Forgatókönyv alapú tesztelés UML/OCL viselkedési modell alapján

Referátum

Eredeti mű címe: Scenario-based testing from UML/OCL behavioral models[1]

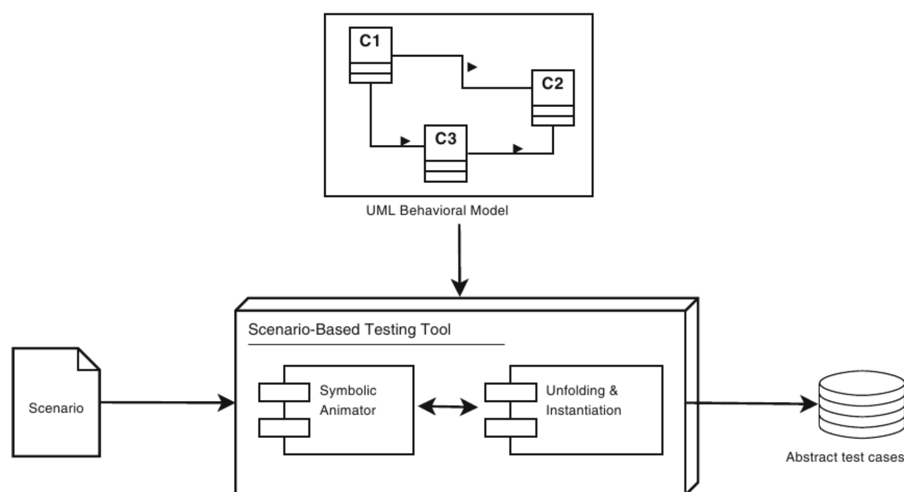
Referátumot készítette: Tóth Márton József

tmarton@iit.bme.hu

2012. december 10.

1. Absztrakt

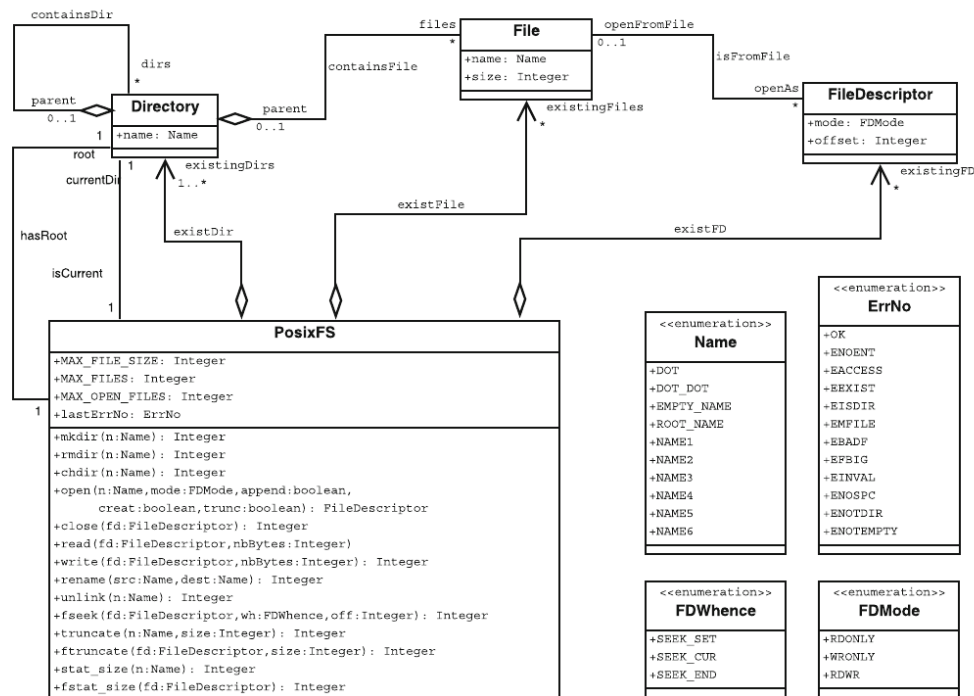
A cikkben leírt eljárás POSIX fájlrendszerek teszteléséhez kíván teszteseteket létrehozni, egy UML/OCL alapú viselkedési modell szerint. Bemutatja a strukturális lefedettségen alapuló automatikus teszteset generátorok hiányosságait és egy funkcionalitáson alapuló kiegészítő eljárást javasol a tesztek létrehozásához. A funkcionalitások leírásához forgatókönyvek készülnek, melyek megadják az elvárt operációk egy sorrendjét és a futtatás közben elérni kívánt állapotokat. Ezen forgatókönyvekből és a rendszer statikus modelljéből a módszer képes legenerálni a szükséges teszteseteket. Az 1. ábrán a létrehozott rendszer összefoglaló képe látszik.



1. ábra Forgatókönyv alapú teszteset generálás

2. A létrehozott modell

A tesztek létrehozásához a cikk írói a *TestDesigner* nevű eszközt használták, melyet a *Smartesting*[1] cég fejleszt. Ez az eszköz képes OCL viselkedési leírásokkal kiegészített UML modellek alapján teszteseteket létrehozni. Mivel az eszköz csak egy részhalmazát támogatja az UML modellezési képességeinek (nem lehet benne öröklés, nem lehet dinamikusan létrehozni objektumokat), ezért egy speciális rendszerleírást kellett létrehozniuk a szerzőknek. Az elkészített modell a 2. ábrán látható. Az API teszteléséhez egyetlen osztály készült, mely az összes tesztelni kívánt metódust tartalmazza. Az egyes metódusok funkciójának leírása OCL kiegészítéssel történt. Az *mkdir* parancs leírását a 3. ábra szemlélteti.



2. ábra A fájlrendszer modellje

---

[1] http://www.smartesting.com

```
context PosixFS::mkdir(n:Name): integer
pre: not(existingDirs->select(parent = NULL)->isEmpty())
post:
  // check name emptyness
  if (n = Name::EMPTY_NAME) then
    ---@AIM: FAIL_ENOENT
    lastErrNo = ErrNo::ENOENT and result = -1
  else
    // check if name already exists in current directory
    if  n = Name::DOT or n=Name::DOT_DOT or
            n = Name::ROOT_NAME or
            currentDir.parent.dirs->exists(name=n) or
            currentDir.parent.files->exists(name=n)
    then
      ---@AIM: FAIL_EEXIST
      lastErrNo = ErrNo:EEXIST and result = -1
    else
      ---@AIM: VALID_NAME
      // check if maximal number of files not reached
      if existingDirs->select(parent<>NULL)->size()+
         existingFiles->select(parent<>NULL)->size()=MAX_FILE
      then
        ---@AIM: FAIL_ENOSPC
        lastErrNo = ErrNo::ENOSPC and result = -1
      else
        // directory is created
        ---@AIM: SUCCESS
     let new_dir:
         existingDirs->select(parent=NULL)->includes(new_dir)
     in
       new_dir.name = n and new_dir.parent = currentDir and
       currentDir.dirs = currentDir@pre->union(Setnew_dir) and
       lastErrNo = ErrNo::OK and result = 0
      end if
    end if
  end if
```

3. ábra Az mkdir parancs OCL leírása

Az *mkdir* függvény meghívásának előfeltétele, hogy legyen még létrehozható könyvtár (az eszköz korlátozásai miatt nem lehet dinamikusan létrehozni objektumokat, így a könyvtárakat reprezentálók már a rendszer indulásakor létrejönnek, csak nincsenek „beépítve” a könyvtár-hierarchiába). A lefutás utáni utófeltétel pedig biztosítja, ha a megfelelő bemeneti paraméterekkel lett meghívva az utasítás, akkor létrejön a kért könyvtár.

3. Tesztesetek létrehozására a statikus modell alapján

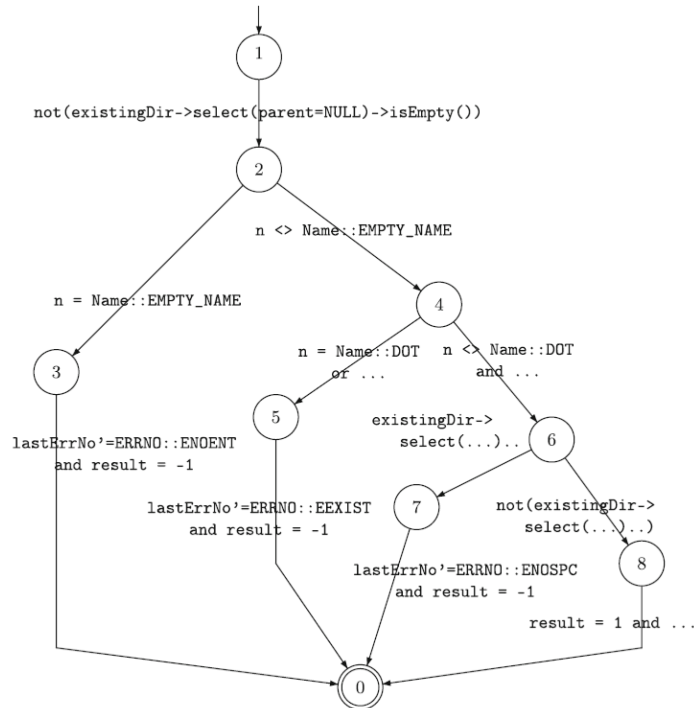
A fejlesztés további lépéseként létrehoztak olyan tesztcélokat, melyeket a teszteseteknek meg kellett vizsgálniuk. Ezeket a célokat az OCL kódban az „@AIM” címke jelenti. Tehát olyan tesztesetet kellett generálni, melyek eljutnak a kérdéses kódrészlethez. A generálási folyamatot a *TestDesigner*-el végezték el.

A generálás első lépéseként az eszköz megállapítja azokat a perkondíciókat, amelyek a tesztcélok eléréshez szükségesek. Ezután az állapotteret szélességi bejárással feltérképezve létrehozza a megfelelő teszteseteket.

A generálás azonban több esetben is sikertelen volt. A problématér exponenciális mérete miatt az eszköz nem volt képes minden esetben levezetni a modellből olyan teszteseteket, melyek kielégítették volna a célokat. Más esetekben, amikor a célok „vagy" kapcsolattal voltak összekötve, az eszköz egyben kezelte őket és csak a legrövidebb úton megvalósítható lefutást vette be a tesztek közé, ami nem minden esetben kielégítő. Mivel az eszköz a teszteseteket függetlenül kezeli sok esetben előfordult olyan, hogy több teszteset is lefedte az állapottér ugyanazon részét, így fölösleges redundanciát vitt a tesztelés folyamatába. A legnagyobb problémát azonban az jelentette, hogy az eszköz nem teszi lehetővé a rendszer dinamikus tulajdonságainak vizsgálatát, hacsak, nem adunk meg további adatokat a modellről, egészítjük ki a vizsgálatainkat felhasználói inputtal vagy nem helyezünk el mesterséges célokat a teszt leírásában. Ezek azonban a modell kiterjesztését igényli, illetve az eszköz működésének mélyebb ismeretét feltételezik. A cikk tanulsága szerint ezek a problémák nem csak az alkalmazott eszköz esetén állnak fenn, hanem több más gyakorlatban is használatos szoftverrel (UML_CASTING, HOL-TestGen, Gatel, Promela) kapcsolatban is felmerülnek.

4. Szimbolikus végrehajtás

Ezeknek a problémáknak a megoldására az UML modellek szimbolikus végrehajtását alkalmazták, így a felhasználónak nem kell a parancsok argumentumát megadnia, a paramétereket változókkal helyettesítik. Az egyes operandusokat formálisan pre és post kondíciókkal írják le, melyek meghatározzák a rendszer állapotát, melyben az adott metódus meghívható illetve leírják annak állapotát a metódus lefutása után.



4. ábra az mkdir parancs vezérlési folyamata

A szerzők definiáltak egy lefutást, viselkedést, a vezérlési gráfban. Ezt jelölték b^{op}-al. Formálisan $act\left(b^{op}, \overline{V}, \overline{I}\right)$ írja le, hogy b^{op} viselkedés milyen feltételek mellet valósul meg, ami függvénye a $\overline{V}$ – a rendszer állapotváltozói – és $\overline{I}$ – az operáció bemenete – paramétereknek is.

$subst\left(b^{op}, \overline{V}, \overline{I}, \overline{V'}, \overline{O}\right)$ adja meg a kapcsolatot, hogy b^{op} viselkedés megvalósulása esetén a rendszer milyen $\overline{V'}$ állapotba kerül és milyen $\overline{O}$ kimenetet szolgáltat a metódus. $C_{\overline{X}}$ jelöli az $\overline{X}$ változókra alkalmazott megkötéseket.

Tehát egy viselkedés megvalósulhat, ha a rendszer állapotváltozóira és a bemenetre adott megkötések megegyeznek lefutáshoz szükséges feltételekkel, azaz $C_{\overline{V}} \wedge C_{\overline{I}} \wedge act\left(b^{op}, \overline{V}, \overline{I}\right)$ teljesül. Ekkor a rendszer új állapotának a következő feltételeknek kell megfelelnie $C_{\overline{V'}} = C_{\overline{V}} \wedge C_{\overline{I}} \wedge act\left(b^{op}, \overline{V}, \overline{I}\right) \wedge subst\left(b^{op}, \overline{V}, \overline{I}, \overline{V'}, \overline{O}\right)$.

A szimbolikus végrehajtás így elvégezhető egy tételbizonyítóval, amely képes kiértékelni a logikai formulák kielégíthetőségét és meg tudja adni az utasításoknak egy olyan szekvenciáját, mely eléri a kérdéses tesztcélokat.

A szimbolikus végrehajtás alkalmazásához az UML modell mellett az adatokat, illetve az OCL kifejezéseket is át kellett alakítani logikai kifejezésekké. Az objektumok leképzése atomi formulákkal történt, OCL elemeket pedig logikai függvényekkel adták meg.

5. Forgatókönyvek megadása

A forgatókönyvek funkcionális szempontból írják le az elvárt viselkedést és így útmutatást adnak a tesztesetek generálásához. A szerzők a forgatókönyv leírást négy szemantikai rétegre osztották, melyek a leírás különböző aspektusait adják vissza.

Az első rész a *modell* réteg. Ez felelős a viselkedési modell és a tesztcélok közötti kapcsolat definiálásáért, a megfelelő predikátumok, állapotváltozók, és függvényhívások használatával.

A második rész a *szekvencia* réteg. Ez biztosít lehetőséget az operációhívások sorrendjének megadására reguláris kifejezések, szimbolikus vagy konkrét paraméterek használatával.

A harmadik rész a *direktíva* réteg. Ez teszi lehetővé, hogy a teszteset generáló számára útmutatást adjunk. Ezt úgy tehetjük meg, hogy olyan paramétereket definiálunk, melyek meghatározzák az állapottér bejárásának a módját, így befolyásolva a generálandó teszteseteket.

A negyedik rész a *definíciós* réteg. Ez a legfelső réteg, ez biztosít lehetőséget a szimbolikus változók megadására.

$$
\begin{aligned}
&MaxOpenFiles :=\\
&\qquad\quad let\ _file : File\ be\ one_of\ (sut.existingFiles),\\
&\qquad\quad _fd : FileDescriptor\ be\ (sut.existingFd)\ in\\
&[sut :: open\ /_w\{@AIM{:}FILE_CREATED_OPENED\}]\\
&\qquad \leadsto (sut.existingFiles \rightarrow exists(_file)\\
&\qquad\qquad and\ sut.existingFD \rightarrow select(desc\\
&\qquad\qquad\qquad \mid not\ desc.openFromFile.oclIsUndefined())\\
&\qquad\qquad\quad \rightarrow\ isEmpty()\\
&\qquad\qquad and\ _file.openAs \rightarrow exists(_fd)\\
&\qquad\qquad and\ _fd.openFromFile = _file)\\
&.sut :: close(_fd)\\
&\qquad \leadsto (sut.existingFd \rightarrow isEmpty())\\
&.[sut :: open(_file.name, _, _, _, _)/_w\{@EXISTS_FILE_OPEN\}]*\\
&\qquad \leadsto (sut.existingFD \rightarrow size() = sut.MAX_OPEN_FILES)\\
&.[sut :: open(_file.name, _, _, _, _)]\\
&where\ @EXISTS_FILE_OPEN\\
&\quad is\ seq(@AIM{:}FILE_EXISTS, @AIM{:}FILE_OPENED)
\end{aligned}
$$

5. ábra MaxOpenFiles teszteset

Az *5. ábra MaxOpenFiles teszteset* látható leírás azt a tesztesetet adja meg, amikor egy fájlt többször próbálunk megnyitni, mint ahány fájl egyszerre a rendszerben nyitva lehet. A *_file* illetve *_fd* tagok szimbolikus változók konkrét értékek nélkül. Először megadja, hogy a tesztnek meg kell nyitni illetve lezárni a fájlt, ezzel biztosítva, hogy az valóban létezik, majd megnyitja *MAX_OPEN_FILES* alkalommal és még egyszer. Az utolsó hívásnak hibát kell jeleznie.

6. Szemantika

Egy forgatókönyv szemantikáját a lehetséges lefutások halmazával definiálták, melyet öt lépésben állítottak elő. Először a reguláris kifejezéseket véges automatává alakították. Ez az automata megadja a lehetséges futások halmazát ∑, amely tulajdonképpen az automata által elfogadott szavak halmaza. Ezután a ∑ halmaz elemeiben lévő paramétereket helyettesítették azok összes lehetséges értékeivel, így kapták ∑' halmazt. Ebben a halmazban azon lefutásokat melyek egyes állapotok előtt ugyanazzal a prefixummal rendelkeznek, tehát a lefutásoknak ugyanaz az előfeltétele a ∑'' halmazba csoportosították. Ezután minden ilyen csoportból egy elemet kiválasztva képeztek teszteseteket.

7. Eredmények

Az elkészített tesztesetekkel két alkalmazást teszteltek le. Az egyik egy 2.6.20-16-generic Ubuntu kernel volt, a másik pedig egy Java implementációja a POSIX szabványnak. A Linux kernel tesztje során számos különbségre derült fény a modell és az implementáció között, azonban ezek mind a szabványban megengedett implementációfüggő részek voltak. A Java megvalósítás tesztje számos programozási hibát fedett fel, úgy mint tömb túlindexeléseket vagy null pointerre való hivatkozásokat.

8. Irodalomjegyzék

[1] K. Castillos, F. Dadeau és J. Julliand, „Scenario-based testing from UML/OCL behavioral models," *International Journal on Software Tools for Technology Transfer (STTT),* %1. kötet13, %1. szám5, pp. 431--448, 2011.