Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Szoftver verifikáció és validáció

Házi feladat

Írásos összefoglaló Doina Bucur és Marta Kwiatkowska *On Software Verification for Sensor Nodes* [1] című munkájáról.

Készítette: Kazi Sándor Antal (GARZXN)

2013. január 15.

1. Feladat

A szerzők olyan, standard C nyelven írt programok verifikációjával foglalkoznak, amelyek szenzorok vezeték nélküli hálózatára tervezettek. A verifikáció célja – szokásos módon – a szoftverek hibáinak minimalizálása már azok célkörnyezetben való bevetése előtt. Tehát a feladat hatékony verifikáció lehetővé tétele ebben a nem szokványos környezetben.

1.1. Jellegzetességek

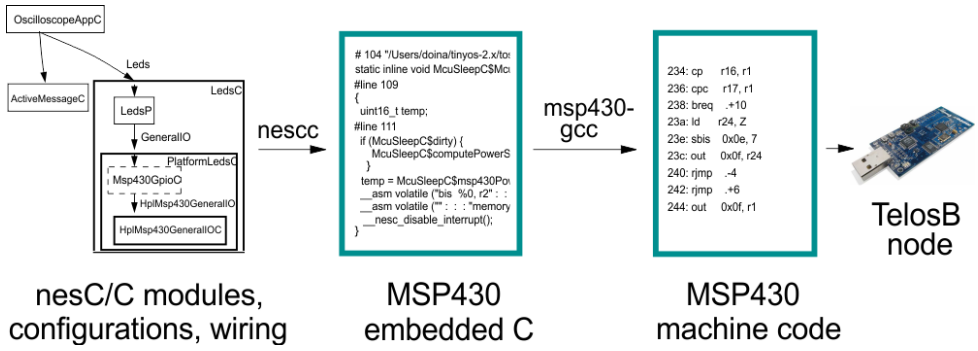
A szenzorrendszerek esetében igen gyakran nagyobb bonyolultságú logikát szükséges megvalósítani, mintsem, hogy gépi kód szintjén implementálható legyen a megoldás úgy, hogy a kód megfelelően hibakereshető és változáskezelhető legyen.

Emiatt bonyolult funkcionálitást ellátó beágyazott rendszerek kapcsán gyakran előfordul magas szintű nyelvek beágyazott változatainak alkalmazása, illetve operációs rendszerek (pl.: TinyOS[1], μC/OS) használata is. Ilyen változkezelési és egyéb kódmenedzsment elvárások esetében általános, hogy a programozók a rendszer belső logikáját valamely magas szintű nyelven írják.

1.2. A feladat speciálitása

A feladat felismerése a szerzők által nesC[2] és TOSThreads [2] implementációk által ihletett. Előbbi a C nyelv egy TinyOS-specifikus kiegészítésének elnevezése, míg utóbbi a TinyOS-re készült szálkezelő csomag. Mindkettő esetében igaz, hogy az beágyazott eseményvezérelt környezetet igyekszik ötvözhetővé tenni a magas szintű nyelvekkel és a szálkezeléssel, elosztott működéssel.

A szerzők által megvalósított feladat nehézsége abban rejlik, hogy a verifikálni kívánt szoftver két elkülönülő, egymás utáni fázis kimeneteként áll elő. Az első lépés az implementáció lefordítása a beágyazott MSP430 C nyelvre, majd ebből az MSP430 gépi kódra (1. ábra), amely a szenzor *mote*-ra kerülő tényleges kód.



1. ábra. A tényleges futtattandó kód elkészülésének fázisai. [1]

[1] http://www.tinyos.net/
[2] http://nescc.sourceforge.net/

1.3. Verifikáció

A feladat tehát ebben a kétlépcsős fordításban biztosítani a verifikáció lehetőségét. A verifikáció fordítási időben történhet, és – a szerzők véleménye szerint – bár több megoldás elérhető, amely a verifikáció elvégzésére hivatott, sok esetben saját programozási és/vagy specifikációs nyelv ismeretét feltételezik. Mindezek mellett az ilyen eszközök gyakran az alábbi gyengeségek valamelyikével rendelkeznek:

- komplex adatstruktúrák támogatásának hiánya
- végtelen állapotú programok esetén nem garantált, hogy a verifikáció valaha is leáll (befejeződik)
- a verifikáció elfogadhatatlanul hosszúra nyúlhat még végesállapotú programok esetén is - szenzorhálózatok elosztott mivolta miatt a sok kommunikációs lehetőség, sok lehetséges állapotot hordoz magában.

2. A javasolt megoldás

A szerzők által megvalósított verifikációs eljárás inputja a platformspecifikus beágyazott C kód (embedded MSP430 C), amely feltételezetten automatikusan generált egy magasabb nyelvű programkódból. A választott tesztalanyok ugyan TinyOS alkalmazások, de a verifikációs eljárás OS-független (de nem platformfüggetlen).

2.1. Eszköz és módszer

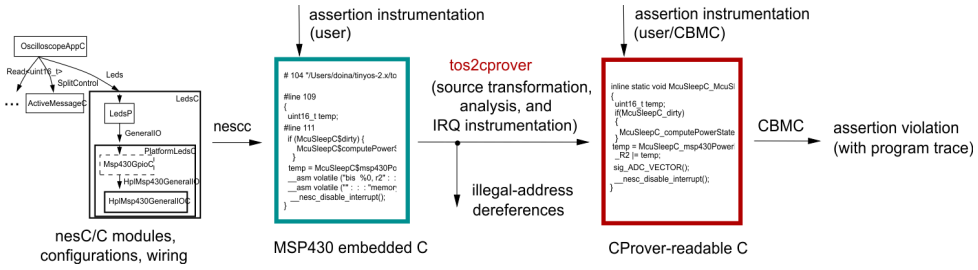
Az elkészült eszköz automatikusan képes lefordítani egy MSP430 C nyelvű, opcionálisan aszinkron interruptokat is tartalmazó kódot standard C nyelvűre. Ezt úgy valósítja meg, hogy a direkt memóriahozzáféréseket és az MCU memóriatérképét képezi le ANSI-C-re (*tos2cprover* lépés a 2. ábrán). A hardveres interruptok verifikációját emulált környezettel teszi lehetővé a rendszer, az előfordulások száma minimalizálásra került egy részleges rendezésen (partial order reduction) alapuló módszerrel – az állapottér méretének kézben tartása érdekében. Fontos, hogy mindeközben minden átalakítás úgy hajtódik végre, hogy a program szemantikája garantáltan megmarad, de az állapottér minimális, de legalábbis minél kisebb.

Az így előállt szekvenciális program már átadható a CMBC-nek [3]. Ez egy verifikációs eszköz szekvenciális ANSI C programokhoz, amely a memóriával kapcsolatos problémák (túlindexelés, null pointer problémák, stb.) detektálásán kívül alkalmazás-specifikus jellemzők (például regiszter- perifériaállapotok) ellenőrzésére is alkalmas.

2.2. Környezet, input és verifikáció

A cikk részletesen bemutatja a használt beágyazott C nyelv sajátosságait, a TinyOS bemutatni érdemes jellemzőit és a tényleges verifikációt végző CMBC-t is.

A CMBC [3] ANSI-C programok formális verifikációjára hivatott, a korlátos modellellenőrzésen (Bounded Model Checking, BMC) alapul. A készítők a alkalmazhatóságra fektették a hangsúlyt, ezért majdnem



2. ábra. A verifikációs módszer (eszköz-láncolat) lépései. [1]

minden ANSI-C nyelvi elem támogatott a CMBC által (például dinamikus memória allokálás, rekurzió, float és double adattípusok). A felhasználó számára a verifikáció nagymértékben automatizált, csak a BMC korlátját kell inputként szolgáltatnia a programkódon kívül.

2.3. Eredmények

Az eljárást három (*Blink*, *Sense*, *TestDissemination*) teszt inputtal tesztelték, mindhárom esetben igaz, hogy nem sikerült új hibára találni, de ez önmagában nem zárja ki a módszer életrevaló mivoltát – viszont sajnálatos módon bizonyítja a rossz tesztválasztást (előzetesen valószínűleg már más eszközzel verifikált kódról van szó). Valamelyest sikerként könyvelhető el az a tény, hogy a módszer kimutatta a TinyOS néhány – aktuálisan létező – ismert hibáját, ezt a szerzők meg is teszik.

A kiértékelés alapos (leszámítva a hibamentességet), a tesztek komplexitását is figyelembe veszi. Ehhez definiálja magát a komplexitást is az azt meghatározó tényezőkön keresztül:

- sorok száma (LOC) – a már átalakított változatban
- a különálló ciklusok száma – amelyhez a CMBC (BMC) számára korlátot kell megadni
- a várt hardver megszakítások száma

2.4. Kitekintés

A cikkben bemutatásra kerül *survey* jelleggel több más, hasonló környezetben használható verifikációt megvalósító, vagy azt támogató eszköz. Ilyen a SafeTinyOS, amely memória- és tipizálási hibák detektálására, jelentésére és minimális szintű kezelésére (pl.: újraindítás) alkalmas már alkalmazásban lévő programok esetében. A SafeTinyOS egyik nagy hátránya, hogy plusz kódírási munkát igényel a használata: *trusted cast* és *access bound* megadásokra is szükség van, azaz jelölni kell bizonyos esetekben, hogy azt tényleg úgy szerettük volna implementálni, illetve, hogy a mérethatárok hol is vannak.

A TinyOS egy másik hátránya, hogy a memóriahibára újraindítással válaszol. A Neutron névre keresztelt kiegészítés a TOSThreads API segítségével ilyen implementáció esetében csupán szálakat indít újra – amennyiben ez lehetséges.

A statikus hibadetekció egy gyenge erejű lehetőségeként említésre kerül a szimuláció, mint verifikációs eszköz. A TOSSIM lehetővé teszi implementáció alapú szimulált futtatását TinyOS alkalmazásoknak, így biztosítva egy lehetőséget hibadetekcióra.

Minden esetben rámutat arra, hogy a kérdéses feladat esetében egy-egy eszköz hogyan használható, miért nem használható, vagy miért nem érdemes használni (a javasolt módszer megfelelőbb).

3. Konklúzió

Az elkészült módszer – bár csak hosszú alkalmazása során bizonyítható használhatósága – szinte iskolapéldája a már ismert eljárások (Partial Order Reduction, CMBC) és a feladatspecifikus lehetőségek ötvözésének. Emellett mind az alkalmazási környezet, mind a részleteinek, mind pedig az alternatívák bemutatásának tekintetében alapos.

Irodalomjegyzék

[1] Doina Bucur and Marta Kwiatkowska. On software verification for sensor nodes. *J. Syst. Softw.*, 84(10):1693–1707, October 2011.

[2] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Răzvan Musăloiu-E, Philip Levis, Andreas Terzis, and Ramesh Govindan. Tosthreads: thread-safe and non-invasive preemption in tinyos. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 127–140, New York, NY, USA, 2009. ACM.

[3] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer Berlin Heidelberg, 2004.