Modelltranszformációk statikus analízise

Ujhelyi Zoltán

Budapesti Műszaki és Gazdaságtudományi Egyetem, Méréstechnika és Információs Rendszerek Tanszék

Manapság a szoftverfejlesztés területén egyre hangsúlyosabb az OMG Modellvezérelt Architektúra (MDA)[1] kezdeményezésének használata, amelynek központi eleme, hogy a magas szintű modellekből kiindulva transzformációs lépések sorozatával lehessen eljutni a futtatható alkalmazásig.

A módszer sikerességéhez szükséges ezen modelltranszformációk[2] fejlesztésének hatékony támogatása. Ennek egyik lehetséges megvalósítása az integrált fejlesztőrendszerekben is alkalmazott statikus forráskód elemzők használata. Ezek a program futtatása nélkül képesek bizonyos tipikus hibák kimutatására, a program struktúrája, valamint a nyelv szemantikája és szintaktikája alapján.

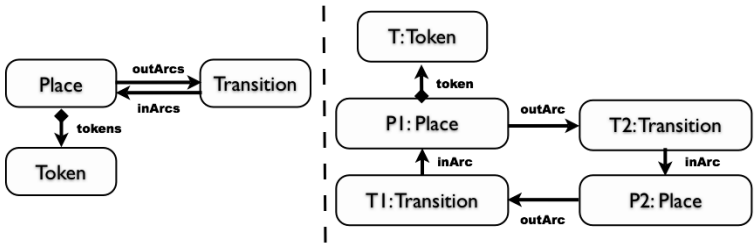
Ebben a cikkben egy modelltranszformációs programok ellenőrzésére szolgáló statikus forráskód ellenőrző eljárást mutatok be, amely arra épít, hogy kényszerkielégítési problémára vezet vissza ellenőrzési feladatokat.

A módszer elvileg könnyen alkalmazható tetszőleges transzformációs keretrendszer vizsgálatára, az én eljárásom a Budapesti Műszaki és Gazdaságtudományi Egyetem Méréstechnika és Információs Rendszerek Tanszékén fejlesztett VIATRA2[3] modelltranszformációs keretrendszerhez illeszkedik.

A cikkben először röviden ismertetem a metamodellezés, a modelltranszformációk, a statikus forráskód analízis és a kényszerkielégítési problémák kezelésének alapjait, majd a saját eljárásom fontosabb gondolatait mutatom be.

Metamodellezés

Annak érdekében, hogy képesek legyünk a modelltranszformációkat specifikálni, szükséges a forrás- és célmodellek osztályának a definíciója. A metamodellek célja, hogy leírja a lehetséges modellelemek típusát, illetve az egyes típusok között felírható kapcsolatokat.



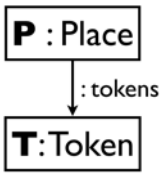
1. ábra: Petri-háló metamodell és modell

Az 1. ábra bal oldali részén a rendszermodellezés terén gyakran használt Petri-hálók metamodellje látható. A metamodellből leolvasható, hogy a modell elemei lehetnek helyek (*Place*), tranzíciók (*Transition*) és tokenek (*Token*). A tokenek helyekhez kötődnek (*tokens*), a helyek és tranzíciók között pedig irányított élek vezethetnek (*inArcs*, *outArcs*). Az ábra jobb oldali része pedig egy ún. példány modellt tartalmaz, azaz egy olyan modellt, amely megfelel a metamodellben leírt feltételeknek.

Transzformációs program

A VIATRA2 rendszerben a transzformációk leírásához háromféle szerkezet használható fel: (1) gráfmintákat (Graph Pattern) lehet használni a modellen a feltételek leírására, (2) gráftranszformációs szabályok (GT Rule) segítségével elemi modellmanipulációkat, (3) absztrakt állapot gép szabályok (ASM Rule) használatával pedig vezérlési szerkezeteket lehet kifejezni.

A gráfminták segítségével olyan feltételeket vagy kényszereket definiálhatunk, amelyeket a modelltér egy részének teljesítenie kell, hogy magát a transzformációs lépést el lehessen végezni. A gráfminták alapvetően modellelemeket és köztük fennálló relációkat tartalmaznak (ezen elemek és relációk származhatnak mind a forrás, mind a cél metamodellből), illetve a minták kényelmesebb leírása érdekében lehetséges más minták meghívása (akár rekurzív módon is).



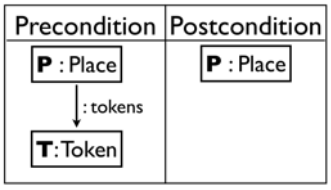

```
pattern placeWithToken(P, T) =
{
  'PetriNet'.'Place'(P);
  'PetriNet'.'Place'.'Token'(T) in P;
  'PetriNet'.'Place'.'tokens'(X, P, T);
}
```

2. ábra: Egyszerű gráfminta

A 2. ábrán egy egyszerű **gráfminta** kódja és szokásos grafikus ábrázolása látható: a minta akkor illeszkedik, ha a *P* nevű hely tartalmazza a *T* tokent. Az első feltétel azt fejezi ki, hogy a *P* paraméter egy *PetriNet.Place* típusú, a második szerint *T* egy *PetriNet.Place.Token* típusú entitás, végül a harmadik feltétel azt tartalmazza, hogy a a két entitás között létezik egy *PetriNet.Place.tokens* típusú reláció.

A gráftranszformációs szabályokat a VIATRA2 rendszerben egy előfeltétel (vagy bal oldali) és egy utófeltétel (vagy jobb oldali) gráfminta segítségével lehet megadni: az előfeltétel a szabály alkalmazásának feltételeit adja meg, míg az utófeltétel megmutatja, hogyan kell kinéznie a mintának a szabály alkalmazása után: azon modellelemek vagy relációk, amelyek csak az előfeltételben szerepelnek, törlendőek, míg azok, melyek csak az utófeltételben, létrehozandóak.

A 3. ábrán egy olyan **gráftranszformációs** szabály látható, amely a paraméterként kapott helyről elvesz egy tokent. Ezt úgy fogalmazza meg, hogy előfeltételnek meghívja a korábban bemutatott *placeWithToken* mintát, majd a talált mintára alkalmazza az utófeltételben szereplő, az előfeltételhez hasonló módon megfogalmazott *placeWithoutToken* mintát.




```
gtrule removeToken(in Place) =
{
  precondition
    find placeWithToken(Place, Token)
  postcondition
    find placeWithoutToken(Place,Token)
}
```

3. ábra: Egyszerű gráftraszformációs szabály

Az ASM szabályok segítségével határozhatóak meg, hogy az egyes gráftranszformációs szabályokat milyen sorrendben kell alkalmazni a komplex transzformáció elvégzéséhez. Ezen szabályok alapvetően procedurális végrehajtási szerkezet leírására használhatóak.

```
forall Place with find sourcePlace(T,Place) do
choose with apply removeToken(Place);
```

4. ábra: Egy bonyolultabb ASM szabály

A 4. ábra egy bonyolultabb folyamatot mutat be: az első sorban felírt *forall* szabály megkeresi az összes *sourcePlace* mintának megfelelő <T, Place> párt, majd a következő, *choose* szabály eltávolít ezekről a *removeToken* gráf transzformációs szabály meghívásával egy tokent. Amennyiben nincs ilyen eltávolítható token, az illesztés meghiúsul.

A nyelvi elemek részletesebb leírásához lásd [3].

Statikus analízis

Ismert tény a szoftverfejlesztés területén, hogy ha egy hiba felderítetlen marad egy munkafázisban, a javításának költsége egy nagyságrenddel növekszik. Emiatt fontos, hogy a fejlesztés során a hibák felderítése minél gyorsabban és minél inkább automatizált módon történhessen.

Erre a célra több módszer is használható. Egy a gyakorlatban sokszor alkalmazott megoldás a statikus analízis, mely a program futtatása nélkül képes bizonyos típusú hibák felderítésére.

Az ellenőrzés alapvető gondolata, hogy a program futását egy ún. absztrakt tartományon[4] követjük nyomon. Ez azt jelenti, hogy a program változóit kisebb méretben reprezentáljuk, és megvizsgáljuk, hogy az elvégzett műveletek mit eredményeznek ezen a tartományon.

Például egy effajta absztrakt értelmezési tartomány lehet, hogy az egész számok halmazát leképezzük az előjelek halmazára: {(0), (+), (−), (±)}. Ekkor a $-1518 \cdot 317$ szorzás a (−)·(+) szorzásra képződik le, és a művelet tulajdonságai alapján tudjuk, hogy a végeredmény is negatív előjelű kell, hogy legyen. Fontos ugyanakkor észrevenni, hogy a $-1517+317$ összeadás a (−)+(+) összeadásra képződik le, viszont ekkor a végeredmény előjelét nem tudjuk meghatározni.

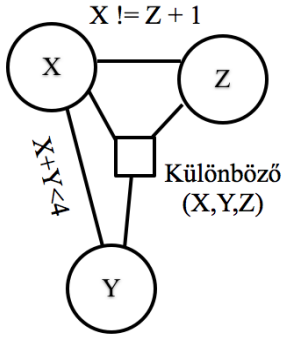
Az efféle pontatlanságok ellenére ez az ellenőrzés gyakorlati problémák felismerésére jól használható: a futás nyomon követése az absztrakt tartományon is megtörténhet, és ha itt a feltételek nem elégíthetőek ki, a probléma a konkrét

tartományon is jelen van.

Absztrakt értelmezési tartománynak tekinthető az is, ha a változók értékét a változók típusával helyettesítjük: ezen a tartományon azt tudjuk megvizsgálni, hogy az egyes műveletek milyen típusú változókon értelmezettek, és ezek alapján az egyes változóknak melyek a lehetséges típusai.

Kényszerkielégítési probléma

A kényszerkielégítési probléma (Constraint Satisfaction Problem, CSP)[5] a mesterséges intelligencia területéről származó eljárás. A feladat röviden: adott változókhoz, értékkészleteikhez és a változókon értelmezett kényszerfeltételekhez keresünk a kényszerfeltételeket kielégítő változóhozzárendelést.



	X	Y	Z
Kiindulás	{1,2,3,4,5}	{1,2,3,4,5}	{1,2,3,4}
X+Y<4	{1,2}	{1,2}	{1,2,3,4}
Különböző	{1,2}	{1,2}	{3,4}
Választás	{1,2}	{1}	{3,4}
Különböző	{2}	{1}	{3,4}
Választás	{2}	{1}	{3}

5. ábra: Egy egyszerű kényszerkielégítési probléma

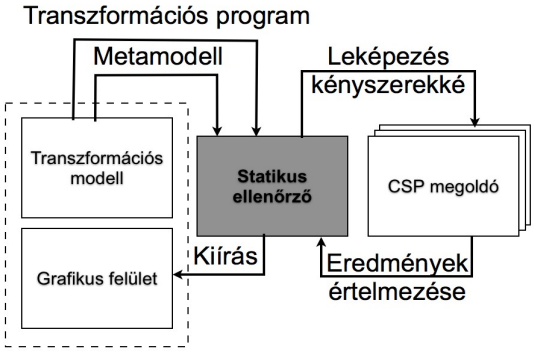
Az 5. ábrán egy egyszerű kényszerkielégítési probléma gráfmegjelenítése látható: a probléma három változóból (*X*, *Y*, *Z*) és három kényszerből (X+Y<4, X!=Z+1 és mindhárom változó értéke különböző) áll. A változók értékkészleteit az ábra jobb oldalán látható táblázat első sora tartalmazza.

A kényszerkielégítési problémák megoldásának gyakori módszere, hogy a változók értékkészleteit a kényszerek figyelembe vételével szűkítjük az összes lehetséges megoldás megőrzésével. Ha a szűkítés során egy változó értékkészlete üres halmaz lesz, akkor a probléma nem kielégíthető. Például az X+Y<4 kényszer miatt nem lehetséges, hogy X vagy Y lehetséges tartományából bármilyen értéket felvehessen; ezt a módosítást a táblázat második sora tartalmazza.

Szűkítéssel nem mindig lehet végeredményhez jutni – ilyenkor egy visszalépéses keresést szokás alkalmazni: egy kiválasztott változó értékkészletét csökkentve próbálunk megoldást keresni (*Választás* sorok a táblázatban); sikertelenség esetén visszalépünk, és így az összes lehetőséget megvizsgáljuk.

A megvalósított rendszer

A 6. ábrán a megvalósított rendszer vázlatos működése látható. A statikus ellenőrző program a modelltranszformációs keretrendszerből (esetünkben a VIATRA2 rendszerből) beolvassa a forrás- illetve a cél metamodelleket, valamint a transzformációs programot, majd ennek az ellenőrzési problémáját leképezi egy



6. ábra: Az ellenőrző rendszer működése

kényszerkielégítési problémává. Ezt a problémát egy kényszermegoldó eszköz segítségével kiértékeli, értelmezi a futás eredményét, és a következtetéseket megjeleníti a keretrendszer grafikus felületén.

Az ellenőrző a beolvasott transzformációs programból egy gráfmodellt épít, amelynek csomópontjai a program elemeit jelentik, míg az élek segítségével az elemek közti kapcsolatok írhatóak le (ilyen például egy gráfminta és a mintát alkotó feltételek között fennálló tartalmazás kapcsolat).

A modell használatával a program egy absztrakt tartományon ellenőrizhető, ugyanis az ellenőrző az építés során bizonyos, az ellenőrzés szempontjából irreleváns részleteket elhagy. Ilyen például a rekurzív hívások kezelése: ezeket csak egy előre rögzített mélységig követi (a nem rekurzív hívásokat ugyanakkor végig követi).

A modell másik fő feladata, hogy lehetővé tegye bejárásokhoz (az ún. „visitor” tervezési mintát használva). Erre azért van szükség, hogy ugyanazon a modellen többféle ellenőrzést is el lehessen végezni: az egyes ellenőrzésekhez szükséges lehet, hogy a modellt más sorrendben lehessen bejárni, illetve más kényszereket lehessen előállítani a menet közben.

Az ellenőrző a bejárás során csomópontonként állítja elő és tölti be a kényszereket a megoldóba, folyamatosan vizsgálva a probléma kielégíthetőségét. Ha a probléma kielégíthető, akkor a bejárást folytatni kell, ellenkező esetben az ellenőrző megkísérli felderíteni, hogy pontosan milyen feltétel sérült, és ezt a transzformáció fejlesztője számára érthető módon meg is jeleníti a modelltranszformációs keretrendszer grafikus felületén.

A bejárás egy fontos feladata, hogy kezelje azt is, hogy a transzformációs program különböző lefutásai a kód más-más részeit érintik: a teljesség érdekében a bejárást többször meg kell ismételni az egyes elágazási pontoknál más-más irányt választva.

Transzformációs programok típushelyesség-ellenőrzése

A statikus ellenőrző eljárás tesztelésére a transzformációs programok típushelyességének vizsgálatát valósítottam meg a fejlesztés során. Ehhez a

típushelyesség problémáját visszavezettem egy kényszerkielégítési problémára.

Az így definiált probléma változói a transzformációs program változóinak és konstansainak a típusát reprezentálják. A transzformációs programban használt változóknak nincsen egy rögzített, statikus típusa, ezért értékadásoknál előfordulhat, hogy egy változó típusa megváltozik – annak érdekében, hogy ezt kezelni lehessen, az értékek változásakor a kényszerkielégítési probléma számára egy újabb változót kell létrehozni.

A kényszerváltozók értékkészlete a transzformációs program változóinak lehetséges típusait tartalmazza, azaz a forrás- és cél metamodellek elemeit, valamint néhány beépített típust (karakterlánc, egész és lebegőpontos szám, stb.)

A kényszerkielégítési problémában a többszörös öröklődést tartalmazó metamodelleket is kezelni kell, ezért a típushierarchiát megfelelően előállított számhalmazokkal reprezentáltam. A módszer legfőbb tulajdonsága, hogy ha egy modellelem leszármazottja egy másiknak, akkor a hozzájuk tartozó halmazok közül az ősosztályé része a leszármazottának. A módszer részletesebben lásd [6].

A kényszerkielégítési probléma kényszereit a transzformációs program modelljének bejárása közben kell előállítani: az aktuális nyelvi elem szemantikája alapján határozhatóak meg a típusinformációk. Például a szorzás művelet esetén tudjuk, hogy mindkét paraméter és a végeredmény egész vagy lebegőpontos szám típusú, valamint az automatikus típuskonverzió miatt ha legalább az egyik paraméter lebegőpontos szám, akkor a végeredmény is lebegőpontos szám lesz.

Hasonló logikát alkalmazva az összes nyelvi elemből fel lehet építeni típuskényszereket.

Általánosan jellemző a transzformációs nyelvre, hogy a gráfminták esetén a típus információk teljes egészében rendelkezésre állnak, míg az ASM szabályok esetén ezeket jelentős részben következtetni kell. Az ellenőrzés az elvégzett tesztek alapján az esetek döntő többségében képes meghatározni ezeket az ismeretlen típusokat.

Felhasznált irodalom

[1] Object Management Group- Model Driven Architecture – A Technical Perspective, September 2001. http://www.omg.org

[2] Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformations: Foundations. World Scientific (1997)

[3] Varró, D. and Balogh, A. The Model Transformation Language if the VIATRA2 Framework, Sci. Comput. Program., 68(3) pp. 214-234. (2007)

[4] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints (Los Angeles, California, 1977), ACM Press, pp. 238–252.

[5] Apt, K. Principles of Constraint Programming. Cambridge University Press, 2003.

[6] Caseau, Y.: Efficient handling of multiple inheritance hierarchies. In OOPSLA ’93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (New York, NY, USA, 1993), ACM, pp. 271–287.