Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Palatinszky Dániel

TÖBBRÉTEGŰ METAMODELLEZÉS GRAALVM ALAPON

KONZULENS

Dr. Mezei Gergely

BUDAPEST, 2019

Tartalomjegyzék

Összefoglaló

Napjainkban a modell-alapú fejlesztés és az ennek egyre gyakrabban alapjául szolgáló metamodellezés egyre nagyobb szerepet kap mind az informatikai kutatások, mind az ipari fejlesztések területén. Az elmúlt évtizedekben a jelenleg elterjedt kétszintű metamodellezési technikákat sikerrel alkalmazták változatos ipari környezetben, azonban a modern ipari környezet kihívásaira már nem minden esetben nyújt kellően rugalmas alapot. A többszintű metamodellezés segítségével a kívánt rugalmasság elérhető lenne, de a témakör komplexitása és újszerűsége miatt a mai napig nem született egységes, az ipar által elfogadott szabványos megoldás a metamodellezés ezen ágazatára.

A többrétegű metamodellezés a többszintű metamodellezés egyik ága, ahol nagy szerepet kap az önleíró, több lépésben finomítható, végig validált specifikáció. A DMLA (Dynamic Multi-Layer Algebra) egy többrétegű metamodellezési keretrendszer, mely arra törekszik, hogy egy széles körben használható, elfogadott megoldást nyújtson. A DMLA jelenleg alapkutatás fázisban tart, az ipari felhasználásig még több nehézséget is meg kell oldani.

A DMLA saját programozási nyelve, a DMLAScript lehetővé teszi a felhasználók számára, hogy egyszerű és átlátható módon készítsék el a szakterületi modelleket, illetve saját validációs logikával egészítsék ki azokat. Eredetileg a DMLAScript egy fordítás-alapú megközelítés segítségével futtatta az elkészült szakterületi modelleket, de ennek a megoldásnak mind a teljesítmény, mind az interaktivitás terén megmutatkoztak a korlátai.

A dolgozat egy teljesen új, interpretált végrehajtáson alapuló megoldást mutat be, mely egy új generációs Java virtuális gépet és fordítót, a GraalVM technológiát használja fel. A GraalVM mindezek mellett biztosít egy nyelvfüggetlen API-t is, a Truffle-t, mely tetszőleges nyelv implementálását teszi lehetővé az új virtuális gép fölé. Ezen két technológia által nyújtott eszközkészlet segítségével elkészítettük a DMLAScript GraalVM alapú implementációját, amely lehetővé tette a modellek futás idejű manipulációját és validációját, a korábbi teljesítmény megtartása vagy bizonyos esetekben javítása mellett.

Abstract

Nowadays, metamodeling-based development is gaining importance in both IT and industrial development. In recent decades, the two-tier metamodeling techniques, which are currently prevalent, have been successfully applied in a diverse industrial environment. However, these techniques do not always provide a sufficiently flexible basis for the challenges of a modern industrial environment. Based on multi-level metamodeling, the required flexibility could be achieved, but due to the complexity and modernity of the topic, a standard solution does still not exists.

Multi-layer metamodeling is a branch of multi-level metamodeling, where the emphasis is on self-description and granularly refined, validated specification. DMLA (Dynamic Multi-Layer Algebra) is a multi-layer metamodeling framework. The primary goal of DMLA is to provide a widely accepted solution in the field, however, it is still under research and we have to solve several problems before applying it in industrial environment.

DMLA has its own programming language referred to as DMLAScript. DMLAScript allows to develop domain languages in a simple and clear way. On top of that, it provides the tools needed for custom validation development. Originally, DMLAScript was processed by a compiler, but this solutions had its limitations in terms of both performance and interactivity.

This paper provides a new, interpreter based approach which uses a new generation Java virtual machine and compiler referred to as GraalVM. Furthermore, GraalVM provides a language agnostic API referred to as Truffle, which allows the users to implement their custom language for the virtual machine. With the help of the combined toolset of these two technologies, we successfully implemented the GraalVM-based DMLAScript. The new approach allows runtime manipulation and validation of the domain models while it maintains or improves upon the performance of the old approach.

1 Bevezetés

A modellezés és a modellvezérelt fejlesztés népszerűsége az iparban fokozatosan növekszik az utóbbi években. A népszerűség kulcsa főként a magas absztrakciós adta rugalmasság és a szakterületi nyelvekből elérhető automatizált kódgenerálás. Az UML-alapú modellezés mellett egyre inkább tért hódítanak a metamodellezésen alapuló technológiák. A közelmúltig a metamodellezés legnépszerűbb képviselői a Meta Object Facility (MOF) által nyújtott lehetőségekre támaszkodtak. A MOF [1] négy modellezési réteget definiál. Az első és egyben a legfelső réteg a meta-meta-model vagy röviden M3. Ez a réteg definiálja az alatta levő rétegben, azaz a metamodellezési rétegben (M2) használható nyelvet. Az M2 réteg modelljei írják le az M1-ben található elemeket, melyre építve végül az M0, azaz az adatréteg már konkrét valós objektumokat definiál. Vegyünk egy közismert példát, az UML-t, melyen keresztül jobban szemléltethető a MOF. Az UML esetében az M2 írja le az UML metamodelljét, azaz azt a nyelvet, amelyet az UML modellek használnak. Ebből következik, hogy az alatta levő réteg, az M1 tartalmazza a konkrét UML modelleket. Végül az M0 rétegben találhatóak az UML modellek példányai.

Habár a klasszikus két, ill. négyszintű metamodellezés rendkívül jól alkalmazható szakterületek definiálására, a gyakorlati alkalmazás során fény derült a hiányosságaira is [2]. A probléma a MOF alapú metamodellezési módszerekkel többrétű. Egyrészt a metamodellezési szintek számának és céljaiknak megkötése nehézkessé teszik a szakterületek nyelvében szereplő fogalmak aprólékos, rétegről rétegre történő megfogalmazását, így csökkentve az eszközök által nyújtható lehetőségek halmazát. Másrészt a hagyományos objektumorientált példányosítás (*„shallow instantiation”*) értelmében az osztályok csak a közvetlen leszármazottaik szemantikáját képesek meghatározni, azonban a további példányosítási láncra már nincsenek hatással.

A metamodellezés hiányosságait áthidalandó tehát szükség van egy olyan modellezési megközelítésre, melyben tetszőleges számú szintet létre lehet hozni, továbbá az egyes modellelemek szemantikáját tetszőlegesen hosszú példányosítási láncban meg lehet határozni *(„deep instantiation”*). A **többszintű** (multi-level) metamodellezés [3] e két probléma megoldására törekszik. A többszintű metamodellezés egyik kulcsa, hogy osztályok és objektumok helyett azok összemosásával dolgozik (a *class* és az *object*

szavakból képzett „*clabject*”-ként hivatkozva az új fogalomra [3]). Egy clabject képes osztályként viselkedni és ezáltal példányosítható, ugyanakkor viszont ezzel egy időben objektumként is használható, azaz konkrét értékek is tárolhatóak benne. A clabjectek közti relációkat több szinten adhatjuk meg, nem korlátozva a MOF által definiált négy szintű hierarchiára. A szakterületi modellezők így képesek arra, hogy leküzdjék a korábbi módszerek által lefektetett akadályokat azáltal, hogy újabb és újabb rétegek bevezetésével tudják finomítani az adott szakterületi nyelvet, amíg az el nem éri a teljesen konkretizált állapotot.

A multi-level megközelítéssel szemben a **többrétegű** (multi-layer) metamodellezés [4] egy másfajta irányvonalat képvisel, melynek lényege hogy elhagyja a szigorúan vett szinteket és a *„deep instantiation”*-t a modellelemekre megfogalmazott megkötések lépésről-lépésre történő entitások és nem modellezési szintek közti szigorításával éri el. Ezen felül a többrétegű megközelítés fontos célja még, hogy önleíró legyen, lehetővé tegye a modellezési mechanizmusok és szakterületi modellelemek szétválasztását, illetve a *„clabject”*-eken végezhető műveletek validált modellezését.

A Dynamic Multi-Layer Algebra (DMLA) [5] [6] [7] [8] egy jelenleg alapkutatás fázisban járó új generációs többrétegű metamodellezési keretrendszer, mely lehetővé teszi a felhasználói számára, hogy validált és önleíró módon, fokozatosan definiálják és finomítsák a szakterület egyes fogalmait és kapcsolatait. A DMLA koncepcióját megalkotó kutatócsoport az elméleti alapok kidolgozását követően az elmúlt évben a gyakorlati felhasználásra helyezte a hangsúlyt. A munka során fény derült rá, hogy az eredetileg fordító (compiler) alapon működő modellfeldolgozás hátrányos, interpreter alapon, egy virtuális gép megalkotására van szükség, ami képes a modelleket tárolni, futtatni és validálni. Dolgozatom célja, hogy bemutassam a munkámat, ami ezen új modellfeldolgozó kidolgozása volt, melynek eredményeképpen elérhetővé vált a valódi, dinamikus, interaktív működés. A munkám lehetővé teszi, hogy a keretrendszer segítségével készített modelleket valósidőben módosítsuk, illetve validáljuk, így kiterjesztve az eszköz lehetőségeit és elérhetővé téve az ipari felhasználást is.

A dolgozat felépítése a következő: a 2. fejezetben ismertetem a DMLA alapvető felépítését és működését, majd a 3. fejezetben bemutatom a modellek valósidejű, dinamikus módosításának megvalósításához használt virtuális gép (VM) technológiát, a GraalVM-et és a Truffle-t. Ezt követően a 4. fejezetben részletezem a projekten végzett munkámat, aminek keretében a DMLA és a virtuális gép ötvözésével kidolgoztam a

keretrendszer teljes interpreter alapú feldolgozását és futtatását. Végül az 4.4. fejezetben összehasonlítom a régi fordításalapú és az általam készített új interpreter alapú DMLA-t egy példa szakterület segítségével, majd a 5. fejezetben összefoglalom az elért eredményeket és bemutatom a DMLA jövőjét.

2 Dynamic Multi-Layer Algebra

Ebben a fejezetben ismertetem a munkám alapjául szolgáló többrétegű keretrendszer, a DMLA alapvető részeit. Fontosnak tartom kiemelni, hogy a fejezetnek nem célja a DMLA minden részletre kiterjedő és precíz bemutatása, az ismertetés csak olyan mértékben és mélységben szerepel, ami később az általam végzett munka megértéséhez és értékeléséhez szükséges.

2.1 A DMLA részei

A DMLA alapvetően két részre választható szét: (i) a Core-ra, ami definiálja a modellezési struktúrát: az adatszerkezetet és az alapvető kezelőfüggvényeket, illetve (ii) a Bootstrap-re, ami az alapvető építőelemeket és validációs logikát tartalmazza a szakterületek definiálásához és használatához.

A Core által meghatározott alapvető építőelemek négyesek (tuple-ök), melyek a modellelemeket (entitásokat) írják le. A négyes a következő adattagokat tartalmazza:

- az entitás azonosítója (ID),
- az entitás meta entitásának az azonosítója (meta ID)
- a tartalmazott entitások, azaz attribútumok listája (ID lista)
- az entitásban tárolt konkrét értékek (érték lista).

A Core nem csupán az adatformátumot definiálja, hanem erre alapulva egy absztrakt állapotgépet az Abstract State Machine (ASM) [9] formalizmust követve. Az ASM állapotai egy-egy pillanatképet reprezentálnak a modellről. Ezeket az állapotokat (és ezáltal a modellelemeket) az ASM formalizmusnak megfelelően függvények segítségével érhetjük el. A függvények közt vannak lekérdezők és vannak az állapotot megváltoztatók is. Pl. az értékadás, vagy az új entitás létrehozása is ilyen függvények segítségével történik. Bár az ASM lehetővé teszi a DMLA működésének formalizálását, ez csupán a kereteket adja meg, a gyakorlati felhasználáshoz nem elegendő. Szükség van ugyanis alapelemekre, amelyek a szerkezetet „élővé” teszik. Mondhatnánk azt is, hogy a Core által definiált központi reprezentáció a „hardver”. Erre a „hardverre” épül az „operációs rendszer”, azaz a Bootstrap, amely egy fix pontot képez a DMLA architektúrájában, melyre a különböző „alkalmazások”, azaz modellek és szakterületek

támaszkodhatnak. Ez a szeparáció a Core és a Bootstrap között lehetővé teszi, hogy a Bootstrap cserélhető legyen, fenntartva annak a lehetőségét, hogy később a szakterületi modellek építéséhez és validációjához más megközelítést alkalmazhassunk. Fontos azonban megjegyezni, hogy a Bootstrap szerepéből adódóan a DMLA implementációja nem teljesen független tőle, így a cseréje esetén az implementációt is módosítani kell bizonyos mértékben.

2.2 Többrétegű modellezés DMLA alapon

Mielőtt kitérnék a Bootstrap részletesebb bemutatására, röviden ismertetem a DMLA néhány alapkoncepcióját, melyek közül az egyik legfontosabb egy sok tekintetben egyedi *példányosítás* reláció. Amikor a DMLA-ban példányosítunk, az új entitás a meta entitásához („ősosztályához") képest új megszorításokat vezethet be, konkretizálja az őselemben megtalálható definíciókat. Ahogy haladunk egyre lejjebb a meta hierarchiában, egyre több megszorítást teszünk, a kezdetben teljesen általános modellelemek finomításával így a hierarchia végén teljes mértékben konkretizált entitások hozhatóak létre. A kezdeti rugalmasság a folyamat során minden részletében kitöltött, konkrét adatokat tartalmazó entitássá változik.

A példányosítás ilyen módon való megközelítését két tulajdonság biztosítása teszi lehetővé: (i) a *„lazy instantiation"* és (ii) a *„fluid meta-modeling"*. A *„lazy instantiation"* azt jelenti, hogy a felhasználó nem köteles az adott absztrakciós szinten levő összes modellelemet egyszerre példányosítani, így a különböző típus-példány relációkat alacsonyabb szinten is képes kezelni. A *„fluid meta-modeling"* azt jelenti, hogy bármelyik modellelem bármelyik másik modellelemre hivatkozhat a meta-hierarchiában, amennyiben a hivatkozás nem ütközik valamilyen validációs szabályba. Ez a két tulajdonság együtt lehetővé teszi, hogy a példányosítás relációt több szinten alkalmazzuk, de a klasszikus többszintű metamodellezéshez képest szabadabb formában [4].

Láthatjuk, hogy a validálás fontos szerepet játszik a példányosítás folyamatában, ugyanis a keretrendszer minden egyes entitás-meta entitás kapcsolat esetén ellenőrzi, hogy a példányosítás valóban helyes-e. Maga a validációs logika azonban – sok más keretrendszerrel ellentétben – nem egy hagyományos programozási nyelven van megvalósítva, hanem modellezett műveletek segítségével magában a Bootstrap-ben van megadva. Ezt a validációs logikát az egyes szakterületek ki is egészíthetik igény szerint. Ez azt jelenti, hogy maga a validációs logika is teljes mértékben modellezett, azaz része

az entitáshalmaznak, amelyből a modell áll, így a validáció ezen entitásokra is lefut. Úgy is mondhatjuk, hogy a validáció is validálásra kerül.

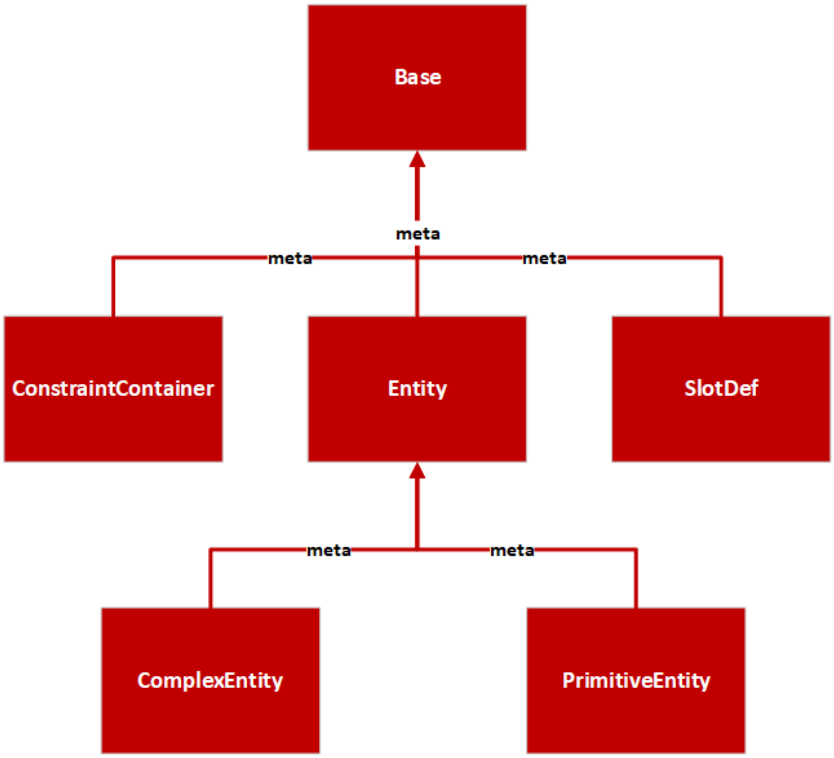
A modellezett validáció igen mély szinten áthatja a DMLA modellezési struktúráját. A legjobb példa erre, hogy a típusok sem a hagyományos módon kerülnek modellezésre, hanem az egyes elemekhez kapcsolt, szintén modellezett típuskényszerek segítségével. Hasonló módon kezeljük a számosság ellenőrzését is.

Fontos, hogy annak ellenére, hogy a validáció modellezett, a legalapvetőbb műveletek, mint az entitások adatainak lekérdezése vagy maguknak az entitásoknak a lekérdezése logikailag a mögöttes absztrakt állapotgép részét képzik. A gyakorlatban ez úgy jelenik meg, hogy ezeket a beépített műveleteket az ASM absztrakt állapotgépét reprezentáló alkalmazásban hagyományos módon egy külső programozási nyelv segítségével valósítottuk meg.

2.3 A Bootstrap elemei

A DMLA működésének megértésében fontos szerepet játszik a Bootstrap alapvető elemeinek ismerete, így ebben a fejezetben erre külön ki szeretnék térni [10].

Ahogyan a 3. ábra is mutatja, a Bootstrap és így minden entitás őse a **Base**. Ez egy nagyon speciális entitás, aminek nincs metája. A meta nélküliség következtében a példányosítás során nincs validálandó feltétel, ezért alapértelmezetten helyesnek tekintjük. A Base ennélfogva egy axiómaként fogható fel, amire a DMLA többi elemét felépítjük.



1. ábra: Egyszerűsített Bootstrap hierarchia

A **Base** három fontosabb funkcionalitást is elérhetővé tesz a példányai számára. Egyrészt a **SlotDef** entitás tartalmazásával lehetővé teszi slotok definiálását, ahol a slot egy olyan konstrukció, amivel az entitás egy adott tulajdonságát írhatjuk le, hasonlóan az osztályok property-jeihez az objektumorientált nyelvek esetén. Másrészt a **ConstraintContainer** entitás tartalmazásával lehetővé teszi, hogy az egyes modellelemekre megkötéseket adjunk meg. Ide kapcsolódik a **Base** harmadik funkciója: a **Base** tartalmazza a validáció alapjait, az ún. alfa, béta és gamma validációs formula képében. Mindhárom formulának meghatározott feladata van a validáció során, továbbá maguk a megkötések is rendelkeznek (ezek részletes ismertetése túlmutat ezen dolgozat keretein, de a [10] publikációban megtalálhatóak) saját validációs formulákkal. Ezen a szinten mindhárom formula nagyon egyszerű és megengedő. A példányosítás során az egyes entitások kiegészíthetik (szigoríthatják) a validációs formulákat, viszont semmiképp sem lazíthatnak rajtuk. Ezt az elvárást a rendszer úgy biztosítja, hogy egy adott entitás validálásakor nem csupán a közvetlen ősben megadott formulák kerülnek ellenőrzésre, hanem a metahierachia minden feljebb eső szintjén lévők is.

A **SlotDef** meta entitása a **Base**, amely furcsán hangozhat a függőségek szempontjából, azonban mivel egyrészt az entitások és attribútumaik feldolgozása nem a referenciák követésével történik, másrészt mivel a validáció nem egyszerre fut a két

entitáson, így ez a látszólagos ellentmondás nem okoz gondot. A **SlotDef** lehetővé teszi slotok létrehozását, valamint változtatás nélkül újra felhasználja a **Base**-ből a **ConstraintContainer**-t, ezzel lehetővé téve, hogy a slotokra megkötéseket helyezhetünk. A slot ennélfogva egy „felokosított” attribútum, amihez kényszerek is csatolhatóak.

A **ConstraintContainer** meta entitása is a **Base,** de ez a **SlotDef** esetén már látott okokból, itt sem okoz gondot. Felmerülhet a kérdés, hogy miért van szükség egy külön konténerre a megkötésekhez? Ennek az az oka, hogy az extra réteg hozzáadásával a megkötések könnyebben elválaszthatóak az entitásoktól, így egyszerűbb őket kezelni és feldolgozni.

A kényszerek kapcsán két alapvető megszorítás típust fontos megemlíteni: (i) a **TypeConstraint**-et és (ii) a **CardinalityConstraint**-et. A **TypeConstraint** típusellenőrzést biztosít a hozzá kapcsolt elemen. Ez azért fontos, mert a DMLA-ban az elemeknek nincs a klasszikus értelemben vett (beépített módon megkövetelt) típusuk, hanem csak típuskényszerek segítségével lehet ezt megadni. A **CardinalityConstraint** egy nem kevésbé hasznos ellenőrzést végez mivel abban segít, hogy egy entitás vagy egy slot példányainak kardinalitását testre lehessen szabni.

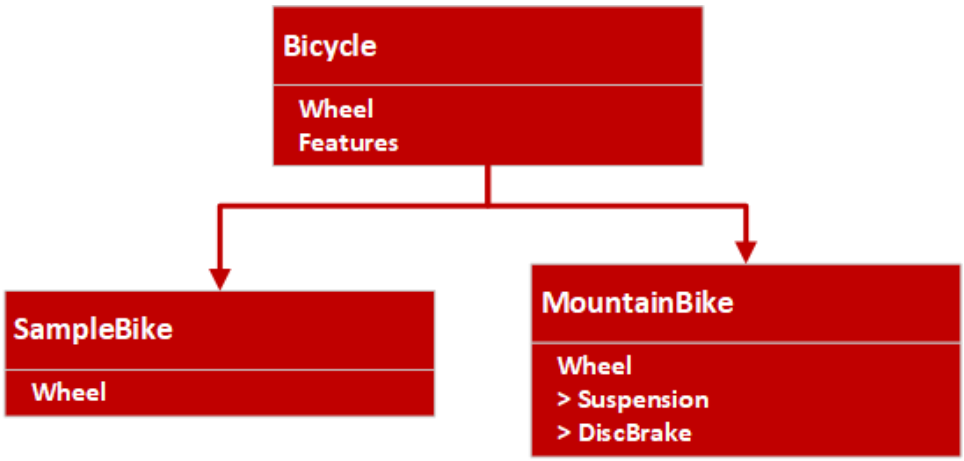
Ezen a ponton érdemes visszatérnünk kicsit a slotokra és a validációra. Ahogy korábban szerepelt, az egyes entitásoknál kötelező megadni a meta entitást, ami az adott entitás szerkezetével kapcsolatos szabályokat adja meg. A szabályok megadása a meta elemhez kapcsolt kényszerek segítségével történik. Ezzel analóg módon az entitásokban található slotok mindegyike hivatkozik egy meta slotra, ami a slot szerkezetére vonatkozó szabályokat adja meg. Mivel a **SlotDef** a **Base**-hez hasonlóan tartalmazza a **ConstraintContainer** elemet, a két validálandó logikai egység azonos módon kezelhető.

A példányosítás során egy slottal négyféle műveletet tudunk végezni: (i) klónozhatjuk, (ii) konkretizálhatjuk, (iii) elhagyhatjuk vagy (iv) feloszthatjuk. A 2. ábra szemlélteti a klónozás és a konkretizálás műveletét. Láthatjuk, hogy a **Bicycle** elem két slotja a **Wheel** és a **Features**. Ez azt modellezi, hogy egy biciklinek lehetnek kerekei, valamint más képességei. Ezután példányosítunk a **Bicycle**-ből egy **MountainBike**-ot, ahol a **Wheel** attribútumot klónozzuk (hiszen a Mountain bike kerekeire nem akarunk eltérő szabályokat megadni), a **Features** attribútumot pedig konkretizáljuk, megadva, hogy a **MountainBike**-nak van felfüggesztése. A konkretizálás itt a slothoz definiált kényszerek pontosítását jelenti, konkrétan ebben az esetben azt, hogy az eredeti

típuskényszert pontosítva megadjuk, hogy a slot értéke csak egy felfüggesztés lehet, nem bármilyen képesség (az ábra ezt nem tartalmazza).

2. ábra: Attribútum klónozás és konkretizálás

A 3. ábra az elhagyás és felosztás műveletét mutatja be. Az előbbi példához hasonlóan, a **Bicycle**-nek továbbra is van egy **Wheel** és **Features** attribútuma, azonban most két példányt hozunk létre, egy **SampleBike**-ot és egy **MountainBike**-ot. A **SampleBike** esetén, elhagyjuk a **Features** attribútumot. Az elhagyás azért lehetséges, mert a meta slot (**Features**) kényszerei megengedik a null kardinalitást. A példányosítás másik ágán, a **MountainBike** esetén, felosztjuk a slotot, nem egy, hanem két konkrét képességet megadva (felfüggesztés és tárcsafék). A slot kardinalitása ebben az esetben is alapvető fontosságú.



3. ábra: Attribútum elhagyás és felosztáss

Miután definiáltuk az alapvető elemeinket, lehetőségünk van az entitások struktúrájának a definiálására, amiben az **Entity** elem és két példánya a **PrimitiveEntity** és a **ComplexEntity** ad segítséget. A nevéből fakadóan a **PrimitiveEntity** feladata a primitív típusok (szám, szöveg, logikai típus stb.) reprezentálása, míg a **ComplexEntity** feladata gyakorlatilag az összes többi entitás, így lényegében a szakterületek entitásainak reprezentálása.

2.4 DMLAScript

Önmagában a DMLA és a mögötte álló tuple rendszer kevés ahhoz, hogy kényelmesen és gyorsan tudjunk modelleket készíteni, mivel a tuple-ök összekapcsolásából adódó struktúra precíz ugyan, de éppen ezért túlságosan bonyolult. Más megközelítésben, ha tuple-ök használatával szeretnénk egy adott szakterületi modellt összeállítani, az olyan lenne mintha assembly-ben akarnánk programozni. Minden feladatot meg tudnánk oldani vele, de túlságosan alacsonyszintű, ezért nehézkes lenne. Ennek megfelelően mind a Bootstrap, mind a DMLA-hoz írt szakterületi nyelvek a keretrendszer saját programozási nyelvével, a DMLAScript-tel készülnek. A DMLAScript képes leírni az entitások szerkezetét és a műveleteiket is. A DMLAScript nagy mértékben hasonlít a hagyományos programozási nyelvekhez, azonban a különleges felhasználási terület miatt sok mindenben különbözik is tőlük.

A következőkben néhány konkrét példán keresztül szemléltetjük a DMLAScript használatát. A példáknak nem célja a nyelv teljes és precíz bemutatása, hanem csak a tipikus műveletekre alkalmazható tipikus megoldásokat szeretném vázolni. A 4. ábra egy DMLAScript-ben írt entitást ábrázol, a **ProRaceFrameComponent**-et. Láthatjuk, hogy egy entitás létrehozása hasonlít egy osztály definiálásához, azaz van egy neve és van egy „ősosztálya”, azaz metája (itt **RacingFrameComponent)**. Ezen belül láthatjuk a különböző klónozott attribútumokat, mint pl. **Component.Weight**, **BicycleComponent.Material**, valamint egy új slot definiálását is a megfelelő meta-slot és egy érték megadásával együtt. A slot felett láthatjuk a megszorítások megadásának módját is, melyeket ebben az esetben a **Base** entitásból vettünk át.

```
ProRaceFrameComponent: RacingFrameComponent {
                Component.Weight;
                Component.Size;
                Component.Colour;
                Component.SerialNumber;

                RacingFrameComponent.TopTubeLength;
                RacingFrameComponent.DownTubeLength;
                RacingFrameComponent.SeatTubeLength;
                BicycleComponent.Material;
                BicycleEntity.AbstractEntity;

                @Base.AlphaValidation.OpSig
                @Base.AlphaValidation.T
                @Base.AlphaValidation.C
                slot AlphaValidation :
                        RacingFrameComponent.AlphaValidation =
                            $ProRaceFrameAlpha;
}
```

4. ábra: DMLAScript entitás

A 5. ábra a műveleti nyelvet ábrázolja, amelyben megtalálhatóak a hagyományos programozási konstrukciók, mint vezérlési szerkezetek („if”, „for”) vagy változó deklaráció és definíció („result”). Azonban ahogyan az imént említettem, vannak érdekességek is, mint a függvényhívás a „call” kulcsszó segítségével vagy a beépített függvényhívás jellegű kifejezések („append”).

```
operation ID[] GetCopiesAndInstances(ID[] attributes, ID metaAttribute) {
        ID[] result = ID:[];

        for(ID attr : attributes) {
                if(attr==metaAttribute || call $Meta(attr)==metaAttribute)
                        append<ID>(result, attr);
        }

        return result;
}
```

5. ábra: DMLAScript művelet

Fontos megjegyezni, hogy a DMLAScript csak egy szintaktikai édesítőszer a DMLA-hoz, amely azért lett létrehozva, hogy leegyszerűsítse a keretrendszer használatát, azonban minden, amit DMLAScript-tel írunk le, adattá, tehát tuple-é képződik le a script feldolgozásakor. Ez vonatkozik mind az entitásokra (a szakterületi nyelv fogalmaira), mind az entitás metájából átvett attribútumokra, ill. az entitás slotjaira és értékeire. A slotokon definiált megkötések, melyek pl. a slotok típusával és kardinalitásával kapcsolatos korlátozásokat írják le, illetve a műveletek is, melyek a modell validációját és dinamizmusát tartalmazzák, szintén tuple-re képződnek le. A kódból adattá való

leképezés egyrészt csökkenti a validáció komplexitását, másrészt lehetővé teszi az optimalizációt, miközben a modellek fokozatos konkretizálása továbbra is lehetséges marad.

A keretrendszer jelenlegi implementációjában az entitások és a műveletek a leképezés után XML-ként lesznek elmentve. Nézzük meg azt az XML leképezést, mely a 4. és 5. ábra feldolgozása után keletkezik. Láthatjuk, hogy a 4. ábra entitásából a 6. ábra XML-je keletkezett, amiben ugyanúgy megtalálhatjuk az entitás nevét és metáját, de immáron a csomagnevekkel együtt. Emellett a tag tartalmazza az attribútumokat is, melyek ugyancsak egy-egy másik XML entitást rejtenek maguk mögött.

```
<Entity ID="Bootstrap.Entity" Meta="Bootstrap.Base">
    <Attributes>
      <Attribute>Bootstrap.SlotDef</Attribute>
      <Attribute>Bootstrap.ConstraintContainer</Attribute>
      <Attribute>Bootstrap.Base.AlphaValidation</Attribute>
      <Attribute>Bootstrap.Base.BetaValidation</Attribute>
      <Attribute>Bootstrap.Base.GammaValidation</Attribute>
      <Attribute>Bootstrap.Entity.CanContainValue</Attribute>
    </Attributes>
</Entity>
```

6. ábra: Entitás XML

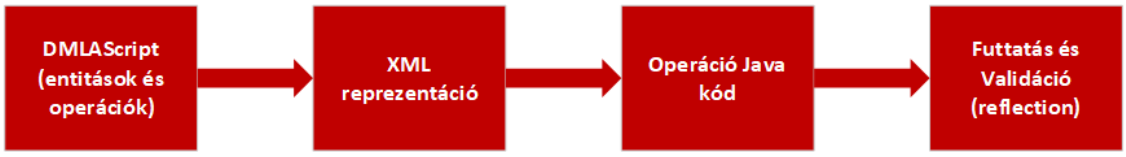
Végül a 7. ábra jól szemlélteti, hogy a műveletekből is generálódik XML, mégpedig a modell entitásaihoz igen hasonló formában. Kiemelném azonban, hogy az ábra által mutatott, a művelet definíciójából generált utasítások azonosítói csak az XML olvashatósága miatt rendelkeznek ilyen speciális azonosítóval. Valójában ezen a szinten az entitások neveinek már nincsen jelentősége, így az bármi lehetne, amíg az entitások között ez konzisztens és egyedi marad.

```
<Entity ID="Bootstrap.Validation.Helper.GetCopiesAndInstances"
Meta="Bootstrap.Operations.OperationDefinition">
        <Attributes>
            <Attribute>Bootstrap.Validation.Helper.GetCopiesAndInstances.Re
            turnType</Attribute>
            <Attribute>Bootstrap.Validation.Helper.GetCopiesAndInstances.Pa
            rams0</Attribute>
            <Attribute>Bootstrap.Validation.Helper.GetCopiesAndInstances.Pa
            rams1</Attribute>
            <Attribute>Bootstrap.Validation.Helper.GetCopiesAndInstances.De
            finition</Attribute>
        </Attributes>
</Entity>

<Entity
ID="Bootstrap.Validation.Helper.GetCopiesAndInstances.Definition.Block.Stat
ements" Meta="Bootstrap.Operations.Block.Statements">
        <Values>
            <Value
            Type="Reference">Bootstrap.Validation.Helper.GetCopiesAndInstan
            ces.Definition.Block.Statements.St0.Declaration</Value>
            <Value
            Type="Reference">Bootstrap.Validation.Helper.GetCopiesAndInstan
            ces.Definition.Block.Statements.St1.Foreach</Value>
            <Value
            Type="Reference">Bootstrap.Validation.Helper.GetCopiesAndInstan
            ces.Definition.Block.Statements.St2.Return</Value>
        </Values>
</Entity>
```

7. ábra: Művelet és utasításai XML

Miután minden entitásból legeneráltuk a megfelelő XML-t, a műveletek feldolgozása következik. Ebben a lépésben a műveletek XML reprezentációjából Java kódot generálunk, amely lényegében az ASM-et reprezentálja. Ezt követően a kódot lefordítjuk és a keretrendszer dinamizmusa miatt reflectionnel végrehajtjuk. A végrehajtás belépési pontja a modell validációs művelet, amely elindítja a teljes modell validálását és jelzi ennek eredményét a felhasználó számára. Ezt a teljes, fordítás alapú folyamatot a 8. ábra mutatja be.



8. ábra: Fordítás alapú DMLA folyamata

Végső soron elmondhatjuk, hogy a teljes folyamatnak és a DMLA-nak az egyik legfontosabb funkciója a validáció, azaz annak a kérdésnek a megválaszolása a szakterület fejlesztője számára, hogy az elkészült modell helyes-e vagy sem. Előfordulhat azonban, hogy ennél többet szeretnénk elérni, dinamikusabbak szeretnénk lenni. Tegyük

fel, szeretnénk módosítani a modellen valamit, például létrehoznánk egy új entitást. Korábban a modell módosításával kapcsolatos műveleteket csak úgy tudtuk megtenni, hogy a teljes folyamatot újra elvégeztük. Azaz módosítottuk a modell DMLAScript forrását, amelyből XML majd Java lett, amit fordítottunk és futtattunk. Ez egy lassú és statikus folyamat volt, melyet szerettünk volna kiküszöbölni és a modellt azonnal módosítani és validálni. Az általam megvalósított új rendszer, a DMLA és a GraalVM ötvözése pontosan ezekre az igényekre nyújtott megoldást, illetve – ahogy a következő fejezetekből látszik majd – ennél többet is.

3 GraalVM és Truffle

Ebben a fejezetben az új, interpreter alapú DMLA megvalósításához szükséges technológiákat ismertetem, melyek lehetővé teszik a szakterületi modellek futásidejű létrehozását és manipulációját.

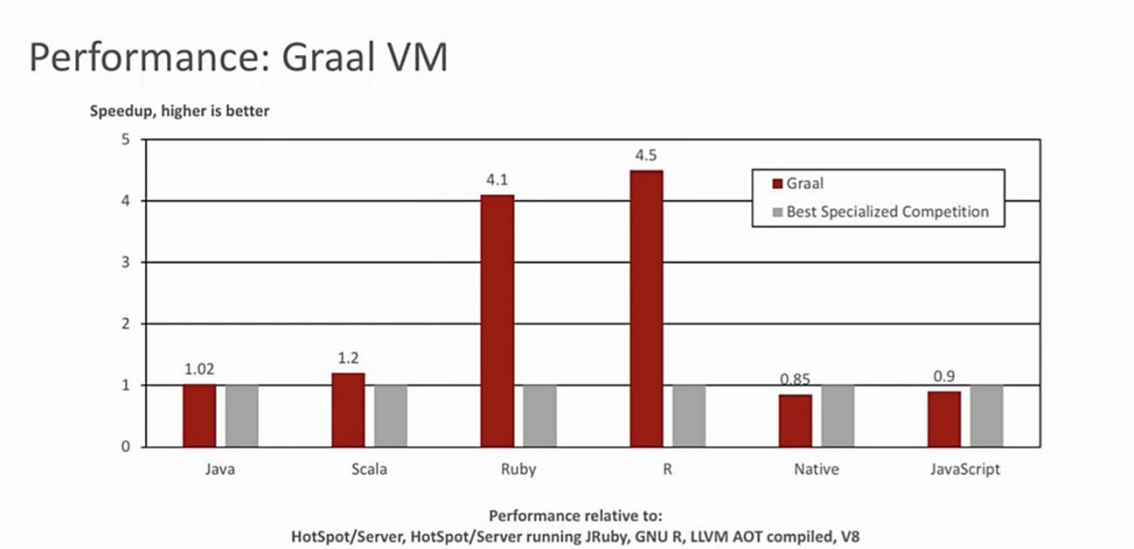
3.1 GraalVM

A GraalVM egy az Oracle Labs által fejlesztett kutatási projekt [11], mely egyrészt egy Java-ban írt új univerzális virtuális gép, másrészt egy JIT fordító, amelyet a modern Java platform által biztosított lehetőségeknek köszönhetően a jelenlegi Java virtuális géphez, a Hotspot VM-hez lehet illeszteni. A GraalVM célja, hogy leváltsa a régi, C++ alapú C2 fordítót, ugyanis a C2 karbantartása egyre nehezebb feladattá vált az utóbbi időben. Ezek a karbantartási nehézségek vezették a Java közösséget és az Oracle-t ahhoz, hogy megkezdjék a Graal fejlesztését. A váltás lehetővé teszi a fejlesztők számára azt is, hogy a jól ismert fejlesztési eszközöket és környezeteket felhasználva fejlesszék és javítsák a virtuális gépet és a fordítót, hiszen mindkettő Java alapú.

Az Oracle Labs a fejlesztés könnyítése mellett azt a célt tűzte ki a GraalVM-mel, hogy egy univerzális virtuális gépet készítsenek. Ehhez egyrészt a nyelvek igen széles választékát kell támogatnia a koncepciónak, másrészt teljesítményében összemérhetőnek kell lennie a natív megoldásokkal.

A projekt haladását igen jól mutatja, hogy mindkét cél tekintetében igen előrehaladott állapotban van. Jelenleg is széles választékát támogatja a legkülönbözőbb programozási nyelveknek, a Pythontól kezdve, a Javan át, egészen a C és C++-ig. Ezen felül a teljesítménye legrosszabb esetben megközelíti, sok esetben pedig túl is szárnyalja az adott nyelv hivatalos natív megvalósítását, ahogyan azt a 9. ábra mutatja.

A keretrendszer ugyancsak lehetővé teszi a különböző Graal felett futó nyelvek együttműködését, képessé téve a felhasználókat arra, hogy az egyik nyelvből áthívjanak egy másikba vagy az egyik nyelv objektumait felhasználják egy másikban.



9. ábra: GraalVM teljesítménye

3.2 Truffle

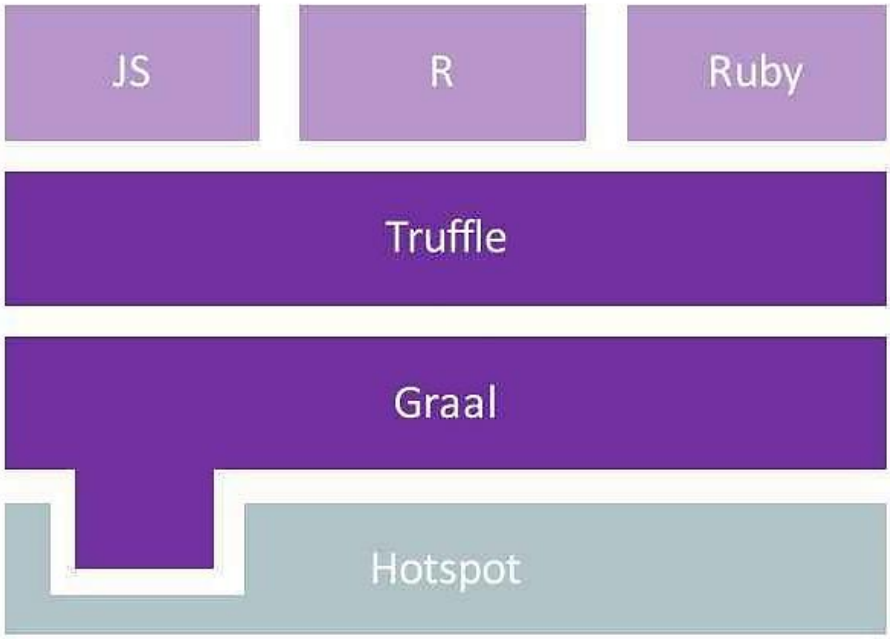
Az előző fejezetben megtudtuk, hogy a GraalVM legfontosabb célja, hogy egy univerzális virtuális gép legyen, azonban egy ilyen gép fejlesztése mögött felmerül a kérdés: miért nem készítünk egy univerzális programozási nyelvet?

A válasz nagyon egyszerű: minden nyelvnek megvan a maga előnye és hátránya. Egy olyan alacsony szintű nyelv esetén, mint a C megvan az az előnyünk, hogy teljes kontrollt kapunk a hardver felett, ami ideális beágyazott rendszerek fejlesztéséhez, azonban kényelmetlen egy asztali alkalmazás fejlesztése esetén, nem beszélve a számos hibalehetőségről, amit a memória manuális menedzselése okoz. Ezzel szemben a C# vagy Java nyelv pont a rapid application development (RAD) területén a leghasznosabb. Tehát mindenképp szükségünk van olyan nyelvekre, amelyek adott helyzetekre és problémákra vannak szabva.

Ezek az alapvető gondolatok vezették az Oracle-t és az Oracle Labs-t arra a következtetésre, hogy egy univerzális nyelv helyett készítsünk egy univerzális futtatókörnyezetet, ami olyan közel áll a natív világ teljesítményéhez, amennyire csak lehet. Ebből következik a GraalVM projekt mottója is: „One VM to rule them all”. Egy kérdés azonban megválaszolatlan maradt. Hogyan implementálhatjuk a saját nyelvünket az Oracle rendszerébe, legyen szó akár egy ismert, akár a saját egyedi nyelvünkről?

A válasz, hogy szükségünk van egy olyan keretrendszerre, amivel tetszőleges programozási nyelvet képesek vagyunk integrálni a GraalVM által biztosított környezetbe. Ezt a keretrendszert hívják Truffle-nek [12]. A Truffle egy nyelvfüggetlen

API, aminek a segítségével közel bármilyen nyelvet megvalósíthatunk a Graal fölött. Továbbá támogatja az általunk készített nyelv instrumentálását, optimalizációját és debuggolását is. Végül, de nem utolsó sorban képes a nyelvek közötti kapcsolatot megteremteni, így a különböző nyelvek interakcióba léphetnek egymással. A GraalVM és Truffle együttes architektúráját a 10. ábra mutatja, ahol jól látható hogyan illeszkedik a GraalVM a jelenlegi Java architektúrába, illetve hogyan épül rá a Graal-ra a Truffle és a felhasználásával készült nyelvek.



10. ábra: A GraalVM és Truffle architektúrája [13]

3.3 Nyelvek implementálása

A kérdések megválaszolása után nézzük is meg hogyan néz ki egy GraalVM feletti nyelv implementálásának a folyamata:

Általában egy fordító elkészítésekor öt alapvető lépést kell a következő sorrendben elvégeznünk: (i) lexikai elemzés, (ii) szintaktikai elemzés, (iii) szemantikai elemzés, (iv) optimalizáció és (v) kódgenerálás. A lexikai elemzés során a lexer a nyers szövegből logikai egységeket, azaz tokeneket hoz létre. A tokeneket felhasználva a második lépésben a parser felépíti a szintaxisfát vagy absztrakt szintaxisfát (ST vagy AST). A szemantikai elemzés során az imént felépített fát szemantikailag validáljuk. Ezt követően a negyedik lépés során a validált fát optimalizáljuk, ügyelve arra, hogy az

optimalizált fa kimenete megegyezzen az eredetivel. Végül a kódgenerálási fázis során alacsony vagy magasszintű nyelvre fordítjuk az optimalizált fát.

A Truffle esetén nem tisztán egy fordítót készítünk, ugyanis az AST-ből nem kódot generálunk, hanem az API segítségével egy a GraalVM által érthető formára hozzuk az AST-t, amit nevezzünk Truffle AST-nek a továbbiakban. Így a folyamat közelebb áll egy értelmező elkészítéséhez, ahol az interpretálás során, sorról-sorra vesszük a kódot, ezt transzformáljuk egy hatékonyabb, köztes reprezentációra, majd azonnal végrehajtjuk.

Tehát első lépésben végre kell hajtanunk a fordítási folyamat első lépését, hogy előállítsunk egy AST-t a forráskódhoz. Manapság számos eszköz létezik arra, hogy egy adott nyelvtan alapján lexert és parsert generáljunk, így ezt a lépést nem kell kézzel elvégeznünk, csak a nyelvtant kell megírnunk. Az ajánlások szerint Truffle esetén az ANother Tool for Language Recognition (ANTLR) [14] eszközt érdemes használni a nyelvtan megvalósításához, illetve a lexer és parser generálásához. Az ANTLR amellett, hogy az egyik legnépszerűbb parser generátor, leginkább azért ajánlott a Graal alapú projektek esetén, mivel szinte minden példa és segédlet, amit Graal-hoz és Truffle-höz találhatunk, akár hivatalos akár nem, az ANTLR-t használja. Ezáltal biztosan elkerülünk minden inkompatibilitási problémát és nem utolsó sorban több példakódot is találhatunk, ami gyakran nagy segítség a jelenleg még erősen hiányos hivatalos dokumentáció miatt.

Miután a parser sikeresen lefutott az elkészült AST bejárásával az összes releváns csomópontot át kell alakítanunk egy vagy több Truffle csomóponttá. Ezekből a Truffle elemekből, amik lényegében a végrehajtandó kódot reprezentálják, végül felépíthetjük a Truffle AST-t. A GraalVM a fa minden egyes csomópontját az interpretálás során végre fogja hajtani, azonban az egyes csomópont típusok jelentését nekünk kell implementálnunk. Például, ha a nyelvünk támogatja az elágazásokat, akkor azt egy Truffle csomópontként implementálnunk kell, mely lényegében azt jelenti, hogy ki kell értékelnünk a feltételt, majd az így keletkező eredmény alapján el kell döntenünk, hogy végrehajtjuk-e a feltétel törzsét, vagy ha van, akkor ugorjunk a különben ágra.

Ha mindezzel megvagyunk, akkor van egy működő implementációnk, amit azonban még ki kell egészíteni némi optimalizációval. Az optimalizációra azért van szükség, mert a Truffle nem ismeri a nyelvünk apró részleteit, belső működését, így a beépített optimalizáció lehetőségei limitáltak, amelyet így nagyrészt nekünk kell elvégeznünk. A keretrendszer azonban számos lehetőséget nyújt az optimalizáció

mikéntjének megadására. Többek között lehetőség van a függvényhívások inline végrehajtására vagy cachelésére, illetve az utasítások implementációjánál optimalizálhatunk az operandusok típusa alapján és még számos egyéb módszert rejt a Truffle.

Ezek az optimalizációk nagymértékben képesek javítani a nyelvünk teljesítményét. Például a függvényhívások cachelése miatt nincs szükség arra, hogy az egyes hívásoknál kikeresse a keretrendszer a függvény belépési pontját. Emellett az inline függvények teljes egészében kiváltják a függvényhívást azáltal, hogy a Graal kicseréli a hívást a függvény törzsére. Ugyancsak sokat segíthet az utasítások típusspecifikus implementációja, másnéven a csomópont specializáció. Ez a módszer lehetővé teszi számunkra azt, hogy az egyes Truffle csomópontokat oly módon valósítsuk meg, hogy a csomópont paramétereinek típusai alapján a keretrendszer ki tudja cserélni az adott csomópontot a típusoknak megfelelő implementációra. Ezáltal az adott csomópont lefutása nem csak gyorsabb, mint az általános megközelítés, de az implementáció maga is olvashatóbb lesz azáltal, hogy minden típuskombinációra külön meg tudjuk adni a végrehajtás mikéntjét.

A teljes nyelvi implementációs folyamat sematikus bemutatását az 11. ábra tartalmazza.

11. ábra: Nyelvek implementálási folyamata Truffle-el

Fontos megemlíteni a nyelvek implementációján felül az integrációs lehetőségeiket is a GraalVM-be. Szám szerint három mód van arra, hogy a Graal indulásakor a nyelvünk is helyet kapjon a virtuális gépben: Az első, hogy a parancssorból megfelelő paraméterezéssel a Graal-hoz linkeljük a nyelvünket tartalmazó JAR fájlt. A második, hogy a Graal „language component” rendszerét használva telepítjük a nyelvünket a Graal futása közben. A harmadik és egyben legizgalmasabb, hogy lehetőségünk van a nyelvünkből és a Graal-ból egy natív binárist generálni. Ezáltal egyetlen futtatható állományban, mindenféle külső függőség nélkül helyet kap egy csökkentett méretű VM és az adott nyelv implementációja.

3.4 Projektek Graal alapon

A GraalVM térhódítása a Java közösségben egyre több aktív projektet teremtett az utóbbi években. Ezen projektek közül az egyik legnépszerűbb a FastR [15], ami a statisztikai számításokhoz készített R [16] programozási nyelvnek egy GraalVM és Truffle alapú implementációja. A FastR nagy előnye az eredeti GNU-R implementációval szemben, hogy sok esetben több nagyságrenddel gyorsabb nála, így egy igen ígéretes alternatívát nyújt. A projekt célja, hogy minél jobban megközelítse az R nyelv specifikációját, tehát bármilyen különbség a GNU-R és a FastR között hibának számít az implementáció szempontjából. Egy másik, a Graal közösség által készített népszerű projekt a Truffle Ruby [17], amely egy alternatív Ruby [18] interpreter. A FastR-hez hasonlóan egy nagy teljesítményű, minden Ruby alkalmazással kompatibilis megoldást nyújt.

Mind a FastR, mind pedig a Truffle Ruby jelenleg is fejlesztés alatt áll, ennek köszönhetően a fejlesztők, tehát az Oracle, egyelőre nem javasolja az éles környezetben való alkalmazásukat. Azonban mindkét implementációt ki lehet próbálni és a tervek szerint hamarosan ipari projektben is alkalmazhatóvá válnak. Továbbá ezen két projekt mellett, számos más nyelvnek és alkalmazásnak a fejlesztése zajlik [19], melyek a Graalt használják, mint futtatókörnyezet, illetve a Truffle-t alkalmazzák, hogy új és meglévő nyelveket implementáljanak, amelyek teljesítmény szempontjából felülmúlhatják az eredetit.

Mindezen projekteken felül, a GraalVM már sikeresen megmutatta képességeit ipari környezetben is, hiszen az egyik legnagyobb közösségi média platform, a Twitter, nemrégiben bejelentette, hogy a korábbi Scala futtatókörnyezet helyett a GraalVM-et alkalmazzák, legfőképpen az ezáltal nyert többlet teljesítmény miatt [20].

4 DMLA és GraalVM

Ebben a fejezetben az elvégzett munkámat ismertetem, melynek célja a DMLA GraalVM feletti futtatásának megvalósítása volt a Truffle segítségével, a modellek dinamikus létrehozása és manipulációja érdekében.

4.1 Bevezetés

A DMLA kutatás kezdetén egy elméleti módszert dolgoztunk ki az önleíró, önvalidáló többrétegű metamodellezés megvalósítására. Az elméleti síkon létező absztrakt állapotgépet a gyakorlatban is meg kellett valósítanunk, hogy a kutatási eredményeit gyakorlati módon igazoljuk. Bár ez az állapotgép inkább egy interpreter jellegű megvalósítással lett volna analóg, a technológiai korlátok és a praktikusság jegyében végül mégis egy compiler jellegű feldolgozást alkalmaztunk. Ahogy a 2.4 fejezetben ismertettem, a fordítás során az entitásokat és a műveleteket XML-be képeztük le, majd a műveletek XML megfelelőit Java kóddá transzformáltuk, végül az így előálló kódot reflection segítségével futtattuk. A végrehajtás során lefutott a validáció, melynek eredménye a modell helyességét adta meg. Bár ez az eredeti megközelítés bebizonyította, hogy a DMLA alapvető koncepciói helyesek, a mechanizmusok működőképesek, a gyakorlati felhasználás kapcsán több hiányosságot is magával hozott. A legfőbb hiányosság az volt, hogy a keretrendszer képességei gyakorlatilag egy pillanatkép validálására korlátozódtak. Ez azt jelenti, hogy össze lehetett rakni bármilyen bonyolult, többrétegű architektúrát, felhasználva a lépésenkénti validálás adta lehetőségeket, de ezzel a modellel a validáció volt az egyetlen interakció. A modell bármilyen változtatása újrafordítást igényelt és a dinamikusság, ami a DMLA nevében is megjelent, nem tűnt elérhetőnek.

Ezzel szemben egy interpreter alapú megvalósítás sokkal közelebb áll az ASM metodológiához, hiszen az interpreter mögött megbújó virtuális gép gyakorlatilag az absztrakt állapotgép közvetlen megvalósítása. A virtuális gép kidolgozására és az interpreter alapú feldolgozáshoz a kutatócsoport a GraalVM-et és a Truffle-t választotta azok korábban (3. fejezet) ismertetett előnyös tulajdonságai miatt.

A projekt során az én feladatom volt, hogy megismerjem mind a DMLA-t, mind a GraalVM által nyújtott lehetőségeket, majd kombinálva a két világot elkészítsem az új

DMLA-t, mely immáron interpretált környezetben, valós időben teszi lehetővé a modellek és szakterületek validációján túl azok szerkesztését és teljesebb feldolgozását is.

A feladat első fázisában a cél az volt, hogy a fordítás alapú DMLA által nyújtott funkciókkal és a keretrendszer kimeneteivel teljes egészében megegyező GraalVM alapú implementációt készítsek. Erre azért volt szükség, hogy bizonyítsuk az interpreter alapú megvalósítás létjogosultságát, illetve, hogy egy szilárd pontot biztosítsunk a továbbfejlesztésekhez az eddig elért eredmények megtartásával. Ezáltal ehhez a fázishoz az implementáció elkészítésén felül hozzátartozott az alapos tesztelés egységtesztek és a már meglévő szakterületi modellek felhasználásával, továbbá teljesítménytesztek elvégzése is.

Miután bizonyítottam, hogy a korábbi implementációval megegyező eredményeket és jobb teljesítményt kaphatunk az új implementáció segítségével, a második fázisban a továbbfejlesztések és a dinamizmus növelése került a középpontba. A lehetséges továbbfejlesztések közé tartozott az implementáció instrumentálása, hogy részletes képet kaphassunk a teljesítményről, illetve ide tartozott még a Bootstrap és a szakterületi modellek inicializásálának és validálásának szétválasztása a teljesítmény növelése érdekében. A második fázis legfontosabb célkitűzése azonban az volt, hogy egy valós idejű, a modellek megváltoztatását is lehetővé tevő interfészt biztosítsunk a keretrendszerhez, ezáltal elérve a dinamikus működést.

4.2 Első fázis

Az első fázisban tehát a cél az volt, hogy elkészítsem a korábbi fordítás alapú megközelítés GraalVM alapú implementációját, a lehető leginkább megközelítve a korábbi eredményeket. A DMLA és a GraalVM párosítása, illetve a DMLAScript implementálása a Truffle API segítségével egy meglehetősen komplikált feladat. A probléma nehézsége főként annak köszönhető, hogy a DMLAScript egy közel sem hagyományos programozási nyelv, hiszen ahogyan azt a 2.4 bekezdésben láttuk, a nyelv egyik fele leginkább egy leírónyelvre hasonlít, amivel a modellek struktúráját írjuk le, a másik fele pedig egy hagyományos programozási nyelv lehetőségeit és szintaktikáját követi. A műveleteket leíró nyelv ráadásul a kényszerek és így a típusrendszer adta magasszintű rugalmasságot is támogatja. Ez a kettőség az új technológiák bevezetése

során számos helyen megnehezítette a feladatomat, így ennek a fejezetnek a célja, hogy ezen nehézségeket és az ezekre adott megoldásokat bemutassa.

4.2.1 Futtatás

A GraalVM és Truffle bevezetésének első lépése az volt, hogy magát a GraalVM-et futtathatóvá tettem az általunk használt fejlesztési környezetben. Gyorsan kiderült, hogy már ez az egyszerűnek hangzó feladat sem triviális, mivel a GraalVM akkoriban korlátozott funkcionalitással volt csak elérhető Windows-on. Mivel publikusan elérhető egy az Oracle Labs által karbantartott Linux alapú GraalVM Docker konténer, így ezt felhasználva Windows alatt is el tudtam kezdeni a fejlesztést. A konténer használatával egyszerre több probléma is megoldódott, egyrészt a környezet beüzemelése egyszerűbb lett, másrészt mivel többen többféle operációs rendszert használunk, a konténer azonos körülményeket teremt a fejlesztéshez és futtatáshoz.

A futtathatóságot követte egy kezdeti teszt implementáció elkészítése, amellyel tesztelni tudtam, hogy a fejlesztési környezetben minden megfelelően működik és kompatibilis egymással. Ehhez a GraalVM-hez készített példa nyelvi implementációt, a Simple Language-et (SL) [21] használtam fel segítségként. Sajnos mivel a Graal még nem elég elterjedt így az elérhető dokumentáció hiányos, ami azt jelenti, hogy egyetlen nagy API dokumentáció található a Graal-hoz és minden más GitHub projektek formájában érhető el, ahol csak a forráskód nyújt támpontot. Ez igaz a SL-re is, így a teszt összeállítása az itt található kódbázisra alapult.

Ahhoz, hogy a teszt működjön és a Graal elinduljon a konténerben, két dologra van szükség. Egyrészt jelen kell lennie a Graal SDK-nak, másrészt a 3.3 fejezetben bemutatott három integrációs mód közül az egyik segítségével el kell indítani az implementációt. A három mód közül végül a build folyamat által előállított JAR linkelésére esett a választás, mivel ez volt a legegyszerűbb módszer. Ehhez az SL által is használt bash scriptet vettem alapul, illetve ezt módosítottam a mi felhasználási esetünknek megfelelően. Az így elkészült bash scriptnek a feladata, hogy a konténerben való meghívást követően elindítsa a teszt implementációt azáltal, hogy megfelelően paraméterezi a Java parancsot, illetve opcionálisan átadja az implementációnak a további, általunk hozzáadott paramétereket is. Miután megbizonyosodtam arról, hogy az összeállított környezetben megfelelően fut a Graal, elkezdtem az új DMLA

megvalósítását. Követve az 11. ábra által bemutatott folyamatot, az új generációs DMLA megvalósításának első lépése a lexer és parser elkészítése volt ANTLR segítségével.

4.2.2 Nyelvtan

A DMLA nyelvtan eredetileg Xtext-ben [22] készült, ami egy Eclipse alapú keretrendszer programozási-, és szakterület-specifikus nyelvek készítéséhez. Mivel a Graal esetén az ajánlott keretrendszer az ANTLR, ezért a meglévő Xtext-es nyelvtant át kellett írni ANTLR specifikus nyelvtanná. Ezt a folyamatot igen mechanikus módon sikerült elvégeznem, mivel minden Xtext-es lexer és parser szabály egyszerűen átmásolható egy ANTLR kompatibilis lexer vagy parser szabállyá. Ez annak köszönhető, hogy a másolás során csak ki kell hagynunk az Xtext specifikus szintaxist (pl. kereszthivatkozások), habár később emiatt bizonyos funkcionalitásokat, amelyeket eddig az Xtext biztosított, magunknak kell implementálnunk. Nézzünk egy példát:

```
Package
   : 'package' name=QN '{'
      imports+=Import*
      (
         entities+=ModelElement
      )*
      '}'
   ;
```

12. ábra: DMLA package Xtext-ben

```
packageBlock
    : 'package' qualifier '{' importStatement* modelElement* '}'
    ;
```

13. ábra: DMLA package ANTLR-ben

Mind a 12. ábra, mind a 13. ábra a DMLA-ban használatos Java-hoz hasonló csomagok parser szabályát írja le. Minden csomag a „package” kulcsszóval kezdődik, amelyet a csomag neve követ („QN”, azaz „qualified name”). A csomagon belül először nulla vagy több import utasítás található, majd nulla vagy több modell elem, azaz entitások és műveletek. Láthatjuk, hogy az Xtext-es változat esetén lehetőségünk van értékadáshoz és összefűzéshez hasonló műveletek elvégzésére: „name”, illetve „imports” és „entities”. A „name” egy speciális tulajdonsága az adott elemnek, amelyen keresztül más elemek referálhatnak rá, és amelyet később az Xtext keretrendszer képes feloldani. Az „imports” és „entities” esetén tulajdonképpen egy listához való hozzáfűzés történik, amelyet később az AST feldolgozása esetén el tudunk érni.

Ha megnézzük az ANTLR-nek megfelelő csomag szintaktikát, észrevehetjük, hogy bár nagyvonalakban megegyezik az Xtext-es megfelelőjével, de hiányoznak a speciális utasítások. Ennek az az oka, hogy az ANTLR nem támogatja az imént felvázolt funkciókat, tehát az AST feldolgozása során ezeket nekünk kell leprogramoznunk, amennyiben szükség van rá. Az átírás során a mi szempontunkból három olyan funkciót sikerült beazonosítanom, mely az Xtext-ben megtalálható, de az ANTLR-ben nem, azonban szükségünk van rá: (i) a szimbólumok névfeloldása, (ii) a kereszthivatkozások kezelése, (iii) a szabályok sorrendjének kiigazítása. Ezeken felül is van számos olyan kényelmi funkciója az Xtext-nek, melyek az átírás miatt hiányozhatnak, azonban a mi esetünkben az említett hármon kívül másra nem volt szükség.

4.2.3 Adatfeldolgozás

Miután az ANTLR nyelvtan alapján legeneráltam a lexert és parsert, illetve megbizonyosodtam arról, hogy az átírás teljes egészében fedi az eredeti nyelvtant, elkezdtem az adatok feldolgozásának az implementációját.

A megvalósítás során első megközelítésben a DMLAScript adatleíró részéből, tehát az entitásokból elkezdtem felépíteni a Truffle fát, azonban ezzel kapcsolatban rögtön felmerült az első fontos kérdés. A Truffle AST-ben helyet kapnak-e az adatentitások? Mivel a Truffle AST-t arra használjuk, hogy az egyes kifejezéseket később végrehajtsuk, ezért ha az adatentitások helyet kapnak itt, akkor fel kell tennünk a kérdést, hogy pontosan mit is jelent az a DMLA szempontjából, hogy végrehajtunk egy entitást? Azt a döntést hoztam, hogy a modellek tisztán adatleíró elemeinek nincs helye a végrehajtási fában, mivel a végrehajtás, mint művelet nincs rajtuk értelmezve. Ez egybeesik a korábbi implementációval, ahol az entitások és a műveletek, mint adatok egy egyszerű map-ben kaptak helyet, ahol a kulcs az entitás azonosítója, az érték pedig maga az entitás volt. Tehát végül ezt a megközelítést alkalmaztuk a GraalVM esetén is, azaz az adatfeldolgozás során nem építünk Truffle AST-t, feldolgozzuk a DMLAScript-et és a memóriában felépítünk egy entitás map-et, amelyet majd a futtatás során el lehet érni.

4.2.4 Névfeloldás

Az adatfeldolgozással párhuzamosan fejlesztettem az egyik legfontosabb Xtext funkció megfelelőjét, a névfeloldást. Minden entitás azonosítója valójában egy úgynevezett „fully qualified name”, azaz tartalmazza az összes csomag és minden más egyéb entitás azonosítóját, amely az adott entitást tartalmazza. A feldolgozás pillanatában

a map-be ezzel a névvel kerülnek be az entitások. Ennek tudatában a névfeloldás feladata kettős. Egyrészt biztosítja, hogy minden hivatkozás tényleg egy létező entitást takar, másrészt a DMLAScript „$” -al jelölt hivatkozásai esetén az aktuális kontextus alapján feloldja a hivatkozott nevet és lecseréli azt a hivatkozott entitás teljes nevére. Nézzünk is mindkettőre egy-egy példát:

```
Primitive : Entity { }

String : Primitive { }
```

14. ábra: Névfeloldás – hivatkozás

```
RelationType : Enum {
        slot Values : Enum.Values =
                [
                        $RelationTypes.LesserThan,
                        $RelationTypes.GreaterThan,
                        $RelationTypes.LesserThanOrEquals,
                        $RelationTypes.GreaterThanOrEquals
                ];
}
```

15. ábra: Névfeloldás – feloldás

A 14. ábra esetén a névfeloldás feladata, hogy a **String** entitás **Primitive** metájához megkeresse a **Primitive** entitást. Vegyük észre, hogy amennyiben a két entitás fel lenne cserélve, úgy a feloldás már nem lenne elvégezhető az entitás feldolgozásának idejében. Tehát a névfeloldásnak mindenképp az összes entitás feldolgozása után kell lefutnia.

A 15. ábra esetén a hivatkozások feloldásához meg kell keresni, hogy a hivatkozás pillanatában és kontextusában pontosan melyik entitásról van szó. Ehhez több információra is szükség van. Egyrészt karban kell tartani egy szimbólumtáblát, amely minden entitás esetén eltárolja, hogy milyen csomagokon és entitásokon keresztül jutottunk el hozzá (scope). Másrészt karban kell tartani a beimportált csomagokat és entitásokat is, hisz, ha lokálisan nem sikerült feloldani egy azonosítót, akkor a beimportált nevekkel kell folytatni a feloldást. Miután sikerült megoldani ezt a két feladatot, utána a névfeloldás algoritmusának elkészítése következett, mely a következőképpen működik: Minden alkalommal, amikor egy azonosítót fel kell oldani, az algoritmus először összegyűjti a lehetséges elemek halmazát, amelyek közötti keresni kell. Erre az összegyűjtésre azért van szükség, mert ha egy műveletben levő azonosítót kell feloldani, akkor azokat az entitásokat ki kell szűrni, amelyek egy művelet definíciójához tartoznak,

mivel enélkül a feloldás téves eredményt hozhat. Ezután megnézzük, hogy önmagában az azonosító létezik-e, mint entitás. Ha igen, akkor kész a feloldás, ha nem akkor folytatjuk a fa bejárásával. Az entitáshoz tartozó scope-ból kiindulva hozzáfűzzük az azonosítóhoz az őt tartalmazó entitás azonosítóját és megnézzük, hogy az előálló új azonosító létezik-e a keresendő entitások között. Ha igen, akkor a feloldás sikeres, ha nem akkor egyel feljebb lépünk a scope-ban és megismételjük a folyamatot. Ha egészen a gyökér azonosítóig eljutva nem sikerül a feloldás, akkor az importokkal folytatódik az algoritmus. Ebben az esetben minden az adott scope-ban elérhető import esetén megvizsgáljuk, hogy a beimportált azonosítót és az entitás azonosítóját összefűzve létezik-e a keresendő entitás. Ha igen, akkor a feloldás sikerült, ha nem akkor a kód hibás, amely végső soron megállítja a forráskód interpretálását, hiszen szintaktikai hiba keletkezett, mivel egy nem létező azonosítót akartunk használni.

A névfeloldás lezárásaként bemutatok két validációs lépést, amelyet a feldolgozás során elvégzek, annak érdekében, hogy bizonyos hibákra minél korábban fény derüljön: (i) az entitás azonosítók ütközése, ill. (ii) az attribútum azonosítók ütközése. Az első esetben minden alkalommal, amikor egy entitás bekerülne a map-be, először ellenőrzöm, hogy az adott „fully qualified name” szerepel-e az eddig összegyűjtött azonosítók között. Ha igen, akkor ismételten egy szintaktikai hiba történt, tehát az interpretálást meg kell állítani és jelezni a felhasználónak a hibát. A második esetben, amikor egy entitáshoz egy attribútumot adok hozzá, akkor ellenőrzöm, hogy az adott azonosító szerepel-e már az entitás attribútumai között. Ha igen, akkor ez egy újabb szintaktikai hiba, tehát az interpretálást ismételten le kell állítani.

4.2.5 Truffle AST feldolgozás

Az adatok feldolgozása után a következő lépés a folyamatban az egyes műveletekhez tartozó Truffle AST-k felépítése. Ehhez a műveletek definíciói alapján le kell generálni a Truffle csomópontokat, melyek az egyes utasításokat és kifejezéseket alkotják, majd ezekből össze kell állítani a művelethez tartozó végrehajtási fát.

Tekintsük meg a folyamatot egy példa műveleten keresztül:

```
operation Bool ID::ClassicBikeAlpha(ID instance){
   Object wheel1 =
       call $GetRelevantAttributeValue(instance, $ClassicBike.FrontWheel);
   Object wheel2 =
       call $GetRelevantAttributeValue(instance, $ClassicBike.RearWheel);

   if(wheel1==null || wheel2 == null) return true;

   return call $GetRelevantAttributeValue(wheel1, $Component.Size) ==
          call $GetRelevantAttributeValue(wheel2, $Component.Size);
}
```

16. ábra: Művelet feldolgozás példa

A feldolgozás során a művelet fejlécével indítjuk a fa építését, mely során először egy új Truffle specifikus adatszerkezet kerül létrehozásra. Ez az adatszerkezet a változókkal kapcsolatos adatokat fogja tárolni, továbbá ezzel párhuzamosan létrejön egy új lexikai hatókör is a szimbólumok adatainak eltárolásához és a hivatkozások feloldásához. Ezt követően megkezdődik a paraméterek feldolgozása. A példánk esetében ez egy darab **ID** típusú **instance** elnevezésű paramétert takar, melyhez egy paraméter olvasása csomópont fog létrejönni a fában. Ennek a csomópontnak a feladata, hogy az **instance** paraméterlistában elfoglalt indexe szerint a Truffle által átadott argumentumok közül kiolvassa a megfelelő értéket, majd beírja egy lokális **instance** elnevezésű változóba. Azaz a paraméterolvasás egy összetett művelet, mely egy olvasásból és egy lokális változó írásából áll.

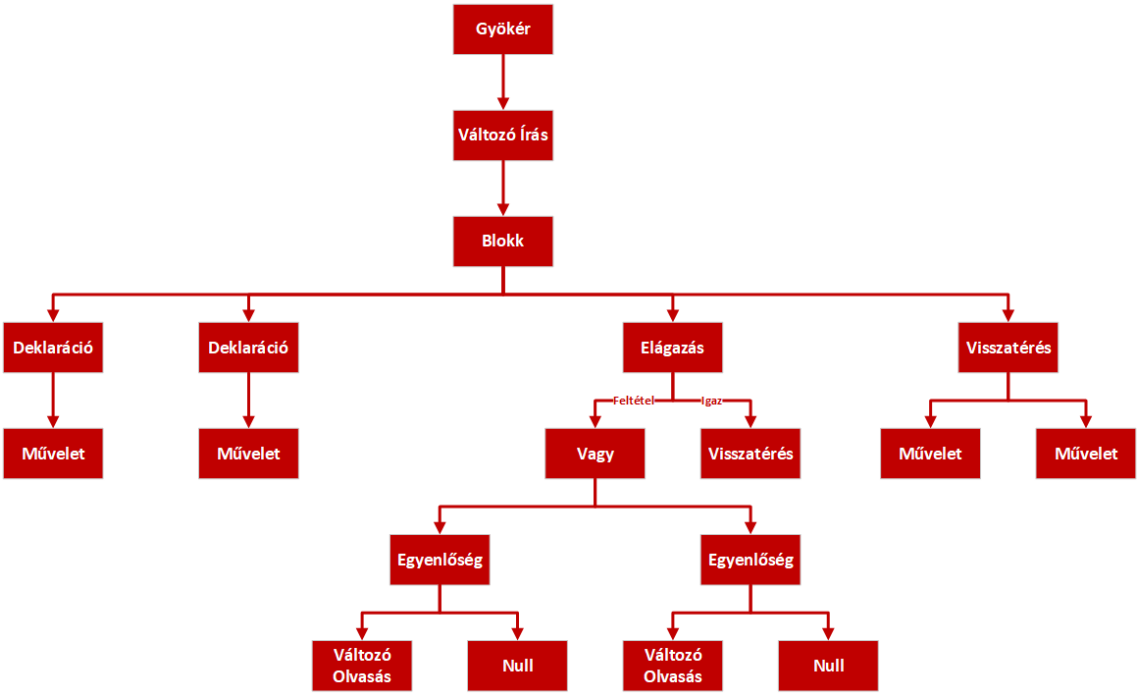
A paraméterek feldolgozása után következik a művelet tényleges definíciójának a feldolgozása. Mivel a nyelvtan alapján a definíció mindig egy blokkal kezdődik, ezért létrejön egy új blokk elem, amely egy újabb lexikai hatókört nyit. Felmerülhet, hogy a fejléc által nyitott hatókört és a blokk hatókörét egybe lehetne vonni. Azonban annak köszönhetően, hogy a redundancia nem okoz gondot, nincs szükség az eset speciális kezelésére, ami feleslegesen bonyolítaná a feldolgozást.

A példában a definíció feldolgozásakor az újonnan létrejött blokk által eltárolt utasításhoz fog hozzáadódni az összes feldolgozott utasítás vagy kifejezés. Ennek során először két változót deklarálunk melynek értéke, egy-egy művelet meghívásának visszatérési értéke lesz. Ezt követően feldolgozzuk az elágazást, melynek következtében az elágazás feltételéhez létrejön egy „vagy” kifejezés, ahol a bal és jobb oldal egy-egy két operandusú egyenlőséget vizsgáló operátor. Mindkét operátornak a bal operandusa egy változó olvasása, míg a jobb operandusa a „null” mint literál. Továbbá láthatjuk, hogy az elágazás egyetlen utasítást tartalmaz, így a blokkot megszokott módon elhagyhatjuk,

tehát az igaz ágon egyetlen „return” kifejezést találunk majd a fában, míg a hamis ág üres lesz.

Ezután a művelet utolsó kifejezése következik, ahol egy „return” -ben találhatunk egy egyenlőségvizsgálatot, melynek mindkét operandusa egy művelet meghívása. Miután a kifejezés feldolgozása megtörtént, a definíció blokkja befejeződik, így lezárásra kerül a hozzátartozó lexikai hatókör.

Végül lezárul a művelet, melyhez létrejön egy gyökércsomópont, amely tartalmazza az operációhoz szükséges adatokat és a teljes végrehajtási fát. Az így előálló fa a következőképpen néz ki:



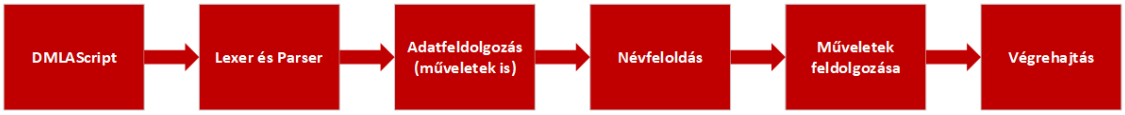
17. ábra: Példa végrehajtási fa

A fa végrehajtása fentről-lefelé és balról-jobbra fog megtörténni, ha eközben az egyik ágon történik egy visszatérés, akkor a művelet azonnal befejeződik a kívánt eredménnyel.

Az eddigieket összegezve tehát a DMLAScript feldolgozása két lépést takar. Mindkét lépés az Xtext-ből ANTLR-be átírt nyelvtan segítségével generált lexert és parsert használja. Első lépésben a csak adatot reprezentáló entitásokat és a műveleteket, mint adat összegyűjtöm egy map-be, hogy az azonosítón keresztül minél gyorsabb hozzáférést biztosítsak. Eközben ügyelek arra, hogy validáljam az entitásokat és attribútumokat a duplikációk ellen. Ezután lefut a névfeloldás, amely egyrészt ellenőrzi a

hivatkozások helyességét, másrészt ahol szükséges fel is oldja őket. A második lépésben a névfeloldás eredményét felhasználva felépítem a műveletekhez tartozó Truffle AST-t, melyekhez egyenként biztosítok egy a végrehajtáshoz szükséges belépési pontot.

Az így előálló folyamatot az alábbi ábrán tekinthetjük meg:



18. ábra: Az 11. ábra alapján előálló folyamat

4.2.6 Eredmény

Az adat és műveletfeldolgozás befejezését követően az első fázisban elért eredmények tesztelése és ellenőrzése következett, mely ugyancsak két lépésben történt. Először meg kellett bizonyosodnom arról, hogy az adatfeldolgozás eredménye, azaz az összeállított map teljes egészében megegyezik a korábbi verzió által előállított adatstruktúrával. Ehhez egységteszteket készítettem, amik beolvasták a korábbi DMLA által előállított XML fájlokat és a bennük található entitásokat összehasonlították a map-ben található entitás objektumokkal. Az összehasonlításba beletartozik az entitás attribútumainak és értékeinek ellenőrzése, melyek esetén külön figyelni kell arra is, hogy az entitás csak és kizárólag a megadott elemeket tartalmazza, se többet, se kevesebbet.

Az egységtesztek futtatásához egyrészt a Bootstrap-et használtuk fel, mivel a Bootstrap minden olyan nyelvi struktúrát tartalmaz amiből entitás keletkezhet, másrészt a biztonság kedvéért a meglévő szakterületi modellekre is lefuttattuk. A megtalált hibákat javítva végül megbizonyosodtunk róla, hogy sikerült teljes egészében lemásolni a régi adatszerkezetet, így következhetett a futtatás tesztelése.

A futtatás teszteléséhez a komplexitásából adódóan az egységtesztek helyett az ellenőrzést a meglevő szakterületi modelljeink és a Bootstrap futtatására bíztuk. Ennek keretében sikerült számos hibára fényt deríteni, köszönhetően a forráskód sokszínűségének, így ezen hibák kijavítását követően több más szakterületi modell beolvasása és validálása sem mutatott ki problémát a nyelvi elemek implementációjában. Ezzel kapcsolatban fontos megemlíteni, hogy bár hibába nem futottunk bele, de az eredeti Xtext alapú megoldáshoz képest találtunk néhány eltérést a két rendszer között. Ezeket az eltéréseket a közös kutatócsoport megbeszéléseken vagy javítottuk, vagy átértékeltük és nem vettük őket hibának.

Miután megbizonyosodtunk róla, hogy az adatszerkezet építése és a futtatás is a vártaknak megfelelően működik, következhetett a teljesítményteszt. Elmondhatjuk, hogy az új DMLA fejlesztésével kapcsolatban az egyik legfontosabb elvárásunk az volt, hogy a fordításalapú megközelítésnél jobb teljesítményt érjünk el. Hiszen a megfelelő futási idő nélkül nem lehetne ipari környezetben felhasználni a keretrendszert.

A teljesítményteszthez a Bootstrap-et és a legnagyobb modellünket a Bicycle modellt használtuk. Együttesen a két modell több mint tízezer entitást tartalmaz, mely mind a feldolgozás, mind a futtatás szempontjából megfelelően sok elemnek számít ahhoz, hogy összemérjük a két rendszert.

Az eredeti fordításalapú megközelítés esetén a teljes feldolgozási és validációs folyamat átlagosan hét- és nyolcezer milliszekundum közötti eredményeket produkált, melynek jelentős részét a fordítási folyamat tette ki. Ezzel szemben az új interpreter alapú futtatása átlagosan háromezer és négyezer milliszekundum alatt végzett a teljes folyamattal, melynek a feldolgozás csak a töredéke volt, körülbelül nyolcszáz és ezer milliszekundum között. Ezen tapasztalatok alapján jól látható, hogy a DMLA új verziója mind a feldolgozás, mind a futtatás tekintetében szignifikánsan jobb eredményeket ért el úgy, hogy emellett számos új szolgáltatás bevezetését teszi lehetővé. Emellett fontos megjegyezni azt is, hogy a GraalVM és Truffle több optimalizációs lehetőséget nyújt, melyeknek eddig csak a töredékét használtuk, így a jövőben további javulás várható a teljesítmény tekintetében.

Összességében tehát az első fázist minden tekintetben sikerrel zártuk, így folytathattuk a munkát a második fázissal, azaz a dinamizmus megvalósításával.

4.3 Második fázis

Ahogy korábban elhangzott, az eredeti Xtext alapú DMLA esetén magukat az entitásokat futásidőben se törölni, se módosítani nem lehetett, azaz a felhasználó minden esetben csak egy pillanatfelvételt látott a modellből. Ahhoz, hogy a modell állapotát módosítani tudja, először módosítania kellett a DMLAScript-et, majd a teljes projektet a Bootstrap-el és a szakterületi modellekkel együtt újra le kellett futtatnia.

Az új DMLA első verziója esetén a helyzet kis mértékben javult azáltal, hogy szelektálni lehetett a szakterületi modellek között, így mindig csak az fordult le és került validálásra, amiket a futtatás során megadtunk. Ahhoz, hogy a korlátokat átlépjük, és egy

futásidőben módosítható és validálható keretrendszert készítsünk, két dologra van szükség: a (i) Bootstrap és a szakterületi modellek futtatásának szétválasztására és (ii) a valósidejű parancsértelmező megvalósítására. Mindkét probléma számos érdekes kérdést vetett fel, így ennek a fejezetnek a célja ezen kérdések és a rájuk adott válaszok bemutatása.

4.3.1 Bootstrap és szakterületi modell szeparáció

A dinamizmus felé az első lépés a Bootstrap és a kiválasztott szakterületi modell feldolgozásának és validálásának szétválasztása volt. Annak ellenére, hogy ennek a problémának a megoldása nem okozott nehézségeket, néhány érdekes kérdést mégis felvetett.

A megoldáshoz először néhány apróbb változtatást végeztem. Első lépésben átalakítottam a GraalVM indítását olyan formában, hogy csak a Bootstrap-et dolgozza fel és validálja. Ezt követően, immáron a GraalVM futása közben, bekértem a szakterületi modellt leíró DMLAScript-et, amelyet az ismert folyamaton keresztül feldolgoztam és validáltam. A feldolgozás sikertelennek bizonyult, ugyanis a névfeloldás során a forráskódban szereplő Bootstrap hivatkozásokat nem sikerült feloldani. Ez annak köszönhető, hogy eredetileg az adatfeldolgozás egyetlen nagy DMLAScript fájlba fűzte össze a Bootstrap-et és a szakterületi modellt, amelyet így a névfeloldás egyben képes volt feldolgozni. A névfeloldás azonban állapotmentes abban az értelemben, hogy két forrásfájl feldolgozása között nem tartja meg a feloldáshoz használt entitásokat. A probléma megoldásához tehát a már korábban feldolgozott entitásokat is hozzá kellett vennem a névfeloldás algoritmusához, majd ezt követően már sikeresen lefutott mind a névfeloldás, mind a validáció.

Bár a szeparáció sikeres volt, azonban a teljesítmény szempontjából felfigyeltünk egy anomáliára. A Bootstrap és a szakterületi modell külön-külön feldolgozása szignifikánsan több időt vett igénybe, mint egyben. A probléma okának kiderítésére a kód instrumentálása felé fordultam, azonban gyorsan kiderült, hogy egyrészt a GraalVM instrumentálása harmadik féltől származó eszközökkel meglehetősen nehézkes, másrészt a validáció számításigénye túl nagy ahhoz, hogy az instrumentálás használható eredményt hozzon. Az Oracle Labs gondolt erre a problémára és lehetőséget biztosítanak saját, beépített instrumentálási eszköz implementálására, melyet meg is valósítottam. Sajnos az eredmények nem voltak elég meggyőzőek ahhoz, hogy biztos következtetéseket tudjunk

levonni a lassulást illetően, így ez irányban további kutatásokat kell még végeznünk. Az anomália feloldásán jelenleg is dolgozom.

Az anomália részleges elkerülése érdekében a szeparációt követően jelentősebb optimalizációt végeztem a validáción. Mivel a Bootstrap entitásait maximum a GraalVM elindulásakor kell validálni, így a szakterületi modellek esetén ezeket az entitásokat ki lehet hagyni. A Bootstrap mérete miatt (kb. hétezer entitás), ez szignifikáns gyorsulást eredményez. A megoldáshoz csak arra volt szükség, hogy az adatfeldolgozás során a Bootstrap entitásait megjelöljem, ezáltal a validáció során egy paraméter segítségével el lehet dönteni, hogy szeretnénk-e a Bootstrap-et is validálni. Az implementációt követően a mérések igazolták az elképzelést, hiszen a szakterületi modellek validálása a modell méretének függvényében többször gyorsabb is lehet a Bootstrap nélkül.

4.3.2 Parancsértelmező

Miután sikerült szeparálni a Bootstrap-et és a szakterületi modelleket, megkezdtem a munkát a parancsértelmező megvalósításán. Az eredeti elképzelésünk szerint az értelmező is DMLAScript-tel dolgozott volna, azonban néhány megbeszélést követően gyorsan kiderült, hogy a DMLAScript alkalmatlan egy ilyen célkitűzéssel rendelkező parancssoros nyelvnek. Az alkalmatlanság forrása kettős: (i) a nyelv nem támogatja a csomagon kívül definiált műveleteket, (ii) a DMLAScript mindig tuple-lé fordul.

Az első probléma azt jelenti, hogy a DMLAScript esetén egy műveletet nem lehet önmagában meghívni, mivel a nyelvtan alapján minden entitásnak és műveletnek egy csomagon belül kell szerepelnie. Tehát a parancsértelmező esetén minden művelet meghívásakor a felhasználónak a parancsot vagy manuálisan egy csomagban kellett volna elhelyeznie vagy a feldolgozás során kellett volna injektálni a műveletet egy csomagba.

A második probléma más jellegű, itt valójában az a nehézség, hogy a DMLAScriptet úgy alkotta meg a kutatócsoport, hogy a leírtakból minden esetben tuple-ök készülnek. Ez azt jelenti, hogy egy-egy műveletet egy, vagy több tuple ír le, amiket adatként el kell helyeznünk a meglévő entitások közé. Azonban, ha a parancsértelmező parancsai is hasonló módon kerülnének feldolgozásra, akkor bekerülnének az entitások közé, ami validációs hibákat eredményezne annak ellenére, hogy maga a modell helyes.

A fenti két probléma hatására úgy döntöttem, hogy egy nagyon egyszerű parancssoros nyelvet készítek, amely teljesen elkülönül a DMLAScript-től, azaz

lényegében csak a VM szintjén fog működni. Ez a megközelítés az imént felsorolt problémák megoldása mellett számos más előnnyel is jár. Egyrészt lehetőségünk van változók használatára vagy tetszőleges függvény megvalósítására, másrészt a meglévő interpreter implementációt egyáltalán nem kell módosítanunk, így a két rendszer független marad egymástól. Ezen felül fontos megjegyezni, hogy a VM ilyen formában történő vezérlése nem mond ellent az alapkoncepciónak, hiszen maga az ASM is lehetővé teszi a külső beavatkozást.

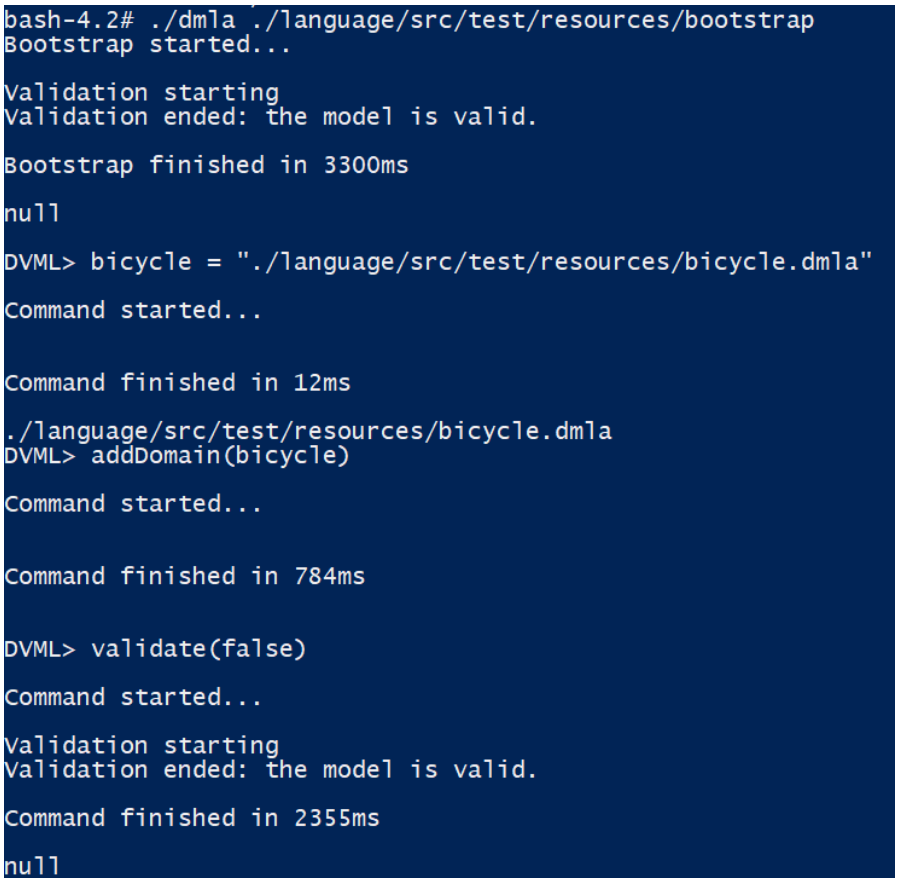
Az új nyelv megvalósításához ANTLR-t használtam. A megvalósításban a lexer és a parser két fontosabb struktúrát képes kezelni: (i) vagy függvényeket hívhatunk a parancssorban (ii) vagy változókat deklarálhatunk, amelyek egy globális scope-ba kerülnek és nem a meglévő entitások közé. A függvények és változók teljesen el vannak szeparálva a DMLAScript műveleteitől, hiszen a parancs végrehajtását elindító egyetlen csomópont kivételével még a végrehajtási fában sem vesznek részt.

A nyelv elkészülését követően a szükséges függvények megvalósítása következett. A kezdeti kitűzött cél a validáció mellett az entitások hozzáadása és törlése volt, illetve a kezelhetőség növelése érdekében szükség volt aktív csomag és entitás megjelölésére is, így egyszerűsítve a parancsok használatát. Az így elkészült függvények a következők:

- Validálás Bootstrap-el vagy anélkül: validate(Bool)
- Domain fájl beolvasása: addDomain(String)
- Aktív csomag beállítása: setActivePackage(String)
- Aktív csomag lekérdezése: getActivePackage(): String
- Aktív entitás beállítása: setActiveEntity(String)
- Aktív entitás lekérdezése: getActiveEntity(): String
- Entitás hozzáadása aktív csomaghoz: addEntity(String, String)
- Attribútum hozzáadása aktív entitáshoz: addAttribute(String)
- Slot hozzáadása aktív entitáshoz: addSlot(String, String)
- Megszorítás hozzáadása aktív entitáshoz: addConstraint(String, String)
- Érték hozzáadása aktív entitáshoz: addValue(<tetszőleges típus>)

- Entitás törlése: deleteEntity(String)
- Minden szakterületi entitás törlése: purgeDomains()

A hozzáadás megvalósítása nem érdemel különösebb magyarázatot, hiszen ebben az esetben csak azonosítókat kell megadnunk a megfelelő függvényeknek, amelyek létrehozzák a kívánt entitást. Ezzel szemben a törlés több kérdést is felvetett, melyek közül a legfontosabb a törlés szabadsága. Azaz meg szeretnénk-e engedni azt, hogy a felhasználó tetszőleges entitást töröljön a modellből? Hiszen, ha megengedjük, hogy a felhasználó egy olyan entitást töröljön, amire több más entitás hivatkozik, akkor a validáció azonnal hamis eredményt fog adni. Végül arra jutottunk, hogy kezdetben csak a „biztonságos” törlést támogatjuk, azaz csak olyan entitást lehet törölni, amire senki sem hivatkozik. Ennek megvalósításához azonban szükség van a hivatkozások detektálására, amely a mi esetünkben egy referenciaszámlálót jelent minden entitást esetén. Ezen felül egy entitás törlése esetén nem elég magának az entitásnak a referenciaszámlálóját vizsgálni, hanem minden attribútumát is ellenőrizni kell, amelyek lényegében további entitásokat jelentenek. Végső soron tehát egy entitás törlésekor rekurzívan ellenőrizni kell az entitással kapcsolatos minden más entitás referenciaszámlálóját, és ha mindegyik nulla, akkor az entitás az összes többi entitásával együtt törölhető. Az elkészült nyelv használatára a 19. ábra mutat példát, melyben a GraalVM elindítása mellett láthatjuk egy változó létrehozását (**bicycle**), illetve ezen a változón keresztül egy domain hozzáadását és validálását.



19. ábra: DMLA parancsértelmező

4.4 Összehasonlítás

Hasonlítsuk össze régi (fordításalapú) és az új (interpreter alapú) DMLA esetén egy szakterület létrehozásának folyamatát! Minden szakterület esetén, az első lépés a szakterület definíciójának az összeállítása. A régi DMLA-ban ezt a lépést egyetlen módon tudtuk megtenni: a DMLAScript segítségével megírtuk a szakterületi nyelvet reprezentáló (meta)modellt. Az új DMLA esetén kétféle módon hozhatjuk létre a szakterületi nyelvet: (i) a DMLAScript megírásával, vagy (ii) a parancssoros interfészen keresztül. Ha a második megoldást választjuk, akkor arra is lehetőségünk van, hogy a nyelv elkészítése közben folyamatosan validáljuk az addigi haladásunkat a megfelelő parancs segítségével, így mindig biztosak lehetünk benne, hogy a modell addig a pontig kidolgozott része konzisztens és érvényes állapotban van. Bár a régi megközelítés esetén is lehetőség lenne a modell folyamatos validálására, azonban ez a felhasználó szempontjából sokkal kényelmetlenebb lenne, hiszen a fordításalapú megvalósítás a folyamat lassúsága miatt nem lenne elég gördülékeny. A régi módszerbe implicit módon bele lett kódolva, hogy a nyelvek és modellek létrehozása történik meg először és csak utána validálunk, amikor már mindennel készen vagyunk. Ez a forgatókönyv az ipari

környezetben nem életszerű. Az interpreter alapú megoldás dinamikussága és interaktivitása az, ami ezt a problémát áthidalhatóvá teszi. Ráadásul, amennyiben a felhasználó az új megvalósítás esetén is a DMLAScript alapú nyelvdefiníciót választaná, akkor a Bootstrap és a szakterületi modellek szeparációja miatt továbbra is lehetősége lenne a modell gyors tesztelésére és validálására.

Az elkészült szakterületi nyelv esetén a régi DMLA-ban nem volt lehetőség a szakterületi környezet publikálásra, mivel a futtatáshoz a teljes DMLA forráskódra és fejlesztőkörnyezetre szükség volt. Az új DMLA esetén ez a korlát megszűnt, mivel a virtuális gép miatt könnyedén lehet az implementációt akár egy Docker konténer keretében publikálni. A keretrendszert akár a felhőbe is feltölthetjük, így egy böngészőn segítségével hálózaton keresztül bármikor, bárhonnan elérhetjük, és a modelleket, szakterületeket webes környezetben szerkeszthetjük. A kutatócsoportban jelenleg is folyik ilyen irányú munka, a webes szerkesztő már elkészült [23], de egyelőre a parancssori interfész még nincs kivezetve erre a felületre.

Az új megközelítés azonban nem csak a nyelv létrehozásakor, hanem a használatakor is nagy segítséget nyújt. Megjegyzendő, hogy a többrétegű modellezés környezetében ez a két fázis nem feltétlenül különül el egymástól élesen, de ha külön szerepkört rendelünk azokhoz, akik a nyelvet definiálják és akik a konkrét példányokat létrehozzák később (pl. egy ipari gyártósor kapcsán), akkor az új megvalósítás a részleges, szakterülethez kötődő validációja révén itt is jelentősen megkönnyíti és meggyorsítja a munkát arról nem is szólva, hogy a parancssori interfész lehetőséget ad arra is, hogy a jövőben egységteszteket lehessen írni, amik ezen az interfészeken keresztül változtatják meg a modellt, majd végzik el annak validációját. Ez a régi megközelítésben nagyon nehézkesen, a DMLAScript fájlok generálásával lett volna csak elérhető.

5 Összefoglalás és a további tervek

A többrétegű metamodellezés egy új, ígéretes irányvonalat képvisel a modellalapú fejlesztés területén, melyre az iparban egyre nagyobb igény mutatkozik. Ennek az új metamodellezési irányvonalnak a képviselője a DMLA, melynek egyik fontos célja az napjainkban felmerülő ipari igényeknek a kielégítése. A DMLA korábbi, fordításalapú megoldása nem volt elég rugalmas a többrétegű metamodellezés által nyújtott lehetőségek megvalósításához, mely legfőképpen a modellek dinamikus létrehozásának, módosításának, törlésének és validálásának hiányában volt keresendő. Szükség volt egy olyan új megoldás létrehozására, mely amellett, hogy követi a korábbi DMLA által lefektetett alapkoncepciókat, képes a modellek dinamikus kezelésére és validálására is.

Ebben a dolgozatban ennek az új megoldásnak a kidolgozásával kapcsolatos munkámat mutattam be. A munkám során elsőként megismerkedtem a DMLA működésével, mind elméleti, mind gyakorlati szinten. Második lépésként kutatást végeztem a megvalósításához szükséges technológiákkal, a GraalVM és a Truffle eszközökkel. Harmadik lépésként részleteiben is megismertem a DMLA programozási nyelvével, a DMLAScript-tel, mely a szakterületi modellek létrehozásában és a validációs logika elkészítésében nyújt segítséget. Ezt követően, immáron aktív alkotói munkára áttérve, megterveztem és megvalósítottam a modellek GraalVM-ben történő adatfeldolgozásához és tárolásához szükséges adatszerkezeteket és módszereket, majd elkészítettem a DMLAScript műveleti nyelvét végrehajtani képes interpretert a Truffle segítségével. Miután tesztek segítségével megbizonyosodtam arról, hogy az új implementáció teljes mértékben megegyezik a fordításalapú DMLA működésével és teljesítmény tekintetében is megfelelő eredményeket nyújt, megkezdtem a modellek dinamikus kezeléséhez szükséges rendszerek megalkotását. Ennek keretében létrehoztam egy új parancsértelmező nyelvet, mely a DMLAScript-től szeparáltan, a VM szintjén képes a megadott parancsok értelmezésére és ezáltal a modellek módosítására és validációjára. A nyelvben lehetőségünk van változók és saját segédfüggvények használatára is.

Annak ellenére, hogy az elkészült új DMLA keretrendszer teljes mértékben alkalmas szakterületi modellek validált módon történő létrehozására és manipulálására, még hosszú út áll az keretrendszer előtt, mielőtt ipari környezetben lehetne alkalmazni.

A jelenlegi implementáció tekintetében számos lehetőségünk van a továbbfejlesztésre. A GraalVM és Truffle szempontjából saját debugger megvalósításával lehetőséget szeretnék biztosítani a szakterületi modellezők számára a validáció logika hibáinak egyszerűbb javítására. A Truffle lehetőségeit és a DMLA sajátosságai kihasználva különféle optimalizációs technikák segítségével szeretnék javítani a futási időn. Emellett a DMLAScript szintaktikájának javításával szeretném elérni, hogy egyszerűbb legyen a szakterületi modellek elkészítése, illetve a validáció implementációja. A parancssoros értelmezőt is tervezem további kényelmi funkciókkal bővíteni az egyszerűbb használat érdekében. Mindezen felül fontos kutatási és fejlesztési irányvonal a modellek inkrementális validálása, mely tovább gyorsítaná a futtatást. Végül az ipari felhasználás szempontjából az egyik legfontosabb irányvonal a vizuális szerkesztő megvalósítása, ami ahhoz fontos, hogy a DMLA egy teljesértékű modellező keretrendszerré váljon.

6 Köszönetnyilvánítás

A kutatás az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg (EFOP-3.6.2-16-2017-00013, Innovatív Informatikai és Infokommunikációs Megoldásokat Megalapozó Tematikus Kutatási Együttműködések).

7 Irodalomjegyzék

[1] „OMG MetaObject Facility,” 2005. [Online]. Available: http://www.omg.org/mof/,. [Hozzáférés dátuma: 27 október 2019].

[2] C. Atkinson és T. Kühne, „Reducing accidental complexity in domain models,” *Software and Systems Modeling*, %1. kötet7, pp. 345-359, 2008.

[3] C. Atkinson és T. Kühne, „The Essence of Multilevel Metamodeling,” in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, Berlin, Heidelberg, Springer-Verlag, 2001, pp. 19-33.

[4] Z. Theisz, S. Bácsi, G. Mezei, F. A. Somogyi és D. Palatinszky, „By multi-layer to multi-level modeling,” 2019. [Online]. Available: https://www.aut.bme.hu/Upload/Pages/Research/VMTS/DMLA/Multi2019_TNumber.pdf. [Hozzáférés dátuma: 27 október 2019].

[5] D. Automation és A. Informatics, „DMLA Homepage,” [Online]. Available: https://www.aut.bme.hu/Pages/Research/VMTS/DMLA. [Hozzáférés dátuma: 27 október 2019].

[6] Z. Theisz, D. Urbán és G. Mezei, „Constraint Modularization Within Multi-level Meta-modeling,” in *Information and Software Technologies - 23rd International Conference, ICIST 2017, Druskininkai, Lithuania, October 12-14, 2017, Proceedings*, 2017.

[7] D. Urbán, G. Mezei és Z. Theisz, „Formalism for Static Aspects of Dynamic Metamodeling,” *Periodica Polytechnica Electrical Engineering and Computer Science*, %1. kötet61, pp. 34-47, 2017.

[8] D. Urbán, Z. Theisz és G. Mezei, „Self-describing Operations for Multi-level Meta-modeling,” in *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018.*, 2018.

[9] B. Egon és S. Robert, Abstract State Machines: A Method for High-Level System Design and Analysis, Springer-Verlag, 2003.

[10] G. Mezei, Z. Theisz, D. Urbán, S. Bácsi, F. A. Somogyi és D. Palatinszky, „A bootstrap for self-describing, self-validating multi-layer metamodeling,” *Automation and Applied Computer Science Workshop,* 2019.

[11] „GraalVM,” Oracle Labs, [Online]. Available: https://www.graalvm.org/. [Hozzáférés dátuma: 27 október 2019].

[12] Oracle, „Truffle GitHub,” [Online]. Available: https://github.com/oracle/graal/tree/master/truffle. [Hozzáférés dátuma: 27 október 2019].

[13] „Faster Ruby, JS and Other Languages Using Graal and Truffle,” [Online]. Available: https://www.infoq.com/presentations/graal-trufle/. [Hozzáférés dátuma: 27 október 2019].

[14] „ANTLR,” [Online]. Available: https://www.antlr.org/. [Hozzáférés dátuma: 27 10 2019].

[15] Oracle, „FastR,” [Online]. Available: https://github.com/oracle/fastr. [Hozzáférés dátuma: 27 október 2019].

[16] „The R Project for Statistical Computing,” [Online]. Available: https://www.r-project.org/. [Hozzáférés dátuma: 27 október 2019].

[17] Oracle, „Truffle Ruby,” [Online]. Available: https://github.com/oracle/truffleruby. [Hozzáférés dátuma: 27 október 2019].

[18] „Ruby,” [Online]. Available: https://www.ruby-lang.org/en/. [Hozzáférés dátuma: 27 október 2019].

[19] „Graal projects,” [Online]. Available: https://github.com/neomatrix369/awesome-graal. [Hozzáférés dátuma: 27 október 2019].

[20] Oracle, „Twitter’s Quest for a Wholly Graal Runtime,” [Online]. Available: https://www.youtube.com/watch?v=G-vlQaPMAxg. [Hozzáférés dátuma: 27 október 2019].

[21] „Simple Language,” [Online]. Available: https://github.com/graalvm/simplelanguage. [Hozzáférés dátuma: 27 október 2019].

[22] „Xtext,” [Online]. Available: https://www.eclipse.org/Xtext/. [Hozzáférés dátuma: 27 október 2019].

[23] „DMLA Web Editor Demo,” [Online]. Available: http://ec2-3-15-199-199.us-east-2.compute.amazonaws.com:8080/hu.ud.dmla.web-1.0.0-SNAPSHOT/. [Hozzáférés dátuma: 27 október 2019].