lépései

A holtpont ellenőrző eszköz megvalósítása során a következő modellt építettük fel. Készítettünk egy köztes kód olvasó modult (scanner). A köztes kódból előállított kimenetet át kell adnunk egy feldolgozó egységnek (előfeldolgozó). Az előfeldolgozó felépíti a foglalási (lock) gráfot, ahol a gráf csúcsai a különböző erőforrások, az élek pedig az őket lefoglaló metódusok.

A gráf felépítéséhez, szükségünk van arra, hogy felismerjük a köztes kódban (Intermediate Language Code, IL code) a foglalási részeket, ezek kezdetét a C#-ban a *Monitor.Enter*, végét *Monitor.Exit* jelzi. Ezeket a részeket kell megkeresni a köztes kódban.

Ha az említett markerek rendelkezésre állnak, akkor az erőforrások megtalálása a következő lépés (ezek alkotják a gráf csúcsait). Az erőforrások a Monitor.Enter(erőforrás) hívások paraméterei. A paramétereket a veremből (stack-ből) kapják a függvények, tehát a stack nyomkövetése révén találjuk meg őket.

Miután felépült a gráf, a körök megtalálásával a lehetséges holtpontokat határoljuk be. Tehát a holtpont keresése leegyszerűsödött egyszerű kör keresésére a foglalási gráfban. A megfelelő megjelenítés a grafikus felhasználói interfész feladata, amelyről külön fejezetben írunk.



1. ábra – Egy foglalási gráf lehetséges holtponttal

Kezdetben adott volt a CsLint nevű eszköz, amely az ILReader nevű köztes kódolvasót használta, ami a .NET 1.0-ra működött. A CsLint feldolgozta a köztes kódot, és kimenetet generált belőle. A kimenet, annak a feldolgozó egységnek az outputja, ami a holtpontok megtalálásáért felelős. Az ILReader a .NET 2.0-ban lévő újításokat nem kezelte, illetve 2.0-s kódra kezdetben le sem futott. Ekkor került a figyelem középpontjába a CECIL [14], ami szintén egy köztes kód értelmező rendszer (a Mono Model Checker is ezt használja).

A CECIL működése sokkal stabilabb, és használható 2.0-ás keretrendszerre is. A feladat ezután az volt, hogy 3.0-ás, és a 3.5-ös keretrendszerre is működőképessé kellett tenni a rendszert, illetve a feldolgozó egység módosítása úgy hogy az eddig számára ismeretlen inputot is lekezelni tudja.

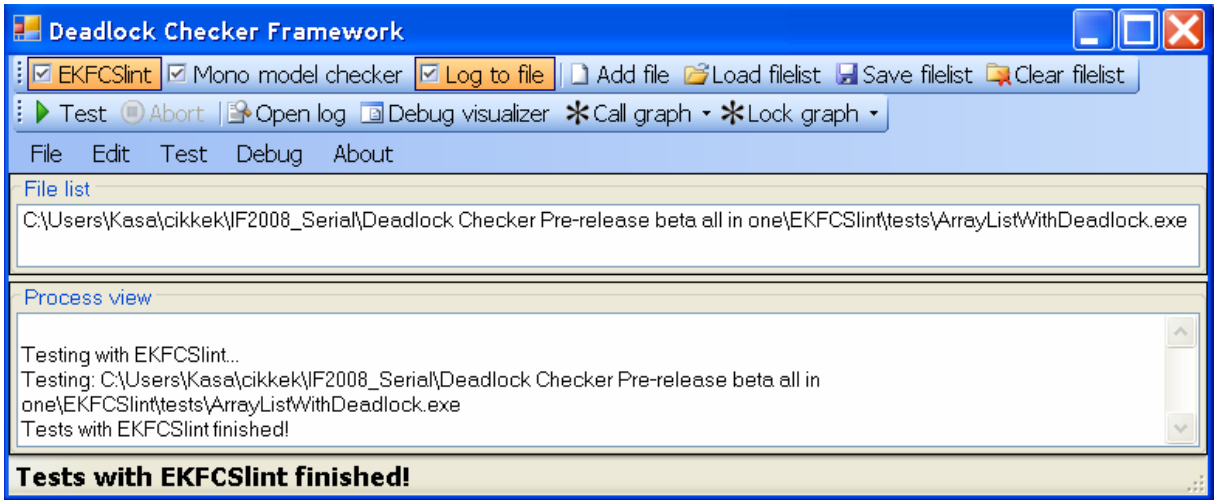
Elemeztük a .NET verziók közti különbségeket a köztes kód szemszögéből:

- az 1.0 és 1.1 között semmi különbség.
- a 2.0-nál bejött 5 új parancs.
- a 3.0-hoz nincs új fordító, tehát a bájtkód nem változott.
- a 3.5 ugyanaz, mint a 2.0, annak ellenére, hogy van új fordítóprogramja.

Miután a feldolgozóval sikerült megoldani az új nyelvi elemek felismerését, és értelmezését, ezen kívül a feldolgozó egység néhány hibáját javítottuk, a prototípus elkészítéséhez minden készen állt. A foglalási gráf megjeleníthető formára hozása teljesen a kutatócsoport fejlesztése.

4. A felhasználói felület

Az EKFCSLinthez felhasználói felület is készült. A fő cél az volt, hogy a deadlock checkert egyszerre több fájlra is le lehessen futtatni. Ehhez az „Add file” gombot kell használni. Egy listát állíthatunk össze a tesztelendő fájlokból, amit el is menthetünk, és bármikor visszatölthetünk. A teszt indítása előtt kiválaszthatjuk, melyik holtpont keresőt használjuk. A keretrendszer egyelőre az EKFCSLint es a Mono model cheker-t (új neve MoonWalker) [9] támogatja. A „Test” gomb megnyomással indíthatjuk a fájlok tesztelését. A tesztelési folyamat bármikor megszakítható. Ez megkönnyíti a hosszú távú tesztelést, és a több fájlból álló projektek tesztelését is.

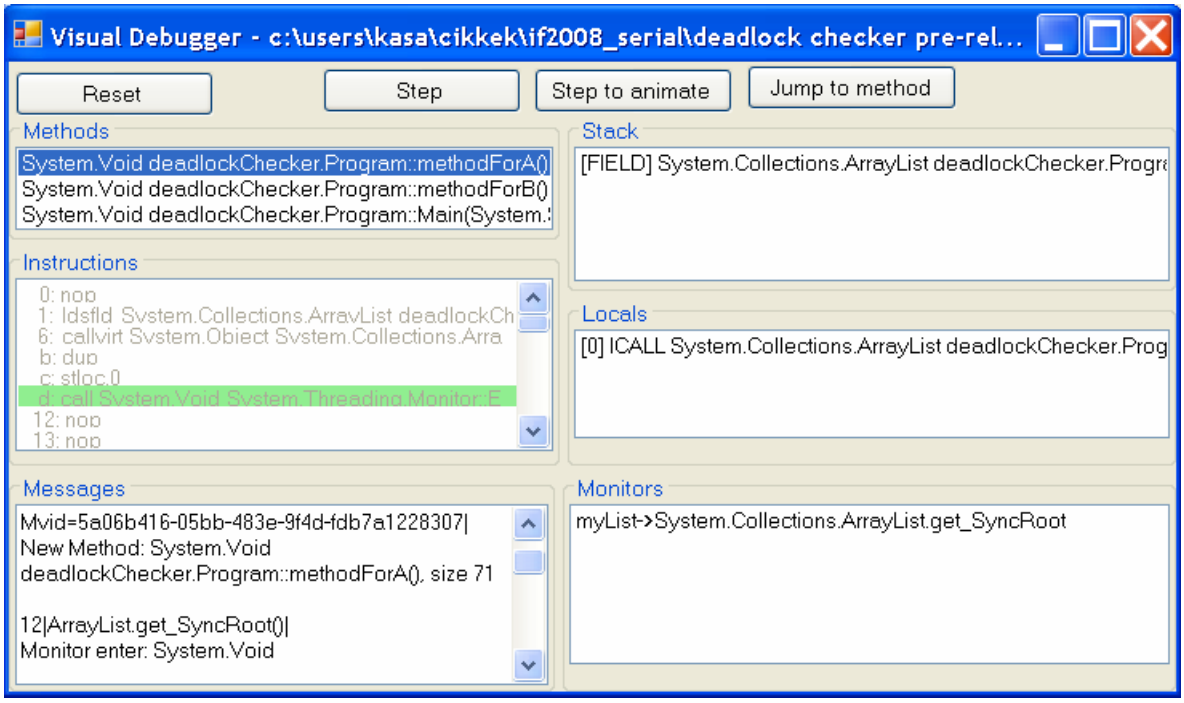


2. ábra – A holtpont kereső keretrendszer

A tesztek eredményét szöveges fájlba menti, így ezek később is elemezhetők, és összehasonlíthatóak korábbi tesztekkel. Ezekből állítja elő a hívási és a foglalási gráfot, amit kétféle megjelenítővel is megtekinthetünk. Ezek a GraphViz és a Glee gráf rajzolóval készülnek. Ezek helyét, illetve a holtpont keresők helyét az Edit / Configure paths… menüben állíthatjuk be.

A program tartalmaz egy „Debug Visualizer”-t is, ami lehetővé teszi a már tesztelt fájlok metódusainak, és azok IL kódjának megjelenítését, és az utasításokon történő lépkedést. Kijelzi a program szimulált futása alatt a stack és a monitorok állapotát. Lényegében azt szimulálja, hogyan járja be az EKFCSLint a tesztprogram IL utasításait. Léptethetünk egy

utasítást is, vagy akár egy kiválasztott metódus adott utasítására is ugorhatunk, és onnan folytathatjuk a léptetést. Ez a funkció a program kézzel történő vizsgálatához nyújt segítséget.



3. ábra – A Visual Debugger

A Visual Debuggerben a *Monitor.Enter* és a *Monitor.Exit* utasításokat kiemeli zöld színnel, hogy lássuk, hol járunk. Az aktuális sort kékkel jelzi. Magát a Visual Debuggert használtuk, hogy az eredeti CsLint rendszer hibáit megtaláljuk és javítsuk.

5. A program teszteléséhez használt tesztesetek

Többféle tesztesetet dolgoztunk ki, keverve a holtpontot tartalmazó és nem tartalmazó eseteket, illetve a .NET keretrendszerek közti különbségeket előhozó kódrészleteket. Feltételeztük, hogy azok a szálak, amiket a .NET maga indít a háttérben, tehát nem kifejezetten a programozó, pl. BackgroundWorker, az nem fog holtpontot okozni, így ezeket nem is vizsgáltuk. Így végül ezekkel a tesztesetekkel dolgoztunk:

- Egyszerű szálindítás deadlock nélkül, illetve deadlock-kal.
- Sok szálindítás deadlock nélkül.
- Két szál keresztbe hívása (deadlockos).
- ArrayList deadlock-kal.
- Verem és sor, mely deadlock-os, illetve zároltba próbál belerakni.
- Tömb, melyben deadlock van.
- Osztályok, melyben az egyik osztály zárolja egy másik osztály publikus mezőjét, melyet egy harmadik osztály is zárolni szeretne.
- DLL-ben található szál fut deadlock-ba.
- Generikus sor és lista, mely nem statikus és nem feltétlenül kerülnek deadlock-ba.

- Generikus lista deadlock-al.
- [Synchronization] paraméterrel ellátott osztály deadlock nélkül, illetve deadlock-kal.
- ThreadPool-ok lehetséges deadlock-kal.
- Étkező filozófusok problémája.
- Delegate-es szálindítás, ami deadlockba fut.
- Delegate-es szálindítás deadlock nélkül.
- Stack megvalósítása egy, illetve két szállal, deadlock helyzettel.

A tesztelés során a program 86%-ban sikeresen kiszűrte a holtpont gyanús eseteket. Ha nem volt holtpont, akkor nem adott hamis riasztást.

6. Összefoglaló

Elkészült egy prototípus elemző rendszer - EKFCSLint - ami az elődökhöz képest nagy előrelépés. A .NET keretrendszer jelenleg létező verzióira (2.0, 3.0, 3.5) elvégzi a holtpontkeresési procedúrát, és nagy biztonsággal megtalálja a lehetséges holtpont helyzeteket.

Az eszköz rendelkezik grafikus felülettel, ahol a keresések, illetve az eredményeik megjeleníthetők. Ez szintén újdonság az ismert rendszerekhez képest, mert a legtöbb ezek közül parancssori működésű, nehezen átlátható, és konfigurálható. Az eszköz használata egyszerű, és az eredmények könnyedén elemezhetőek, s transzformálhatóságuk révén jól illeszthetőek más rendszerek bemenetére.

A fentieken kívül van mód a hívási, és a foglalási gráfok kirajzoltatására, ami igen szemléletessé teszi az eredményt. A gráfok segítik mind a további elemzést, mind az átalakítását a teszteszköznek.

A rendszer jelenleg működőképes állapotban van, ipari környezetben is tesztelik magasan képzett programozók. A beérkező tapasztalatok elemzése után folyamatosan javítunk az eszközön.

Meg kell még említenünk, hogy a létrejött kutatócsoport munkájában a programtervező informatikus hallgatók és az intézet oktatói vettek részt. A projektlaborban végzett sikeres munka kiváló példája a hallgatók és a kutatók közös munkájának, s megalapozta a továbbfejlesztési lehetőségeket. Egy ilyen összetett probléma (projekt) elemzése, analizálása a megoldás megkeresése a prototípus elkészítése csak az előbb említett keretek között valósulhatott meg.

Irodalomjegyzék

[1] Kovács Emőd, Hernyák Zoltán, Radványi Tibor, Király Roland: Felsőoktatási Tankönyv- és Szakkönyvtámogatási Pályázat: *A C# programozási nyelv oktatása a felsőfokú oktatási intézményekben*, Digitális Tankönyv, 312 oldal, http://csharptk.ektf.hu, 2005.

[2] B.A. Botkin & A.F. Harlow: *A Treasury of Railroad Folklore*, Western Folklore, Vol. 13, No. 2/3, Published by: Western States Folklore Society, 1954.

[3] K.M. Chandy, J. Misra, and L.M. Haas, *Distributed Deadlock Detection*, ACM Transactions on Computer Systems, 1(2), 143-156, May 1983.

[4] A.K. Elmagarmid, *A Survey of Distributed Deadlock Detection Algorithms*, ACM SIGMOD Record, 15(3), September 1986.

[5] E. Knapp, *Deadlock Detection in Distributed Databases*, ACM Computing Surveys, 19(4), 303-328, December 1987.

[6] D.P. Mitchell and M.J. Merritt, *A Distributed Algorithm for Deadlock Detection and Resolution*, Proceedings, ACM Symposium on Principles of Distributed Computing, ACM, 282-284, 1984.

[7] Gerard J. Holzmann: *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley Professional, 2003.

[8] Konstantin Knizhnik, JLint, *Java code verifier* http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm#deadlock.sync_loop

[9] *MoonWalker, model checking .NET applications*, http://wwwhome.cs.utwente.nl/~ruys/moonwalker/

[10] http://javapathfinder.sourceforge.net/

[11] http://www.monodevelop.com

[12] http://www.mono-project.com/

[13] Konstantin Knizhnik: *CSLint: Deadlock Detector for C#.* 2003.

[14] P. Oscar Boykin: *Finding Deadlocks in .NET code, or CSLint with Cecil*, http://boykin.acis.ufl.edu/