Multidiszciplináris tudományok, 10. kötet. (2020) 4 sz. pp. 285-291 **https://doi.org/10.35925/j.multi.2020.4.32**

A MICROFRI C KERETRENDSZER ÁTALAKÍTÁSA CSOPORTOS FEJLESZTÉSRE

Krizsán Zoltán
egyetemi docens, Miskolci Egyetem, Informatikai Intézet, Általános Informatikai Tanszék
3515 Miskolc, Miskolc-Egyetemváros, e-mail: krizsan@iit.uni-miskolc.hu

Absztrakt

A uFRI a Fuzzy Rule Iinterpolation (FRI) számítási módszereket összefogó C implementáció, amelyet Bartók Roland implementált. A forráskód már több éve készül támogatva a Ph.D. kutatásait. Alapvetően FPGA-ra készült, de egy általános C forrás kód. Elsődleges szempont volt az elméleti kutatások igazolása, komplexitás mérés és beágyazott rendszerekben futtatása, illetve azok oktatása. A későbbiekben szeretnénk több embert is bevonni a fejlesztésbe, szeretnénk, ha kialakulna egy nagyobb felhasználó programozói közösség, ezért az iparban sikeresen alkalmazott sztenderdek segítségével szeretnénk biztosítani a kód minőségét, csapatmunka lehetőségét, ezzel növelve a bizalmat a kód iránt. Ezen cikk a későbbiekben ismertet néhány manapság használt ipari eljárásokat, folyamatokat, eszközöket, valamint a szükséges átalakításokat.

Kulcsszavak: *alkalmazás-fejlesztés, C, refactoring*

Abstract

The uFRI is a C implementation of computational algorithms developed by Roland Bartók. The source code has been developing for more years based on the Ph.D. research. Basically, it is built for FPGA, but it does not contain any specialty it can be used for any processor. The main goal was the verification of theoretical research, measuring the complexity, applying for embedded systems, and teaching. In the future we would like to develop the source code in small team, and increase the popularity in the developer community, making a real product. For this reason, we would like to use success standards and patterns to improve the code quality and possibility of teamwork. In this article we will present some industry patterns, flows, tools and suggest some modifications of existing source code as well.

Keywords: *application development, C, refactoring*

1. Bevezetés

Ebben a fejezetben elsőként a termék minőség és fejlesztői bizalom növelésének lehetőségeit, technikáit ismertetem. Általánosan leírom az iparban folyamatosan alkalmazott lépéseket. A konkrét, kiválasztott variánsokat pedig a következő fejezetben ismertetem.

Az implementált logika egy fuzzy logikát implementáló C forráskód, amiben néhány függvény található (koncepciója a [1] cikkben, alkalmazása a [2] folyóiratban olvasható).

Mivel idáig a programozói könyvtárat egy személy fejlesztette, mindig tudta, hogy hol mit talál, mi hogy működik. Számos projekt így kezdődik, de csapat esetén ez már nem használható. Amikor egy hipotézist szeretnénk igazolni, akkor minimális energia befektetéssel szeretnénk a minimális rendszert implementálni. Ha a hipotézist az implementáció igazolja, akkor már bevonhatunk újabb fejlesztőket

és bevezethetünk technikákat. Itt is ez történik, ezért csak most, utólag vezetünk be erőforrás igényes megszorításokat, lépéseket.

Egy termék folyamatos, csoportban történő fejlesztésének több speciális követelménye van. Mivel több fejlesztő sokkal több kódot épít a termékbe ezért nagyon fontos, hogy amiből kiindul egy feature fejlesztés az biztosan jó legyen.

Ha egy nem tökéletes kódbázisból indul ki a fejlesztő, akkor azt továbbfejlesztve (lehet, hogy pont azt a hibát kihasználva, továbbfejlesztve építi azt) csak mélyül a hiba, és a kijavításához is sokkal több energiára és időre van szükség. Például ha egy hibát csak 1 hónap után fedeznek fel, és 6 ember dolgozik rajta, akkor annak a javítása nagyon nagy feladat, és a javításig lehet, hogy nem is lehet folytatni a fejlesztést újabb feature-ökkel.

1.1. Verziókövető rendszer bevezetése

Egy fejlesztő esetén is, de több esetén mindenképp, szükséges a kódot egy központi verziókövető rendszerben tárolni. Egyrészt ez egy biztonsági mentése a kódnak, másrészt visszakövethető pontokat definiálhatunk a kód fejlesztése során. Minden új feature, hiba javítás során általában több forrást szükséges módosítani, komitokkal konzisztens állapotokat definiálhatunk. Így nyomon követhető a teljes kódbázis, vagy egyes forrás file-ok változása és annak oka. A git verziókövető rendszert választottuk, mert manapság a legnépszerűbb, elterjedt, így valószínű, hogy a későbbi fejlesztők is ismerni fogják.

Az alkalmazás fejlesztése során a git flow-t alkalmazzák a legtöbben, így mi is ezt alkalmaztuk. A git csak egy verziókezelő eszköz, a git flow pedig ajánlásokat definiál, hogyan alkalmazzuk a git eszközt. Általános elv, hogy a termék fejlesztésének van egy fő ága (git branch), ami védett, neve master. A termék különböző szállított verziói itt találhatóak. A fejlesztő csapat belső verziói a devel branch-en találhatóak. Ezekbe az ágakba tiltott a direkt kommit, csak leágazni lehet belőle majd új verzió elkészülte után, visszafűzni ebbe.

1.2. Források projekt management rendszere

Jellemzően minden projekt több erőforrásból és forrás fájlból áll. A végtermék olykor bináris állomány, olykor maga a forrás állomány (script nyelvek esetén). Egy projekten belül is lehetnek különböző szabályok, amik kapcsolódnak a termék létrehozásához, üzemeltetéséhez. Ezek miatt le kell írni a részek kapcsolatát, különböző környezetek fejlesztő (development), éles (production) és teszt (test) környezet kapcsolóit, sajátságait. Minden programozási nyelvnek vannak ajánlott, javasolt megoldásai, és vannak általános célúak is, amelyek bármely nyelv esetén alkalmazhatóak. Ezek folyamatosan változnak, megújulnak ezért ezeket a projekt elején érdemes rögzíteni, a későbbiek során cseréjük nehézkes lehet. Java esetén például régebben az Ant volt az egyedüli elérhető, manapság a Maven vagy Gradle a javasolt. A döntéskor érdemes figyelembe venni a meglévő és a leendő fejlesztők kompetenciáit, a projekt forrásainak a típusát, és jellegét, és az aktuális trendeket.

1.3. Kód teszt lefedettség ellenőrzése

A fejlesztőktől meg kell követelni az egység teszt kód lefedettség százalékos értékét. Erre több különböző ajánlás is van, ezekről a bővebben a [3] tanulmányban. Ezen lefedettség mértéke projektenként változhat, de meg kell egyezni az értékében a projekt legelején, és ehhez ragaszkodni kell mindvégig minden csapattagnak. Az egység tesztek az alkalmazás alap építő elemeit, az egységeket tesztelik, ami a C esetében a függvény.

Ha jók az egység tesztek, teljes a lefedettség, és minden feltöltés esetén lefut, akkor ezzel biztosítható a termék minősége. A feladat egyszerű: a fejlesztő egy olyan branch-ből ágazik le, aminek a teszt lefedettsége a megkövetelt. A kód tesztlefedettség mértéke nem csökkenhet, miután befűztük a feladat ágat.

2. A uFRI projekt bővítése, átalakításának részletei

Ezen fejezetben a bevezetőben ismertetett technológiák, eszközök beállítását és a már meglévő forráskód átalakítását ismertetem.

Elsőként a library forráskódja mellé létrehoztam egy teszt mappát, abban a tesztek forráskódját, egy *demo* mappát és abban a demóalkalmazások forráskódot is. Terveink szerint több demó forráskód is lesz a jövőben, segítendő a később csatlakozó felhasználókat. Miután több forráskód (demó és teszt) is lesz, érdemes bevezetni valamilyen projekt management eszközt.

Bármilyen fordítót, tesztelést is használunk, a forrás verziózása fontos feladat, erre a célra a git-et választottuk (bővebb információk a [4] helyen), mert elterjedt, és a legtöbb nyilvános központosított verziókezelő támogatja. A munkafolyamatnak pedig a Gitflow workflow-t használjuk és javasoljuk (leírás, ajánlások összefoglalása a [5] és [6] helyen).

Jelenleg a forrás fájlból egy lépésben bináris állományt generáltatunk a gcc fordítóval, amit feltöltünk a beágyazott rendszerekbe. Ez a jelenlegi megoldás megfelelő az eddigi célnak, de a jövőben szeretnénk, hogy minél szélesebb körben használják a megoldásunkat, akár más programozó nyelv használói is. Kézenfekvő megoldás tehát, ha egy dinamikusan linkelhető programozói könyvtárat választunk a projekt céljaként. Ez a Windows rendszereken a dll, Linux, Mac operációs rendszereken az so állományok. Ezen állományokra szinte bármely programozói nyelvből lehet hivatkozni (Java Nativ interface, C#, Python, ...), de akár kész alkalmazásokból is (Matlab, Octave, ...).

2.1. Egységtesztek bevezetése

Miután a csoportos fejlesztés esetén nagyon fontos az egység tesztelés, és a teszt lefedettség, be kellett vezetnünk valamilyen C könyvtárat tesztelő lehetőséget. Egyik lehetőség, hogy a létrehozott C könyvtárat valamilyen C++ teszt keretrendszerrel teszteljük, vagy C teszt rendszert használunk. Ha C++ tesztelést használunk, (manapság a legnépszerűbb a googletest), akkor számos kényelmes lehetőség áll rendelkezésre, és a tesztelés könnyűvé válik. Ha C teszt rendszerek közül választunk, akkor a tesztelés nehézkes, kevés lehetőség kínálkozik. Arra számítunk azonban, hogy a legtöbb fejlesztő nem C++ lesz, így a C teszt kód megértése bármely fejlesztőnek sikerülni fog. A tesztek valójában a rendszerünk egy dokumentációja, ezért fontos, hogy a későbbi felhasználók tudják azokat értelmezni.

Átvizsgálva a C programozó könyvtárakat a libcester-t választottam (dokumentáció, letöltés [7]), mert a használatához csak egy header fájlt kell beilleszteni a kódunkba, és lehetővé válik a globális függvények mock-olása is. A manapság népszerű teszt keretrendszer lista a [8] helyen található.

C programok írásánál a kódunkat globális függvények segítségével tudjuk kisebb részekre bontatni. Az üzleti logika implementálása során használhatunk már megírt külső függvényeket, vagy általunk létrehozott saját függvényeket. A probléma összetett:

- Egyrészt a tesztek írása során követelmény, hogy lehetőleg minél jobban lokalizáljuk a hibát, azaz csak azok a tesztek bukjanak, amelyek a hibás üzemi kódot tesztelik. Ha egy üzemi metódus egy másik üzemi vagy külső metódust használ, és a másikban van hiba, ekkor minkét teszt sikertelen lesz, emiatt a hiba felderítése nehezebb lesz. Emiatt találták ki az üzemi metódusok mock-olását. Legtöbb teszt keretrendszerben a teszt metódus elején nyilatkozunk, hogy ebben a

teszt metódusban a használt, kimock-olt elemek hogyan működjenek, mit adjanak vissza, milyen paramétereik vannak.

- Másrészt a C nyelvben minden függvény globális, nem lehet felül definiálni, lecserélni teszt idejére, „kimockolni”. Más modern objektum orientált programozói nyelv (Java, C++, JS, …) esetén a tesztelendő üzemi metódus virtuális, ami felül definiálható, vagy JS esetében script részlet, ami kicserélhető a teszt idejére.

Ezen problémákra ad közösen megoldás a libcrest és a gcc fordító együttes használata. Egyrészt a libcrest biztosít két függvény makrót, amivel ki lehet mock-olni egy üzemi függvényt: ezek a CESTER_MOCK_SIMPLE_FUNCTION és a CESTER_MOCK_FUNCTION.
Másrészt jelenleg ez a megoldás csak gcc fordító esetén működik Windows és Linux operációs rendszerek estén, mert ott a fordító biztosítja a –wrap kapcsolót, ami globális függvények cseréjét teszi lehetővé.

2.2. Projekt management eszköz bevezetése

A CMake management eszközt választottam, mert platform független, és mert képes Visual Studio workspace file-okat is generálni a saját leíróiból (koncepciója a [9] cikkben olvasható, dokumentáció és leírás a [10] helyen található). Nem szerettük volna elveszíteni sem a Windows-on Visual Studio fejlesztőket, sem a Linux-on/Mac-en make és gcc alapokon fejlesztőket.

A CMake valójában többet is nyújt ennél, hiszen build, teszt és csomag management eszköz is. Saját leírója van, amiben deklaratív módon leírjuk a forrásokat, eredmények létrehozásának lépéseit. Ebből az általános leíróból a CMake parancssori vagy grafikus eszköze valamilyen generátor segítségével makefile-t Visual Studio workspace file vagy Ninja file-t is tud generálni. Itt lehetőségünk van CMake változókat megadni (külső könyvtárak, include file-ok helyét, ..), amik alapján a generálás lefut. Miután ez a generálás lefutott, a következő lépés már CMake független, a generált új leíró alapján a megfelelő külső alkalmazás létrehozza a szükséges kimeneteket. A CMake valójában csak szöveges fájlokból más szöveges állományokat hoz létre, de közben használhatunk CMake csomagokat is.

A CMake esetében CMakeList.txt file-ok írják le a projektünket. A következő példa az alkalmazás fő szerkezetét írja le, amely a gyökér mappában található.

```
project(five
        VERSION 0.1
        DESCRIPTION "FIVE library implementation & examples"
        LANGUAGES C)
add_subdirectory(src)
add_subdirectory(demo)
add_subdirectory(tests)

if (UNIX)
        set(EXTRA_LIBS ${EXTRA_LIBS} m)
endif (UNIX)

include(CTest)
enable_testing()
```

A definíciós rész (1. sor) megadja az alkalmazás verzióját, leírását és a programozói nyelvet. A további sorok kijelölik az almappákat, amelyekben újabb leíró file-ok találhatóak.

Az azt követő szekcióban hozzáadjuk az EXTRA_LIBS CMAKE változó értékéhez az m (matematikai library) értéket, ha Linux alatt fordítunk. Ez a lista vagy üres vagy kívülről beállított. Itt hozzá-

fűzzük a matematika library-t (-lm kapcsoló), mert sajnos Linux és gcc estén alapértelmezetten nem linkelik a matematikai könyvtárat, itt ezt explicit kell kérni.

Az utolsó szekció biztosítja a projektünk számára a tesztelés lehetőségét. A CMake rendszerben alapértelmezetten elérhető a CTest modul. Ez a modul, mint parancssori eszköz használható, futtatja és aggregálja a tesztjeinket. A beállításait később a megfelelő helyen ismertetem.

Az *src* mappában találhatóak az alkalmazás forrásai, és az újabb CMakeLists.txt állomány.

```
add_library(five STATIC uFRI.c uFRI_helpers.c toString.c)
target_link_libraries(five PUBLIC ${EXTRA_LIBS})
target_include_directories(five PUBLIC ../include)
```

Itt előírjuk a szükséges kimenet nevét, és típusát. Az előzőekben ismertetett programozói könyvtár létrehozását az *add_library* CMake parancs hatására majd olyan makefile fog generálóni, ami a gcc fordítót a megfelelő kapcsolókkal és paraméterekkel hívja meg. Ennek a parancsnak gyakorlatilag két adatot kell megadni:

- A könyvtár (kimenet) neve. Ennek hatására fogja a gcc a folyamat végén létrehozni a five.so, vagy five.dll állományokat.
- A források listája. Ezekből a forrásokból object fájlokat készít, amiket a linker fog majd egy so vagy dll file-okká összefűzni.

A második sorban a szükséges függőségeket linkelteti majd az új állományunkhoz.

Az utolsó sor pedig megadja a saját készítésű header fájlok mappáját, így használhatjuk azokat a saját forrásainkban az *#include "header.h"* előfordító direktíva segítségével.

A tests mappában pedig a következő CMakeLists.txt található:

```
include_directories("${PROJECT_SOURCE_DIR}/include")
include_directories("${CESTER_HOME}/include/")

add_executable(five_base_test five_base_test.c speed_distance_test_system.c)
target link libraries(five base test PRIVATE
        five
        ${EXTRA_LIBS}
        "-Wl,--wrap=calloc")

add_test(five_base_test five_base_test)
```

Itt az első két sor definiálja a használható header fájlok mappáját. Az első sor a saját header helyéről nyilatkozik, a második a külsőről. A libcester használatának egyetlen feltétele, hogy a teszt forrásokba be kell illeszteni az egyetlen cester.h állományt. Ez az egyetlen külső paraméter, amit be kell juttatni a CMake feldolgozónak. Ez egy CMake változó. Első lépésként hozzuk le a libcester forrását valahová, majd adjuk meg ezt kívülről a CESTER_HOME útvonal típusú változójába. A megadása történhet cmake parancssori argumentumában, melyet a következőképpen lehet hívni:

```
cmake -DCESTER_HOME:FILEPATH=/home/devel/c/libcester ..
```

Ennek hatására a CMake futása közben a CESTER_HOME CMake változóban már ott lesz az a mappa, és bárhol használható a CMakeList.txt állományokban.

A következő részben a teszt állományok definíciója látható, ezt kell minden teszt állományra megadni:

- Egy egyszerű futtatható állomány definíció, amelynek megadjuk az állomány nevét és a források listáját.
- A tesztelendő Five programozói könyvtárat mindenképpen hozzá kell linkeltetnünk az állományhoz, erre szolgál a *target_link_libraries és a five*, ${EXTRA_LIBS}. Ha szeretnénk hasz-

nálni a libcester és gcc által nyújtott mock-olás lehetőségét, akkor a *target_link_libraries* listájában meg kell adni a gcc –wrap kapcsolóját is. Itt most a calloc gyári függvény kitakarását kérjük, mert a tesztekben ezt a globális függvényt használtuk.

- Végül hozzáadjuk a végleges futtatható állományt a tesztek közé. Ezek után a CTest tudni fogja, hogy ezt az állományt is futtatnia kell, ha tesztelni szeretnénk. A CTest lefuttatja az összes tesztet, összegyűjti a tesztek statisztikáit (sikeres, sikertelen), es visszatérési értéke 0 lesz, ha minden teszt sikeresen lefutott. Ezt a visszatérési értéket figyelik a CI/CD rendszerek, és ez alapján minősítik az építési folyamatot is sikeresnek.

3. Összefoglalás

A uFive forráskódjának átírása, egység tesztek implementálása, CMake keretrendszer, és bizonyos konvenciók bevezetése után a kód minőség garantálható, a későbbi fejlesztők bizalma így növelhető. Ezek bevezetése után, ha egy új fejlesztő csatlakozik, akkor valószínű, hogy hamarabb be tud illeszkedni, hiszen ipari szenderdeket használunk, és azok külső helyen jól vannak dokumentálva. Valamint az új források bevezetése is egyszerűsödik. A tesztek bevezetésével a forrás megértése, a kód minősége is jelentősen javul.

Későbbi fejlesztésben a CI/CD rendszer bevezetése, és beállítása lesz az elsődleges cél, valamint értesítések küldése a fejlesztő csapatnak, több dokumentáció és minta alkalmazások fejlesztése. Miután a körülmények adottak, egy egységesített keretrendszert szeretnénk kialakítani, amelyben a Five módszer implementációja csak egy lehetséges probléma megoldási alternatíva lesz. Egy olyan keretrendszert, ahová bárki beilleszthetí a saját megoldását, benchmark problémáját, ahol az egységesítést a megfelelő projekt szerkezet, a CMake, CTest és a tesztek biztosítják. A teljes forrás a [11] helyről letölthető.

Irodalom

[1] Bartók, R., Vásárhelyi, J.: *Two methods for autonomous robot obstacle sensing and application programming interface for fuzzy rule interpolation*, Dan, Popescu; Dorin, Şendrescu; Monica, Roman; Elvira, Popescu; Lucian, Bărbulescu (szerk.) 2017 18th International Carpathian Control Conference (ICCC) Craiova, Románia: IEEE, (2017) pp. 87-92. Paper: 7970376, 6 p **https://doi.org/10.1109/CarpathianCC.2017.7970376**

[2] Bartók, R., Bouzid, A., Vásárhelyi, J., L. Kiss, M.: *Wall and object detection with FRI and bayes-classifier for autonomous robot,* Lecture Notes in Mechanical Enginnering F12 (2017) pp. 383-389., 7 p. **https://doi.org/10.1007/978-3-319-51189-4_34**

[3] Runeson, Per. *A survey of unit testing practices*. IEEE software, 2006, 23(4):22-29. **https://doi.org/10.1109/MS.2006.91**

[4] Software Freedom Conservancy: *Git verziókezelő hivatalos oldala*. https://git-scm.com/

[5] Atlassian: *Git flow hivatalos oldala*, https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[6] Github közösség: *Javaslatok a Git flow alkalmazására*, https://guides.github.com/introduction/flow/

[7] Libcester fejlesztő csapat: *libcester hivatalos oldala*, https://github.com/exoticlibraries/libcester.

[8] Wikipedia: *teszt keretrendszerek listája*, https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

[9] Martin, K., Hoffman, B.: *An open source approach to developing software in a small organization*, IEEE Software 2007, 24(1):46-53. **https://doi.org/10.1109/MS.2007.5**

[10] CMake fejlesztő csapat: *CMake hivatalos oldala*, https://cmake.org/

[11] Five fejlesztő csapat: *uFive programozói könyvtár repository*, https://bitbucket.org/krzzzzz/five/src/master/