Konfiguráció modellezés
Informatikai technikák laboratórium I.

Szatmári Zoltán
szatmari@mit.bme.hu

Ujhelyi Zoltán
ujhelyiz@mit.bme.hu

2013. április 30.

Az informatikai infrastruktúra tervezése és üzemeltetése egyre komplexebb feladatokat ad az üzemeltetők számára. Összetett infrastruktúrák üzemeltetését elosztott környezetben, egyre több csomópont segítségével lehet csak megvalósítani, melynek beállítási és folyamatos karbantartási igénye túlmutat a kézi módszerek alkalmazásán. Ezres nagyságrendű csomópontból álló hálózat üzemeltetése már csak központosított, automatizált infrastruktúra menedzsment segítségével képzelhető el.

Az egyre terjedő Cloud Computing megközelítés szerint a fizikai erőforrások háttérbe szorulnak, kizárólag a szolgáltatásokkal és azok konfigurációjával kell a rendszermérnöknek foglalkozni. A megközelítés szerint nem is kell feltétlenül ismernünk a szolgáltatásunk fizikai elhelyezkedését, csupán konfigurációját kell megadnunk, a többi a rendszer automatikusan elintézi.

A szolgáltatáskészlet ilyen módszerekkel történő megvalósítása során cél, hogy azt gyorsan és hatékonyan összeállíthassuk és a későbbiekben minél nagyobb fokú automatizálás mellett üzemeltessük. Az automatizálás szükséges eleme egy struktúrált, a célnak megfelelő *konfigurációs modell* elkészítése, mely alapján a konfiguráció programozottan végezhető.

A mérés során egyszerű példán keresztül bemutatásra kerül egy konfigurációs modell készítése és az alapján történő automatizálás lehetősége.

A mérési feladat egy virtuális gépekből és azon futó szolgáltatásokból álló infrastruktúra leírásához használt konfigurációs modell elkészítése. A modell segítségével megadhatóak egyszerűsített formában a virtuális gépek paraméterei és a rajtuk futó szolgáltatások, ami alapján automatizált folyamat létrehozza a kívánt infrastruktúrát kézi beavatkozás nélkül.

A konfigurációs modell a mérés során egy egyedi, egyszerűsített konfiguráció leíró nyelv lesz, melynek pontos szintaxisát kell elkészítenünk. A modellek megalkotásához egy szöveges editort, amelyhez az *Eclipse* fejlesztőkörnyezetbe integrálódó *Xtext* technológiát használjuk.

A konfiguráció által definiált infrastruktúrát az *OpenVZ* virtualizációs technológia segítségével hozzuk létre, mely egy egyszerűen automatizálható, kis erőforrásigényű, virtuális gépek futtatására és menedzsmentjére alkalmas rendszer.

A segédlet ezen technológiákba nyújt egy rövid bevezetőt.

Tartalomjegyzék

1. Szöveges editorok készítése az Xtext használatával

A gyakorlatban sokszor van szükség különböző szöveges állományok szerkesztésére, például programozási nyelvek, konfigurációs állományok, vagy akár wiki lapok forrásai esetén. A szerkesztésük megkönnyítéséhez sokféle szolgáltatást kell megvalósítani, kezdve az egyszerű forráskódszínezéstől gépelési hibák jelzésén keresztül az áttekintő nézetig, amely a legfontosabb elemeket hierarchikus struktúrában jelzi. Ezen funkciók hatékony együttes megvalósítása komplex feladat. Az Xtext keretrendszer célja, hogy megkönnyítse az efféle szöveges szerkesztők készítését az Eclipse fejlesztőeszköz belsejében.

A keretrendszer elsősorban azt várja el a felhasználótól, hogy adjon meg egy nyelvtant, aminek segítségével értelmezni lehet az éppen szerkesztett szöveget, majd a nyelvtan alapján képes legenerálni egy egyszerű szerkesztőt. A generált szerkesztő alapszinten támogatja a hibajelzést, forráskódszínezést, kód váz megjelenítés és automatikus kiegészítés (*content assist*, javaslatok nyújtása szerkesztett elemek befejezésére) szolgáltatásokat, és lehetőséget biztosít a fejlesztőnek, hogy Java kód készítésével tovább finomítsa a megoldást.

A keretrendszer az Eclipse Modeling Framework (EMF) technológiára építve készít egy modellt (és metamodellt) a nyelvhez. Ez a modell az alapja a megvalósított szolgáltatásoknak, és emellett programozottan elérhető, bejárható és módosítható. A módosítások egyszerűen visszaírhatók a szerkesztett szöveges fájlba.

A következőkben bemutatjuk, hogyan lehet az Xtext segítségével ilyen szöveges editorokat készíteni, míg a segédlet végén egy egyszerű mintapélda található.

2. Nyelvek megadása Xtext környezetben

Nyelvnek nevezzük egy véges karakterkészletből generálható, véges hosszúságú karakterláncok egy halmazát. A nyelvek megadásának szokásos módja *generatív nyelvtan* definiálása, amely alkalmazható szabályok egy listáját adja meg, amely szabályok alkalmazásával az összes elfogadott karaktersorozat előállítható.

Fontos észrevenni, hogy a szabályok alkalmazása során kétféle elemet lehet megkülönböztetni:

Terminális szimbólum: olyan karakter (esetleg karaktersorozat), amelynek egy az egyben meg kell jelennie a nyelvben.

Nemterminális szimbólum: olyan elem, amely a nyelvben önmagában nem jelenhet meg, célja, hogy összetettebb elemeket lehessen leírni benne.

A szabályok alkalmazása egy behelyettesítésként képzelhető el. A szabály megadásakor a behelyettesítendő nemterminális szimbólumot a szabály *bal oldala* tartalmazza, a behelyettesített szimbólumokat pedig a *jobb oldala*.

Az Xtext környezetben a nyelvtani szabályoknak teljesíteniük kell a következő feltételeket[1]:

[1] Más elemző generátorok (*parser generator*) esetleg ennél megengedőbb formátumú nyelvtani szabályok is használhatóak.

- A szabályok baloldalán pontosan egy darab nemterminális szimbólum található[2].
- A szabályok jobboldalán terminális és nemterminális szimbólumok tetszőleges sorrendben és számban helyezkedhetnek el.
- A terminális szimbólumokat idézőjelbe (akár ', akár ") kell tenni. Minden mást a rendszer nemterminális szimbólumnak vesz.
- Van néhány előre definiált nemterminális szimbólum, ilyen az `INT`, az `ID` vagy a `STRING`.
- Minden előre nem definiált nemterminális szimbólumnak szerepelnie kell egy szabály baloldalán.

1. példa. *A szabályok szerkezetét a következő példa ismerteti, amely azt mutatja meg, hogyan lehet egy IP címet, mint nemterminális szimbólumot leírni:*

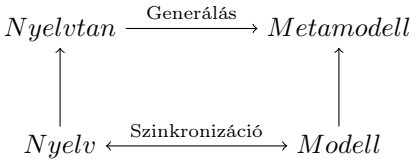
```
IP:     INT "." INT "." INT "." INT;
```

A szabály két oldalát `:` *választja el, a baloldali szimbólum definícióját tartalmazza a jobb oldal. Az* `IP` *szimbólum definíciója összesen hét elemet tartalmaz: négy* `INT` *és három* `"."` *elemet, ezeket kötött sorrendben.*

2.1. Nyelvtanok és metamodellek

Az Xtext nyelvtanleírásai a terminális és nemterminális szimbólumokon kívül tartalmaznak egyéb elemeket is, amelyek azért szükségesek, hogy ebből automatikusan egy belső modellt lehessen generálni. Pontosabban a nyelvtan alapján generál egy metamodellt, amelynek a példányait képes szinkronban tartani a szöveges változattal[3].

A nyelvtan és a modellek közötti kapcsolatot az 1. ábra mutatja be. A nyelvtanból generálható a metamodell; a nyelvtannak megfelelő nyelveknek pedig megfeleltethető példánymodellje a metamodellnek.



1. ábra. Kapcsolat a nyelvtanok és metamodellek között

Mivel az egyes nemterminális szimbólumokból képzett konkrét nyelvi kifejezéseket szeretnénk a modellben is eltárolni, ezért meg lehet adni, hogy az érték melyik attribútumba kerüljön. Erre a célra szolgál az `=` szimbólum: a szimbólum bal oldalán szerepel a név, míg a jobb oldalán egy típus. Többszörös multiplicitású attribútumok esetén, például elemek listájánál a `+=` szimbólum segítségével

[2]Formális nyelvek elméletét ismerők számára: környezetfüggetlen nyelvtanokat kell megadni.

[3]Egészen pontosan az Xtext nyelvtanleírásai több információt hordoznak, mint amit általában egy nyelvtannak tartalmaznia kell, több elem is felhasználható, amely a modellgenerálást szabná testre.

jelöljük, hogy egy listához szeretnénk egy elemet hozzáadni. A nemterminális szimbólum attribútumhoz hozzárendelését a a 2. példa ilusztrálja.

2. példa. *A következő kód szolgál a Place nemterminális definiálására. A végső nyelvben a következő szintaxissal szeretnénk egy Place elemet megadni:* `place p1;`*, ahol a* `p1` *egy tetszőleges egyedi azonosító.*

A nemterminális `name` *attribútuma ID típusú lesz, azaz egy azonosító, ami gyakorlatilag egy karakterlánc. Ugyan explicit módon nem kell megadni, de a rendszer figyel rá, hogy az azonosítók egyediek legyenek.*

```
Place: 'place' name=ID ';' ;
```

Az így definiált nyelvtanrészlethez a következő modell tartozik: keletkezik egy `Place` *osztály, amelynek van egy* `name` *nevű attribútuma, amely egy szöveges azonosítót fog tartalmazni.*

A fenti példában egy a nyelvben új elemet adunk meg, ezt a továbbiakban *tartalmazásnak* nevezzük.

Ettől eltérően lehet már létező elemekre hivatkozni is a típusnév szögletes zárójelbe tételével, melyet a 3. példa mutatja be. Az elemző az ilyen hivatkozásokat automatikusan megkeresi és feloldja, ha nem definiált elemre hivatkozunk, akkor hibát jelez. Ezen túl az automatikus kiegészítés használatakor fel fogja ajánlani a már definiált elemek azonosítóját is.

3. példa. *A nyelvben így szeretnénk megadni, ha egy Token egy Place-ben található:* `token in p1;`*, ahol a* `p1` *egy máshol definiált Place elem azonosítója.*

A nyelvtanban a hivatkozott nemterminálist szögletes zárójelben kell feltüntetni:

```
Token: 'token in' place=[Place] ';' ;
```

2.2. További elemek a nyelvtanban

A szabályok tetszőleges részéhez rendelhető multiplicitás, amely a következő értékek egyike lehet:

- Pontosan 1. Nem kell jelölni.
- Legfeljebb 1. Jelölés: `?`.
- Bármennyi ($0..\infty$). Jelölés: `*`.
- Legalább 1 ($1..\infty$). Jelölés: `+`.

Annak érdekében, hogy egyértelmű legyen, mire vonatkozik a multiplicitás, zárójelek használhatóak, ennek alkalmazását a 4. példa mutatja be.

4. példa. *Opcionálisan szeretnénk megadni, hogy egy Place elembe legfejlebb hány Token helyezhető a következő módon:* `Place p1 bounded 3;`*. A* `bounded` *egy kulcsszó, utána pedig egy egész számot várunk. Vagy mind a két elemet megadjuk a Place definícióban, vagy egyiket sem.*

A nyelvtanban ilyenkor zárójelezéssel kell csoportosítani a két elemet, majd az egész csoportnak megadni a multiplicitását:

```
Place: 'place' name=ID ("bounded" bound=INT)? ';';
```

Lehetőség van egy baloldalhoz többféle jobboldalt illeszteni: ebben az esetben az elemző szabadon választhat az egyes ágak közül. A nyelvtanleírás során ezt a | jellel (pipe) választhatóak el az egyes ágak.

5. példa. *A következő kód azt fejezi ki, hogy egy elem lehet hely, tranzíció, token, be- vagy kimenő él egyaránt.*

```
Element: Place | Transition | Token | InArc | OutArc;
```

A többszörös multiplicitású attribútumok megadásakor gyakran használunk többszörös multiplicitású nem-terminálisokat. Ezt szemlélteti a 6. példa.

6. példa. *A következő kódrészlet definiál egy PetriNet nemterminálist, amely az előbb definiált* `Element` *értékek egy sorozatára hivatkozik. Ezek az elemek mind az* `elements` *nevű, többszörös multiplicitású listába (ld. +=) kerülnek.*

```
PetriNet:
        (elements += Element)*;
```

Gyakran szükséges rekurzív szabályalkalmazást leírni, tipikusan listák, illetve kifejezések leírásánál. Ez lehetséges, a szabályok jobboldalán előfordulhat az a szimbólum, amelyet éppen definiálunk. Ilyenkor mindig fontos, hogy legyen olyan alternatíva, amelyikben nem szerepel a jobboldalon a baloldali nemterminális szimbólum.

Efféle rekurzív szabályokat fel lehet venni az Xtextben, egy megkötéssel: az elemző nem képes ún. *balrekurzív* nyelvtanokat kezelni. Egy nyelvtan balrekurzív, ha létezik olyan szabálya, amelynek valamelyik jobboldala az éppen definiálandó nemterminálissal kezdődik. A balrekurziót a szabályok szétbontásával lehet megszüntetni, amit itt most nem tárgyalunk.

A használható nyelvtanelemek egy részletesebb leírása megtalálható a hivatalos dokumentációban: `http://www.eclipse.org/Xtext/documentation.html#Overview`[4].

3. Modellezési technikák Eclipse környezetben

3.1. Az Eclipse Modeling Framework

Az *Eclipse Modeling Framework* (EMF) célja, hogy lehetővé tegye modellek magas szintű (akár grafikus) megalkotását és menedzselését.

Az EMF által kezelhető modellek nagyban hasonlítanak az UML osztálydiagramokhoz: a központi elemek az *osztályok* (EClass), amelyeknek lehetnek *attribútumai* (EAttribute) és *referenciái* (EReference). Referenciák segítségével más osztályokra lehet hivatkozni, míg attribútumok segítségével egyéb tulajdonságok írhatóak le (tipikusan számok, szövegek vagy felsorolások). Annak érdekében, hogy csoportosítani lehessen a különböző osztályokat, *csomagokba* (EPackage) szervezhetőek.

[4] A dokumentáció egy példánya megtalálható az Eclipse súgóban is, kérdések esetén érdemes megtekinteni.

Mind az attribútumok, mind a referenciák esetén lehetőség van a multiplicitás megadására, amelyek az Xtext nyelvtanokban szereplő multiplicitásfogalomhoz hasonlóan adhatóak meg.

A keretrendszer támogatja, hogy a magas szintű modellből Java forráskódot generáljunk, amely támogatja a modell elemeinek áttekintését, módosítását és sorosítását. Minden egyes EMF osztályhoz egy Java osztály keletkezik[5], amely elérhetővé teszi az összes osztályhoz tartozó EMF attribútumot és referenciát getter/setter metódusokon keresztül.

Egyszeres multiplicitású (pontosan 1 vagy legfeljebb 1) attribútumokhoz és referenciákhoz mind egy `getName`, mind egy `setName` metódust generált, míg többszörös multiplicitás (bármennyi, legalább 1) esetén pedig egy `getName` metódus keletkezik, amely visszaad egy szerkeszthető listát az összes vonatkozó elemről.

Ezen felül a csomagokhoz keletkezik egy `Package` osztály, amely néhány konstans értéket ad vissza, amelyre időnként szükség lehet, valamint egy `Factory` osztályt, aminek a célja az egyes osztályok példányosítása (EMF modellek példányosításához nem szabad Java konstruktorokra hagyatkozni).

7. példa. *A következőkben egy rövid kód segítségével mutatjuk be, hogyan lehet a generált Java kódot felhasználni modellek beolvasására és írására.*

A 2. sorban egy `PetriNet net` osztály által hivatkozott elemek listáját kérjük le a `net.getElements()` hívással (többszörös multiplicitás, kiolvasás). Ezután ezen a listán egy `for`-ciklussal végigiterálunk, és megvizsgáljuk, hogy az aktuális elem Place típusú-e. Amennyiben igen, akkor a 9. sorban a `place.getName()` hívással lekérjük a név attribútumát (egyszeres multiplicitás, kiolvasás).

Ezután a PetriNetFactory segítségével példányosítunk egy Token objektumot, amelynek a 11. sorban beállítjuk a Place referenciáját (egyszeres multiplicitás, szerkesztés). Ezután a kapott elemet felvesszük a elemek listájába a 13. sorban, amely lista szinkronizálódik a Petri háló elemeinek listájával (többszörös multiplicitás, szerkesztés).

```
1  //Getting reference with the multiplicity of many
2  EList<Element> elements = net.getElements();
3  for (Element element : elements){
4    if (element instanceof Place){
5      Place place = (Place) element;
6      //Getting attribute with the multiplicity of one
7      String name = place.getName();
8      //Creating an EClass
9      Token token = PetriNetFactory.eInstance.createToken();
10     //Setting a reference with the multiplicity of one
11     token.setPlace(place);
12     //Setting a reference with the multiplicity of many
13     elements.add(token);
14   }
15 }
```

A kód először lekéri az összes elemet a Petri-hálóból, majd ezeket bejárja egy for-ciklussal. Ha az aktuális elem egy Place, akkor lekéri a nevét, létrehoz egy Tokent és hozzárendeli a Place-hez.

[5]Egészen pontosan egy Java interfész és egy osztály, amelyből az interfész használható fel programozottan.

Az EMF a modellek fájlba írására, illetve beolvasására is ad lehetőséget, de ezt itt nem részletezzük.

3.2. Kódgenerálás EMF modellekből

Gyakori feladat az EMF modellek kezelése kapcsán, hogy a modellt végig kell olvasni, és modell felhasználásával valamiféle szöveges kimenetet kell előállítani. Tulajdonképpen ez a folyamat történik akkor is, amikor a magas szintű EMF modellekből a keretrendszer előállítja a Java forráskódot.

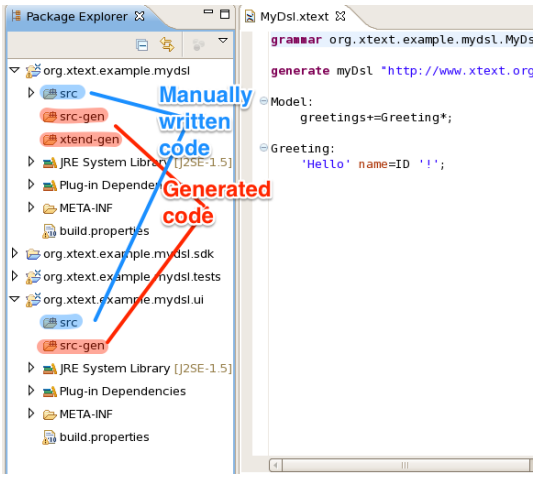
A legegyszerűbb módszer a kódgenerálásra, hogy Java kódból szisztematikusan bejárjuk a modellt, és kézzel összeállítjuk a megfelelő karakterláncokat. Ehhez ki kell indulni a modell gyökeréből, és onnan a megfelelő getter metódusok használatával elérjük a gyermekelemeket, ill. ezek tulajdonságait, amelyek segítségével összeállíthatóak a karakterláncok, amelyek utána akár a `System.out.println()` hívással is kiírhatóak.

Annak érdekében, hogy a modell bejárását kényelmesebb, magasabb szintű kóddal is el lehessen végezni, léteznek különböző sablon alapú kódgenerátorok, mint az Acceleo[6] vagy Xtend[7], de ezekkel ebben a segédletben nem foglalkozunk részletesen.

4. A generált szerkesztő kiegészítése

4.1. Az Xtext alapvető projektjei

Az Xtext biztosít egy varázslót, ami képes legenerálni a fájlokat, amiből a fejlesztés kiindulhat. Az alapértelmezett struktúra a 2. ábrán látható.



2. ábra. Az alapértelmezett projekt struktúra

A varázsló két projektet hoz létre az editor definiálásához, egyet a nyelv elemzőjének és központi szolgáltatásainak (az ábrán `org.xtext.example.mydsl`), a másik pedig az editor grafikus felületének megadásához (az ábrán `org.xtext.example.mydsl.ui`).

[6] `http://eclipse.org/acceleo`
[7] `http://eclipse.org/xtend`

Mindkét projekt tartalmaz egy `src`, egy `src-gen` és egy `xtend-gen` mappát, amelybe Java forrásfájlok kerülhetnek. **Fontos!** Az `src-gen` és `xtend-gen` mappák generált Java-fájlok tárolására szolgálnak, kézzel ne írjunk bele semmit - bármikor törlődhet.

Ezen felül még két segédprojekt is létrejön, amelyek közül az `org.xtext.example.mydsl.test` az elkészült elemző teszteléséhez nyújt segítséget, míg az `org.xtext.example.mydsl.test` projekt az elkészült szerkesztő csomagolásában nyújt segítséget. A mérés során ezekre nem lesz szükség.

Az elemző projekt `src` mappájában létrejön még négy fájl, amelyek közül kettő lényeges a további munkához: létrejön egy `xtext` kiterjesztésű fájl, amelyben meg lehet adni a nyelvtant, és egy `mwe2` fájl, amelynek segítségével le lehet generálni az alapvető szerkesztőt.

Az `xtext` fájl tartalmazza a nyelvtant, az előző fejezetben írtuk le a használható elemeket. A nyelvtanfájl elején még két sorban meg kell adni azonosítókat - ezt az új projekt varázsló létrehozza, ritkán van szükség arra, hogy ehhez kézzel hozzányúljunk.

Az `mwe2` fájl az MWE (Modeling Workflow Engine) számára hoz létre egy végrehajtható munkafolyamatot. A munkafolyamat magában foglalja a különböző modellezési/kódgenerálási lépések elvégzését, ezek közül a legfontosabbak: egy elemző készítése a nyelvtanból, metamodell származtatása a nyelvtanból, metamodellből osztálykönyvtár előállítása, valamint a szerkesztő forráskódjának legenerálása.

A megadott workflow fájlt tipikusan nem kell szerkeszteni, a gyakorlati esetekben egyszerűen csak futtatni kell. A futtatás eredményeképpen további fájlok, valamint egy további projekt jelenik meg a projekthierarchiában. A létrejött `.model` végű projekt tartalmazza az EMF modellt, amit a központi elemek `src-gen` mappájában megjelenő elemző a nyelvtannak megfelelő fájlokból képes előállítani. Végül az `.ui` projekt tartalmazza az elkészült szerkesztőt.

Ebben a fázisban az elkészült szerkesztő alapszinten már használható, egyszerűen elindítható az Eclipse platform futtatás opciója segítségével.

4.2. Validátorok készítése

Gyakori, hogy olyan feltételeket kell megadni, amiket nehéz a nyelvtanban megadni (például egy szám nem vehet fel tetszőleges egész értéket, hanem csak meghatározott értékeket). Ezek megadásához a rendszer támogatja a validátorok készítését.

A kódgenerátor futtatása után megjelenik a központi elemek mellett egy `validation` csomag, amelyben egy `JavaValidator`-ra végződő nevű osztály található. Ebben a projektben lehetőség van további ellenőrzési feltételek elvégzésére.

A következő kódrészlet tartalmazza az alapértelmezetten generálódó tartalmat, amelyen a főbb elemek megfigyelhetőek.

```
@Check
public void checkTypeNameStartsWithCapital(Type type) {
  if (!Character.isUpperCase(type.getName().charAt(0))) {
    warning("Name should start with a capital", MyDslPackage.Literals.TYPE__NAME);
  }
}
```

Az `@Check` annotációval jelezzük a Java számára, hogy a következő metódust meg kell hívni minden olyan elemre, amelynek a típusa megegyezik a paraméterként megadott metamodellbeli elem típusának. A metódus belsejében tetszőleges ellenőrző kód meghívható, és probléma esetén az örökölt `warning()` vagy `error()` metódusokat kell meghívni. Paraméternek át kell adni a hiba szöveges leírását (az itt megadott leírás jelenik meg a felhasználói felületen is), valamint egy azonosítót a hibás elemről, amelyet az EMF modell Package osztályából lehet kinyerni.

4.3. A modell programozott elérése

Szükség esetén a szöveges fájl beolvasható a programozottan is, ebben az esetben ugyanazt az EMF modellt kapjuk, mint amikor feldolgozzuk a szerkesztőn keresztül. A következő kódrészlet bemutatja, hogyan kell ezt körülbelül végezni, ha a `path` változó a beolvasandó fájl elérési útját tartalmazza:

```
ResourceSet set = new ResourceSetImpl();

Resource res = set.getResource(URI
               .createPlatformResourceURI((pathString, true), true);
try {
       System system = (System) res.getContents().get(0);
       //EMF model processing here
       res.save(Collections.emptyMap());
} catch (IOException e) {
       // TODO Auto-generated catch block
       e.printStackTrace();
}
```

Első lépésben előállítunk egy erőforrás objektumot, amely képes beolvasni a forrásfájlt. Ezt az erőforrást az EMF definiálja, és az Xtext megfelelően beállítja a generált elemzőt, hogy itt is megfelelően működjön.

Ezután lehetővé válik a modell gyökérelemének kinyerése a `try`-blokkon belül, majd az EMF modell tetszőlegesen feldolgozható, felhasználva a getter/setter metódusokat és a `Package` osztályt példányosításhoz.

Ha a feldolgozás megtörtént, a változtatásokat az erőforrás objektum `save()` metódusával elmenthetjük - ez a változás azonnal szinkronizálódik a megnyitott editorba (ha van ilyen). A `save()` metódus paraméterével a mentés módját lehet finomhangolni - a mérés során erre nem lesz szükség.

4.4. Kódgenerátor hívása

A validátorok kezeléséhez hasonlóan az Xtext környezet szolgáltatásként teszi lehetővé egy kódgenerátor meghívását futás közben, mely Java vagy Xtend nyelven írható meg. Egy alapértelmezett, Xtend alapú implementáció váza legenerálódik az elemző projektjébe.

A generátor egyszerűen használható tetszőleges szöveges adatok legenerálására, különös tekintettel a template kifejezések támogatására. A felhasználható szintaxisról részletesebb dokumentáció elérhető a projekt weboldalán: `http://www.eclipse.org/xtend/documentation.html#templates`.

5. OpenVZ konténerek kezelése

Az OpenVZ[8] egy operációs rendszer szintű virtualizációs megoldás, mely lehetővé teszi, hogy izolált környezetek jöjjenek létre egy közös hoszt operációs rendszer kernel felett. Az ilyen izolált környezetet *konténernek* nevezzük, mely a benne futó folyamatok számára olyan állapotot mutat, mintha a konténeren kívül más folyamatok nem léteznének a gépen.

A konténer felfogható egy nagyon pehelysúlyú virtuális gépként. Ennek megfelelően egy konténer külön virtuális hálózati interfésszel rendelkezik, külön IP címet kap, külön hallgatózhatnak a hálózati portokon szerverfolyamatok. Minden konténer önálló fájlrendszerrel rendelkezik, így a rendszert alkotó összes alkalmazás, programkönyvtár, scriptek és konfigurációs fájlok konténerenként függetlenek lehetnek. Egy konténeren belül az összesített erőforrásfoglalásra (fájlrendszer, memória, CPU idő) részletesen finomhangolható, szigorú korlátok adhatóak meg.

Ennek a megoldásnak a platform virtualizációval [9] szembeni előnye, hogy a közös kernel és általában az operációs rendszer szintű erőforrások virtualizálása miatt nagyon erőforrástakarékos. Hátránya viszont, hogy a közös kernel miatt csak egyféle operációs rendszer - pl. az OpenVZ esetén Linux - példányai lehetnek a konténerekben, viszont a disztribúció minden egyéb része eltérő lehet.

A mérés során a virtuális infrastruktúra modelljéből OpenVZ környezetben hozzuk létre a gépeket a megfelelő szoftverekkel a környezet alacsony erőforrásigénye miatt. A felhasznált módszer ugyanakkor teljesen hasonló módon alkalmazható más technológiák felett, mint a hasonló lxc[10] eszköz, vagy akár a teljesen eltérő elvek alapján működő platform virtualizációs technikák, netán cloud megoldások felett is, feltéve, hogy lehetőségünk van új virtuális gépeket programozottan létrehozni.

5.1. Konténerek kezelése parancssorból

OpenVZ alatt a konténerek elnevezése *Virtual Private Server (VPS)*. Egy VPS életciklusa következő:

Create: egy előretelepített Linux disztribúciót tartalmazó fájlrendszer-sablonból készül egy új másolat melyhez egy VPS azonosítót (`veid`), valamint néhány egyéb beállítást rendelünk hozzá. Létrejön a gyökér fájlrendszer a konténert leíró konfigurációs állomány.

Start: egy már létező VPS-t elindítunk, mely azzal jár, hogy a VPS fájlrendszerén belüli init scriptek lefutnak, a hálózati interfésze létrejön, végül a VPS parancssori interfészére (konzol) be lehet lépni.

Stop: a VPS-ben futó folyamatokat leállítja.

Destroy: törli a teljes a VPS-t a fájlrendszerből és a konfigurációs fájlokból.

Konténerek kezeléséhez elsődlegesen a `vzctl` parancssori program áll rendelkezésre. Minden műveletnél fontos megadni a megfelelő VPS azonosítóját a *veid*-t, amely szokásosan egy egész szám, 101-től kezdve.

[8] Ld. `http://wiki.openvz.org/Main_Page`
[9] Pl. VMware, XEN, KVM technológiák
[10] Ld. `http://lxc.sourceforge.net/`

- Egy VPS létrehozása:

  `vzctl create` *<veid>* `--ostemplate` *<sablon neve>* `--config` *<konfigurációs beállítások neve>* `--ipadd` *<ip cím>* `--hostname` *<hosztnév>*

  Az `ostemplate` paraméterrel kell megadni a példányosítani kívánt fájlrendszer sablon nevét. A `config` paraméter egy előre elkészített erőforráskorlát konfigurációs csomag (*plan*[11]) megadását teszi lehetővé. Ezek a csomagok határozzák meg a maximális memória- és diszk használatot. A hosztnév és IP cím beállítás előre konfigurálja a létrejövő VPS-t.

- Egy már létező vagy akár működő VPS-en számos beállítást is végezhetünk. Például:

  - Névszerver beállítása:
    `vzctl set` *<veid>* `--nameserver` *<DNS szerver>* `--save`
  - Új IP cím felvétele:
    `vzctl set` *<veid>* `--ipadd` *<IP cím>* `--save`
  - IP cím törlése:
    `vzctl set` *<veid>* `--ipdel` *<IP cím>* `--save`
  - Hosztnév módosítása:
    `vzctl set` *<veid>* `--hostname` *<hosztnév>* `--save`

  A `--save` fontos, ez jelzi, hogy a konfigurációváltozás perzisztensen kerüljön elmentésre. Minden indításkor az OpenVZ által eltárolt konfigurációs beállítások kerülnek alkalmazásra, a legtöbb esetben felülírva a VPS fájlrendszerén belüli módosításokat is, ezért fontos, hogy ahol lehetséges a „külső” beállításokat használjuk.

- VPS indítása:

  `vzctl start` *<veid>*

- Közvetlen belépés root felhasználóval a VPS parancssori felületére, akár hálózat nélkül is:

  `vzctl enter` *<veid>*

  A konténerből való kilépés az `exit` paranccsal, vagy a Ctrl+D billentyűkobinációval lehetséges.

- Parancs futtatása a VPS-en belül:

  `vzctl exec` *<veid>* *<parancs>* *[<paraméterek...>]*

- VPS leállítása:

  `vzctl stop` *<veid>*

- VPS teljes törlése:

  `vzctl destroy` *<veid>*

[11] A plan kifejezés itt előfizetési díjcsomagra utal, mivel VPS-t gyakran nyújtanak hosting szolgáltatók és az előfizetési díjcsomagot a beállított erőforráskorlátoknak megfelelően határozzák meg.

8. példa. *Egy VPS létrehozásához, majd ezen belül alkalmazások telepítéséhez a következő lépéseket kell elvégezni:*

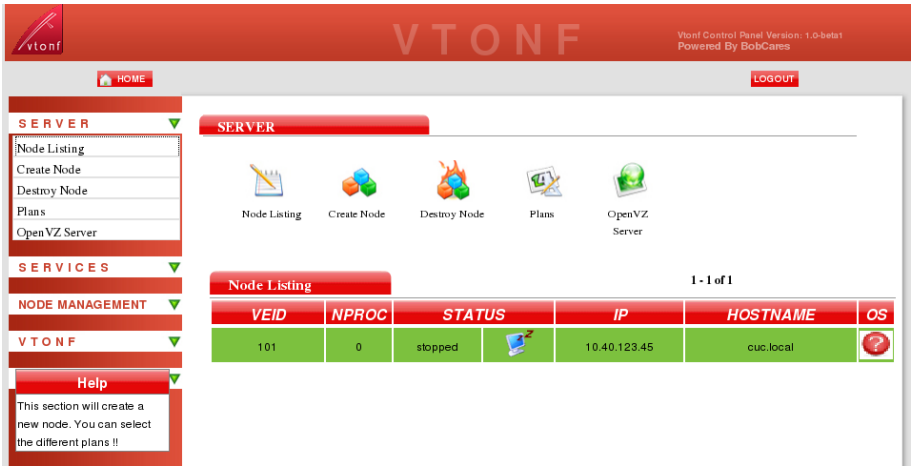
1. *VPS létrehozása, IP cím és hosztnév megadása.*
2. *Névszerver beállítása - erre azért van szükség, mert az alkalmazást telepítés előtt hálózatról kell letölteni.*
3. *VPS indítása*
4. *Kívülről a vendég operációs rendszer csomagkezelőjének meghívása az alkalmazás telepítésére*
5. *Opcionálisan a VPS-be belépés telepített alkalmazás kipróbálása*

Minden műveletet root felhasználó nevében kell végezni, szükség esetén sudo használatával.

```
1 sudo vzctl create 102 --ostemplate ubuntu-9.04-x86 \
2         --config vtonf.128MB --ipadd 10.40.123.56 --hostname demo1
3 sudo vzctl set 102 --nameserver 10.40.1.1 --save
4 sudo vzctl start 102
5 sudo vzctl exec 102 apt-get update
6 sudo vzctl exec 102 apt-get install mc
7 sudo vzctl enter 102
```

5.2. Konténerek kezelése webes felületről

A méréshez kiadott virtuális gép tartalmaz egy külön telepített webes felületet is az OpenVZ VPS-ek kezelésére, mely alapértelmezetten a `http://localhost:8001/index.php` címen érhető el. Ennek most elsődleges célja a hibakeresés, de használható a rendelkezésre álló operációs rendszer sablonok listázására és az erőforrás konfigurációs plan beállítására is. A 3. ábra egy állapotát mutatja ennek az adminisztrációs felületnek.



3. ábra. A VTONF webes kezelőfelület OpenVZ-hez

6. A mérési feladat

6.1. A felhasználandó konfigurációs modell

A mérés célja egy konfigurációs nyelv és hozzá kapcsolódó szöveges editor létrehozáss, amely képes leírni OpenVZ virtuális gépek egyszerűsített konfigurációját és definiálni rájuk telepítendő szoftvereket.

A virtuális gépek egyszerűsített paraméterei a következők:

- **Azonosító**, mely egy egész érték, jellemzően 100-tól kezdődően,
- **név**, mely a virtuális gép elnevezését adja,
- **IP cím**,
- **névszerverek** címei,
- telepített **szoftverek** azonosítója.

Egy telepítendő szoftver megadásához a következő adatokat kell leírni:

- A **szoftver neve**, egy azonosító, melynek segítségével hivatkozhatunk rá,
- **verzió**ja,
- a telepítéshez használt **csomag neve**.

9. példa. *A következő kód egy javasolt szintaxist mutat be két gép és két szoftver leírására: először definiálja a* `joe` *és* `midnightcommander` *szoftvereket, majd két virtuális gépet definiál,* `Machine1`*, ill.* `Machine2` *néven, beállítva nekik az azonosítót, egyikre a joe-t, másikra az mc-t telepítve, és beállítva a hálózati paramétereket.*

```
[SOFTWARE]
joe version "1.5" package "joe"
midnightcommander version "4.62" package "mc"

[Machine1]
id = 101
software = joe
address = 192.168.125.123
nameserver = 192.168.125.2

[Machine2]
id = 102
software = midnightcommander
address = 192.168.125.124
nameserver = 192.168.125.2
```

6.2. Mérési feladatok összegzése

A mérés során a bemutatott technológiák és konfigurációs modell felhasználásával a következő feladatokat kell megvalósítani (az elvégzendő feladatok részletes listája elérhető lesz a mérési jegyzőkönyvsablonban):

1. Szöveges editor készítése a konfigurációs állományok létrehozására Xtext technológia segítségével.

2. A szöveges editor kiegészítése úgy, hogy lehessen az azonosítókat automatikusan generálni a forráskódba (azaz ne kelljen a felhasználónak gondoskodni az egyedi értékekről).

3. Kódgenerálás megvalósítása, mely az elkészített konfiguráció alapján a előállítja a szükséges parancsok, amelyek előállítják OpenVZ platformon a beállított konfigurációt (virtuális gépek létrehozása, szoftverek telepítése).

4. Opcionális feladat: a parancsok közvetlen futtatása helyett egy megfelelő shell script előállítása kódgenerátor technikákkal.

5. Opcionális feladat: validátor készítése az editorhoz Java kód használatával, ami segít pontosítani az elfogadott nyelvtant (ötlet: IP cím nem tetszőleges számokból állhat).

7. Ellenőrző kérdések

A mérésre készülés során gondolja át a következő kérdésekre adandó választ. A felkészülés a mérés elején beugró írása során kerül ellenőrzésre, amely tartalmazhat az alábbiaktól eltérő kérdéseket is.

1. Hasonlítsa össze a terminális és nemterminális szimbólumok szerepét nyelvtanokban!

2. Hogy néz ki egy szabály generatív nyelvtanokban?

3. Miket generál az Xtext a nyelvtanfájl alapján?

4. Milyen metódusok generálódnak egy EMF osztály többszörös multiplicitású attribútumaihoz?

5. Milyen elemekből áll egy VPS életciklusa?

6. Mivel azonosítja az OpenVZ a definiált VPS-eket?

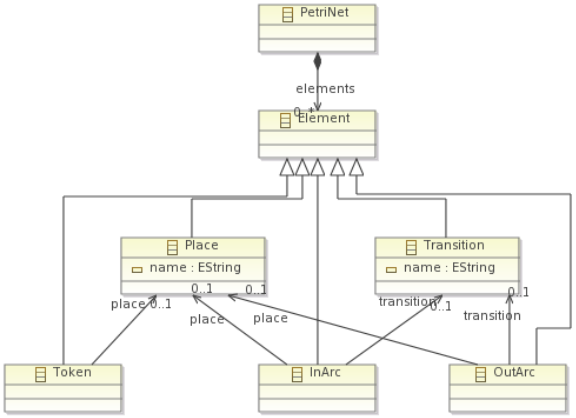
A. Példa: Nyelvtan Petri-hálók leírására

Példaként bemutatjuk, hogyan lehet az Xtext segítségével definiálni egy szöveges leírást a Petri-hálókhoz. Egy Petri-háló helyekből és tranzíciókból áll, amelyeket élek kötnek össze. Az élek irányítottak, és vagy helyekből tranzíciókba, vagy tranzíciókból élekbe haladhatnak.

Egy egyszerű nyelvtan a Petri-hálókhoz:

```
1  grammar hu.bme.mit.petrinet.Language with org.eclipse.xtext.common.
       Terminals
2
3  generate language "http://www.bme.hu/mit/petrinet/Language"
4
5  PetriNet:
6          (elements += Element)*;
7
8  Element:
9          Place | Transition | Token | InArc | OutArc;
10
11 Place:
12         'place' name=ID ';'
13 ;
14
15 Transition:
16         'transition' name=ID ';'
17 ;
18
19 Token:
20         'token in' place=[Place] ';'
21 ;
22
23 InArc:
24         'inarc from' transition=[Transition] 'to' place=[Place] ';'
25 ;
26
27 OutArc:
28         'outarc from' place=[Place] 'to' transition=[Transition] ';'
29 ;
```

A nyelvtanhoz az Xtext a következő metamodellt generálja:



Egy példa háló szöveges leírása, amely megfelel a nyelvtannak:

```
1  place p1;
2  place p2;
3  transition t1;
4  transition t2;
5  token in p1;
6  inarc from t1 to p1;
7  outarc from p1 to t2;
8  inarc from t2 to p2;
9  outarc from p2 to t1;
10 inarc from t1 to p1;
```