Egy erősen párhuzamosított számításokat végző alkalmazásszerver verifikációja a Plural nevű eszköz segítségével

Magyar nyelvű összefoglaló a *Lightweight Verification of a Multi-Task Threaded Server: A Case Study With The Plural Tool* [1] című cikkről

Pósfai Gergely

December 12, 2014

1 Bevezetés

A Plural nevű alkalmazás objektumok állapottér-ellenőrzéséhez (*typestates*) és hozzáférési jogosultságainak ellenőrzéséhez (*access permissions*) nyújt támogatást.

A(z) [1] cikk egy esettanulmányt mutat be, amely során a Plural nevű eszköz segítségével hajtották végre az **MTTS** nevű multiprocesszes, többszálas, erősen párhuzamosított számításokat végző alkalmazás specifikációját és verifikációját. Az MTTS alkalmazás Java nyelven íródott, a Novabase nevű cég fejlesztette, azonban a Plural segítségével történő ellenőrzéseket egy független csoport végezte.

A kutatás célja verifikálni az MTTS alkalmazás bizonyos elvárt tulajdonságait, továbbá felderíteni, mennyire alkalmas egy ilyen feladatra a Plural nevű eszköz, mik az eszköz alkalmazásának korlátai.

2 A Plural specifikációs nyelve

A Plural eszköz egy Eclipse plugin, amely egy specifikációs nyelvet nyújt a vizsgálni kívánt rendszerrel kapcsolatos különböző követelmények, ellenőrizendő feltételek megfogalmazásához. A specifikációs nyelv kétféle elemből épül fel: állapotokból (*typestates*) és hozzáférési jogosultságokból (*access permissions*). Az állapotok segítségével megadható, hogy egy objektumon mikor, milyen műveletek hajthatóak végre, míg a hozzáférési jogosultságok absztrakt fogalmak, amelyek jelzik, hogy egy adott referencián keresztül milyen módon férhetünk hozzá az általa címzett objektumhoz (read/write), ill. milyen egyéb referenciák létezhetnek, amelyek ugyanahhoz az objektumhoz nyújtanak hozzáférést. A Plural ötféle hozzáférési jogosultságot definiál:

1. **Unique(x):** garantálja, hogy x az egyetlen referencia a hivatkozott objektumra, így x-nek olvasási és írási joga is van.

2. **Full(x):** az x referencia rendelkezik olvasási és írási jogokkal is. Továbbá engedélyezi egyéb olvasási jogokkal rendelkező referenciák létezését, azonban írási jogokkal bírókét nem.

3. **Share(x):** hasonló a Full-hoz, azzal a kivétellel, hogy a további referenciák írási jogokkal is rendelkezhetnek.

4. **Pure(x):** x csak olvasási jogokkal rendelkezik a hivatkozott objektumon. Továbbá megengedi egyéb (akár kizárólag olvasási, akár olvasási-írási) referenciák létezését az adott objektumhoz.

5. **Immutable(x):** x számára, ill. minden más, a hivatkozott objektumra mutató egyéb referencia számára kizárólag olvasási (read-only) jogot biztosít. Az immutable referencia garantálja, hogy a hivatkozott objektumra mutató minden egyéb referencia szintén immutable típusú.

A Plural-specifikációkat Java annotációkkal írhatjuk le. Az osztályok lehetséges állapotainak megadására a *@ClassStates* annotáció szolgál, míg a metódusokra vonatkozó feltételeket a *@Perm* annotációval adhatjuk meg. A *@Perm* annotáción belül a *requires* attribútummal adhatjuk meg a metódus végrehajtásához szükséges erőforrásokat, az *ensures* attribútummal pedig meghatározhatjuk

a metódus végrehajtásával keletkező erőforrásokat. Az erőforrások közötti „és" kapcsolat a * karakterrel definiálható.

Az annotációk használatának egy példáját a(z) 1. ábrán láthatjuk, amely a Task osztály specifikációját mutatja. A Task osztály egy generikus feladat-feldolgozó osztály az MTTS rendszerben. Az osztály belső információi egy *MttsTaskDataX* típusú, *data* nevű változóban kerülnek tárolásra. Az osztálynak négy lehetséges állapota van: **Created, Ready, Running, Finished**. A kezdeti állapot a **Created**. Az osztály akkor kerül **Ready** állapotba, ha kap valamilyen adatot a futáshoz. A **Running** állapot egy köztes állapot, melybe akkor kerül az objektum, ha éppen fut az *execute*() metódus, míg a **Finished** állapotba akkor kerül, ha az *execute*() metódus lefutott, azaz a megadott adatokat feldolgozta. A *setData*() metódus annotációjában a „#0! = *null*" kifejezés megköveteli, hogy a függvény első paramétere nem lehet *null* értékű.

```
@ClassStates({
 @State(name = ``Created'', inv = ``data == null''),
 @State(name = ``Ready'', inv = ``data != null''),
 @State(name = ``Running'', inv = ``data != null''),
 @State(name = ``Finished'', inv = ``data == null'')
})
public Task implements AbstractTask {
  private MttsTaskDataX data;

  @Perm(ensures = ``Unique(this) in Created'')
  public Task() { ... }

  @Perm(requires = ``Full(this) in Created * #0 != null'',
        ensures = ``Full(this) in Ready'')
  public void setData(MttsTaskDataX data) { ... }

  @Perm(requires = ``Full(this) in Ready'',
        ensures = ``Full(this) in Finished'')
  public void execute() throws Exception { ... }
}
```

Ábra 1: A Task osztály specifikációja Plurallal.

A példában az annotációk garantálják, hogy egy *task* egyszer fog lefutni (hiszen kizárólag az egyszer végrehajtott konstruktor viszi **Created** állapotba a *taskot*), továbbá a *setData*() előbb fog végrehajtódni, mint az *execute*() (mivel kizárólag a *setData*() metódus viszi **Ready** állapotba az objektumot).

3 Az MTTS rendszer struktúrája

Az MTTS egy feladat-elosztó rendszer, amely a beérkező feladatokat várakozási sorokba csoportosítja, majd feldolgozó szálakat rendel hozzájuk, amelyek elvégzik a feladatok végrehajtását. Az MTTS egy tipikus kliens-szerver architektúrájú alkalmazás, amely három nagy részre bontható:

1. **TaskRegistration**: interfészt biztosít a kliensek számára feladatok regisztrálásához, melyek adatbázisban kerülnek tárolásra.

2. **QueueManager**: a feldolgozó szálak kezeléséért felelős, amelyek a várakozási sorokból egyesével kivett feladatokat végrehajtják.

3. **RemoteOperationControl**: interfészt biztosít a kliensek számára a feladatok feldolgozási folyamatának monitorozására, vezérlésére.

4 Az MTTS specifikációja és verifikációja Plural segítségével, az elemzés korlátai

Az alábbiakban bemutatásra kerülnek az MTTS rendszer verifikációja során megvizsgált főbb rendszerkövetelmények, amelyek alapvető fontossággal bírnak egy ilyen típusú, erősen párhuzamosított számításokat végző, kereskedelmi rendszer esetében.

4.1 Mutual Exclusion

Az MTTS rendszerben a kölcsönös kizárás megvalósítására az $IMutex$ osztály szolgál, amely két fontos metódussal rendelkezik: az *acquire()* metódus megszerzi a szükséges zárat, amelyet a *release()* metódus enged el. Ennek a Plural specifikációval történő leírását a(z) 2. ábra szemlélteti. Az *acquire()* az egyetlen metódus az osztályban, amely az **Acq** állapotba visz, ill. a *release()* metódus az egyetlen, amely az **FStat** állapotba visz. Az $acquire()$ és a $release()$ metódusok közrefogásával biztosítható a kölcsönös kizárás egy kritikus blokk számára. Ennek ellenőrzésére a Plural specifikációkban a **Full** jogosultság meglétét kell ellenőriznünk az $IMutex$ típusú zárobjektumra a kritikus blokkok metódusainak annotációjában.

```
@Full(requires="FStat", ensures="Acq")
public abstract void acquire() { }

@Full(requires="Acq", ensures="FStat")
public abstract void release() { }
```

Ábra 2: Kölcsönös kizárást megvalósító metódusok specifikációja Plurallal: zár megszerzése, ill. elengedése.

4.2 Nem-elengedett zárak törlése

Az MTTS rendszer rendelkezik egy metódussal, amely zárak törlését végzi. Helyes működés esetén nyilván csak olyan zárakat lehetne törölni, amelyek már el vannak engedve, hiszen ha egy lefoglalt zárat törölnénk az deadlockhoz, vagy más abnormális viselkedéshez vezethetne. Annak biztosítására, hogy egy lefoglalt zár ne kerüljön törlésre, a törlést vegző metódus Plural annotációjának *requires* attribútumában az alábbi feltétel került definiálásra: *"Full(#0) in **NotAcq**"*. Ez a feltétel biztosítja, hogy a törlendő zár (a függvény első paramétere, melynek jelölése: *#0*) nincs lefoglalva, azaz nem birtokolja jelenleg egyetlen szál sem.

4.3 Standard Java könyvtárak specifikációja

Az MTTS szerver adatbázisban tárolja a végrehajtandó feladatokat. Mivel a Plural nem biztosítja az adatbázis-műveletekhez szükséges standard Java osztályok specifikációját, ezért ezen osztályok annotálását is el kellett végezni az MTTS rendszer verifikációjához. Ennek végrehajtásával az is láthatóvá vált,

hogy az MTTS rendszer megfelelő módon hajtja végre az alapvető adatbázis-műveleteket, azaz pl. mindig rendelkezésre áll nyitott adatbázis-kapcsolat az adatbázis-műveletek végrehajtásakor.

A(z) 3. ábrán szereplő példában látható az *MttsConnection* nevű osztály specifikációjának egy részlete, amely definiál egy **Connection** állapotot két alállapottal: **OpenConnection** és **ClosedConnection**. Az *open*() metódus az adtabáziskapcsolatot leíró objektumot **OpenConnection** állapotba viszi, míg a *close*() metódus **ClosedConnection** állapotba.

```
@Refine({
 @States(dim=``Connection'',
         value={``OpenConnection'', ``ClosedConnection''})
})
public abstract class MttsConnection {
  @Perm(ensures = ``Unique(this) in OpenConnection'')
  MttsConnection() { }

  @Full(value=``Connection'', ensures=``OpenConnection'')
  public abstract void open() throws java.sql.SQLException;

  @Full(value=``Connection'', ensures=``ClosedConnection'')
  public abstract void close() throws java.sql.SQLException;
}
```

Ábra 3: Az alapvető adatbázisműveletek specifikációja Plurallal.

4.4 Null értékek ellenőrzése

Az egyik leggyakoribb adatellenőrzési feltétel, hogy egy adat értéke ne legyen *null*. Ennek egy példája a *setName*() metódus, amely a(z) 4. ábrán látható módon került annotálásra annak a hibának az elkerülése érdekében, hogy a metódus paraméterének értéke ne lehessen *null*. (A *#0* kifejezéssel hivatkozhatunk a metódus első paraméterére.)

```
@Perm(requires = ``#0 != null'')
public void setName(String name) { ... }
```

Ábra 4: Hibás *null* értékek ellenőrzése Plural specifikáció segítségével.

4.5 Az MTTS szerver indítása, leállítása

A MTTS alkalmazás fő osztálya az *MttsServer* osztály, amely tartalmazza a szerver indítását és leállítását végző *start*() és *stop*() metódusokat, továbbá definiál három változót, amelyek a rendszer fő komponenseit alkotják: *OpControlRemote, TaskRegistrationRemote* és *queueManager*.

A rendszer vizsgálata során néhány, az indítással és leállítással kapcsolatos alapvető követelmény is specifikálásra került a Plural segítségével, melyek elengedhetetlenek a rendszer konzisztens működéséhez, mint pl.: a szerver pontosan akkor van **ServerStart** állapotban, ha a három fő komponense rendre **TStart, CStart**, ill. **QStart** állapotokban vannak. Továbbá a szerver pontosan akkor van **ServerShutdown** állapotban, ha a három komponens rendre **TShutdown, CShutdown**, ill. **QShutdown** állapotokban vannak. A két

említett követelmény közül az első teljesült, azonban a másodikról kiderült a vizsgálat során, hogy pusztán a *queueManager* változóra vonatkozó követelmény teljesül, így ezt a működést hibaként jelentették a szerver fejlesztői számára. A hiba felderítése pusztán a kódok vizsgálatával igen nehézkes lett volna az osztály nagy mérete és bonyolultsága miatt.

4.6 A Plural eszköz korlátai

A Plural eszköz hatékonyan alkalmazható bizonyos rendszerkövetelmények teljesülésének ellenőrzésére, azonban sok korlátja van az alkalmazhatóságának.

Ha egy zárat egy szál egymásba ágyazottan többször is megszerezhet, akkor nyilván kell tartani egy egészértékű változót, amelyet mindig növelünk, ill. csökkentünk a zár egy újabb megszerzésekor, ill. elengedésekor. Ennek leírása a Plural segítségével nem lehetséges, mivel a Plural nem nyújt lehetőséget egész számokkal való aritmetikai műveletek végrehajtására.

A Plural nem nyújt lehetőséget elérési vizsgálatra (*reachability analysis*) sem, azaz nem képes olyan kérdésekre választ adni, mint pl. ha egy zárobjektum **Acq** (lefoglalt) állapotban van, akkor eléri-e valaha a **NotAcq** (elengedett) állapotot. Emiatt a Plural nem képes deadlockok elkerülésének verifikációjára sem.

Emellett a Plural nem képes a metódusokban alkalmazott ciklusfeltételeket kezelni, amelyek a ciklus bizonyos számú lefutása után válnak igazzá/hamissá.

A(z) 1. ábrán bemutatott *execute*() metódus során a $Task$ objektum **Ready** állapotból **Running** állapotba, majd **Finished** állapotba kerül. Mindkét állapotváltás a metóduson belül megy végbe, így a **Running** állapot pusztán egy köztes állapotként jelentkezik a metódus futása közben. Az ilyen köztes állapotok által hordozott információt nem képes a Plural felhasználni, így a köztes állapotokra nem lehet ellenőrzési feltételeket megfogalmazni. (A metódus kódját író szoftverfejlesztők hozzáférhetnek a köztes állapotértékekhez, azonban a Plural erre nem képes.)

5 Összefoglalás

A vizsgálatok során a Plural egy hatékony eszköznek bizonyult komplex rendszerek bizonyos tulajdonságainak verifikációjára. Segítségével sikerült a meglehetősen nagy méretű MTTS rendszer egyes problémáit felderíteni, néhány alapvető fontosságú tulajdonságát verifikálni, mint pl.: kölcsönös kizárás helyes kezelése, *null* értékek vizsgálata, adatbáziskapcsolatok megfelelő használata stb. Azonban az eszköz számos korláttal rendelkezik, így a rendszer elvégzett verifikációja semmiképpen sem mondható teljeskörűnek.

A Plural új funkciókkal történő kiegészítése jelenleg is folyamatban van. A továbbfejlesztési célok közt szerepel a hozzáférési jogosultságok Petri-hálószerű szemantikával történő leírásának lehetővé tétele, valamint az állapotelérési vizsgálatok (*reachability analysis*) végrehajtásának lehetősége is.

Referenciák

[1] Néstor Catano and Ijaz Ahmed. Lightweight verification of a multi-task threaded server: A case study with the plural tool. In *Formal Methods for Industrial Critical Systems*, pages 6–20. Springer, 2011.