IEGULDĪJUMS TAVĀ NĀKOTNĒ!

Eiropas Reģionālās attīstības fonds

Prioritāte: 2.1. Zinātne un inovācijas

Pasākums: 2.1.1. Zinātne, pētniecība un attīstība

Aktivitāte: 2.1.1.1. Atbalsts zinātnei un pētniecībai

Projekts: ”Multi - modeļu izstrādes tehnoloģija .NET pielietojumu projektiem”

Projekta sākuma datums: 2014.gada 1.janvāris.

Projekta beigu datums: 2015.gada 30.jūnijs.

Līguma Nr. 2013/0031/2DP/2.1.1.1.0/13/APIA/VIAA/010

ESF finansējuma saņēmējs: SIA, SWH SETS

Sadarbības partneris: Elektronikas un datorzinātņu institūts (EDI)

Projekta aktivitātes Nr.1.3.2 "Modeļa transformāciju atbalsta izstrāde (M2M transformāciju atbalsts)" progresa pārskats

Pārskats Nr.19. par periodu no 2014.gada 1.jūlija līdz 2014.gada 31.decembrim.

SATURS

1. Kopsavilkums

Pārskata periodā (2014-07-01 – 2014-12-31.) projekta „Multi - modeļu izstrādes tehnoloģija .NET pielietojumu projektiem" aktivitātes Nr.1.3.2 "Modeļa transformāciju atbalsta izstrāde (M2M transformāciju atbalsts)" ietvaros ir turpināts pirmā pārskata periodā iesāktais un veikti šādi darbi:

1. Veiktas projektējuma izmaiņas.
2. Transformācijas elementu izstrāde, tai skaitā:
    a. Clause izstrāde,
    b. ClauseVirtual izstrāde,
    c. Clause_list izstrāde,
    d. Clause_NonLoop izstrāde,
    e. Clause_any izstrāde,
    f. Clause_external izstrāde,
    g. Clause_mirror izstrāde,
    h. Clause_mirrorSet izstrāde,
    i. Clause_pushD izstrāde,
    j. Clause_varGet izstrāde,
    k. Clause_varSet izstrāde,
    l. Clause_callNonLoop izstrāde,
    m. Clause_create izstrāde.
3. Transformācijas valodas izstrāde.
4. Apstaigājošās loģikas izstrāde.
5. Transformācijas valodas analizatora izstrāde.
6. Aktivitātes pētnieciskā darbība apspriesta ik nedēļas projekta semināros.

2. Ievads

Šis pārskats ir veltīts projekta apakšaktivitātes Nr.1.3.2 "Modeļa transformāciju atbalsta izstrāde (M2M transformāciju atbalsts)" ietvaros veiktajiem pētījumiem.

Aktivitātes ietvaros tika izstrādāta transformācijas elementārā loģika, kas aprakstīta sadaļā 3. Transformācijas elementārā loģika, un saturīgie elementi (sadaļā 4. Saturīgie elementi).

Papildus tika izstrādāta transformāciju valoda, kuras apraksts ir izklāstīts sadaļā 5. Transformāciju valoda.

3. Transformācijas elementārā loģika

MEDUS projektā M2M transformāciju veikšanas pamats ir transformācijas elements *ClauseVirtual*. No vienas puses vispārīgā gadījumā gribētos iespēju veikt jebkuru transformāciju, kas patiesībā nozīmē patvaļīgu programmu, kura darbojas vairāku metamodeļu vidē. Savukārt, no otras puses, tikai ierobežojot vispārīgās iespējas, ir iespējams iegūt automatizāciju. Lai saglabātu iespējas veidot ļoti dažāda veida transformācijas elementus, *ClauseVirtual* sevī satur tikai sekojošo loģiku:

- elementa konteksts (šajā klašu hierarhijas līmenī netiek precizēts) satur visu transformēšanai nepieciešamo informāciju
- transformēšana tiek veikta virtuālajā metodē `public virtual void doJob()`, kuras pārdefinējumos iespējams noprogrammēt patvaļīgu loģiku.

Paralēli acīmredzamajai pilnīgi "manuālai" apstrādei tiek piedāvāta "apstaigājošā" loģika, kas balstīta uz [1] izklāstītajām modeļu transformācijas idejām (konstrukcija dod iespēju brīvi piedefinēt arī citas "loģikas").

3.1. "Apstaigājošā" loģika

Tipiska transformācija satur darbības, kas jāpiemēro viena tipa objektiem - bieži nākas lietot cikla operatoru pa atbilstošo kopu (C# tas būtu *foreach* operators). To varētu realizēt, atklāti rakstot cikla operatorus. Alternatīva ir loģiskajā programmēšanā (Prolog) iebūvētais apstaigāšanas (backtracking) mehānisms. Šeit tiks lietota analoģiska pieeja.

Vispārīgā apstaigāšanas loģika tiks realizēta klasē *Clause* (apakšklase no *ClauseVirtual*). Realizācijas elementi:

- `Clause next` elementi ir sasieti virknītē - katrs rāda uz nākošo (neobligāts) elementu.
- `public sealed override void doJob()` šeit ir "apstaigājošās" loģikas realizācija, ko specifiskos transformācijas elementos vairs nebūs iespējams mainīt.
    1. Jebkura elementa izpilde vienmēr beidzas ar loģisko vērtību, kas norāda, vai izpilde ir bijusi veiksmīga. Ja šī elementa izpilde beidzas veiksmīgi, tad tiek iedarbināts nākošais elements (ja izpilde nav veiksmīga, tad šajā apstrādes zarā nākošais elements netiek iedarbināts), ja tāds eksistē. Ja izpilde ir bijusi neveiksmīga, vadība tiek atgriezta izsaucošajai videi.
    2. Pēc nākošā(o) elementa(u) izpildes (vai neizpildes) vadība nonāk atpakaļ aplūkojamajā elementā.
    3. Elementu izpilda vēlreiz un loģiski atgriežamies algoritma 1. solī. Tehniski elementa izpilde tiek realizēta virtuālajās metodēs (to saturs atkarīgs no saturīgās darbības)
        - `protected virtual bool performFirst()` šī metode tiek izsaukta, tikai pirmo reizi izpildot elementa darbību (solis 1) un, parasti, pirms saturīgās darbības satur vides sagatavošanu (metode `protected virtual bool init()`, kas var beigties neveiksmīgi, līdz

ar to terminējot visu elementa izpildi). "Pirmā reize" attiecas uz šī elementa izpildi vienā zarā (nākošajā apstrādes zarā šī metode atkal tiks izsaukta).

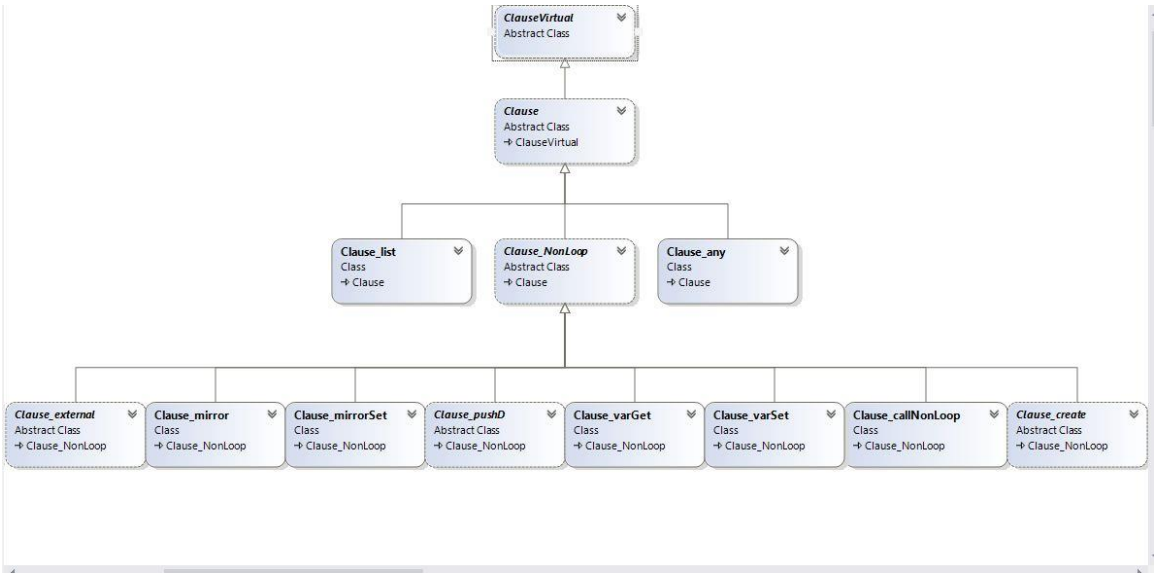
- `protected virtual bool performNext()` šī metode tiek izsaukta (solis 3), atkārtoti (vienā zarā) izpildot elementu.

Visi saturīgie elementi (tiks aprakstīti nākošajā nodaļā) ir klases *Clause* apakšklasēs un realizē individuālo loģiku.

Samērā daudziem saturīgajiem elementiem patiesībā iekšējā atkārtotā izpilde nav nepieciešama. Tādēļ izveidota klase *Clause_NonLoop* (apakšklase no *Clause*), kurā metode *performNext* neko saturīgu nedara, bet uzreiz atgriež vērtību *false*, kas nozīmē to, ka saturīgi elements tiek izpildīts vienu reizi katrā apstrādes zarā.

4. Saturīgie elementi

Saturīgie elementi ir ar samērā šauru loģiku, un tas ir transformācijas būvētāja uzdevums izveidot saturīgu darbību virkni (pilnīga analoģija ar programmēšanas valodas operatoriem un konkrētu programmu). Zīmējumā zemāk attēlota transformācijas elementu (kā klašu) hierarhija.



Zīmējums 1. Transformācijas elementu hierarhija.

4.1. Clause_list

Šis elements tiek izmantots, lai organizētu apstrādi objektu kolekcijām. Metode *init* tiek izmantota, lai aizpildītu objektu kolekciju. *perform*-metodēs tiek izvēlēts kārtējais elements no kolekcijas un uzstādīts kā ***tekošais elements*** izpildes vidē.

Tipiski šī klase būtu lietojama kā abstrakta, un atbilstoši kolekcijas veidošanai (metode *init*) būtu jāveido apakšklase.

4.2. Clause_varSet

Šis elements neveido ciklu un tiek izmantots transformācijas mainīgo kontekstā. Transformācijas elementu virknes izpildes laikā tiek uzturēti ***transformācijas mainīgie*** - katrs mainīgais identificējas ar unikālu vārdu. Šis elements izveido transformācijas mainīgo ar norādīto vārdu (`public string varName`) un piešķir tam *tekošo elementu*. Ja mainīgais ar norādīto vārdu jau eksistē, tad izpildes rezultāts ir nedefinēts.

Atgriežoties no elementa izpildes (atkāpšanās solis), transformācijas mainīgais tiek likvidēts.

4.3. Clause_varGet

Šis ir *transformācijas mainīgo* izmantošanas elements (neveido ciklu) - no konteksta pēc vārda (`public string varName`) tiek paņemta vērtība un piešķirta *tekošajam elementam*.

4.4. Clause_pushD

Transformācijas elementu virknes izpildes laikā tiek nodrošināts ***datu steks*** (samērā pilna analoģija ar tipiskajām programmēšanas valodām). Šis elements (nav ciklisks) *datu stekā* ievieto noradīto (`UniversalObject`) objektu. Atgriežoties no elementa izpildes (atkāpšanās solis), ievietotais steka elements tiek likvidēts.

4.5. Clause_create

Šis elements paredzēts jaunu modeļa objektu izveidošanai (nav ciklisks). *init* darbībā tiek izveidots jauns objekts (patiesībā tīri tehniski *init* ir tikai jāaizpilda objekta identifikacija), un tas tiek ielikts *transformācijas datu stekā*.

4.6. Clause_mirrorSet

Starp transformācijā iesaistīto modeļu objektiem (viens objekts ir vienā modelī, bet otrs ir citā modelī) ir iespēja nodibināt "spoguļobjekta" saites. Parasti tas tiek izmantots, lai otrā modelī būtu viegli atrast "spoguļobjektu". Katrs objekts vienā "spoguļmodelī" var rādīt ne vairāk kā uz vienu "spoguļobjektu" (norādes ir neatkarīgas un simetrija nav obligāta).

Šis elements (neciklisks) izveido saiti(es) starp diviem objektiem *a* un *b*. Parametriski tiek regulēts, kura virziena saiti veidot (tiek atbalstīti visi 3 varianti, ieskaitot simetriskās saites, kas ir tipiskākais pielietojums).

Elementa kontekstā jādefinē, kurus 2 modeļus izmantot.

4.7. Clause_mirror

Šis elements (neciklisks) paņem *transformācijas steka objektu* (objekts paliek stekā), atrod tam "spoguļobjektu" un ievieto to stekā (rezultāts nedefinēts, ja "spoguļsaite" nav novilkta).

Elementa kontekstā jādefinē, kurus 2 modeļus izmantot.

4.8. Clause_callNonLoop

Šis elements neveido ciklu un ir paredzēts transformāciju pieraksta strukturizācijas atbalstam. Tā būtība ir izsaukt kādu transformācijas elementu virkni - ***transformācijas likumu*** - neveidojot tradicionālo apstaigāšanu starp izsaucēju un izsaukto likumu. Apstrādes parametri ir *likuma identifikators* un ***apstrādes veids***:

- `internalLoop` izsauktais likums tiek izdarbināts (standarta apstaigāšanas loģika), un šis elements vienmēr beidzas veiksmīgi. Pilnīgs analogs apakšprogrammai, kura neatdod rezultātu.

- `exists` izsauktais likums tiek darbināts speciālā režīmā līdz elementu virknes beigām. Ja pēdējo virknes elementu izdodas veiksmīgi izpildīt (pirmo un vienīgo reizi), tad vadība tiek nodota atpakaļ (tālāka izsauktā likuma apstaigāšana nenotiek) aplūkojamajam Clause_callNonLoop elementam un tas beidz izpildi veiksmīgi. Jāņem vērā, ka izsauktā likuma darbināšanas blakusefekti (piemēram, izmaiņas modeļos) parādās pilnīgi normāli.

- `empty` apstrāde ir pilnīgi analoga `exists` apstrādei. Vienīgā atšķirība ir tas, ka veiksmīgums/neveiksmīgums ir pretējs.

4.9. Clause_any

Šis elements paredzēts brīvai manuālai darbību definēšanai (nav nekāda predefinēta loģika).

4.10. Clause_external

Šis elements neveido ciklu un paredzēts sarežģītu tehnisku manipulāciju veikšanai, kuras grūti noformēt kā funkciju. Tehniski metodē *init* ir jāaizpilda kopējais mainīgais (bool stats), kas signalizē veiksmīgu/neveiksmīgu izpildi.

5. Transformāciju valoda

[2] nodaļa „Transformāciju struktūra" apraksta pamata iespējas transformāciju būvēšanā. Diemžēl šādi veidotas transformācijas nav viegli izveidot un, galvenais, grūti pēc tam lasīt un labot. Tādēļ tiek ieviesta transformāciju pieraksta valoda, kas dod cilvēkiem ērtākas iespējas transformāciju veidošanā un uzturēšanā.
Sintakses likumu pierakstam izmantosim BNF ([3]) paveidu:

- netermināli tiek likti leņķiekavās. Piemēram <unit>.
- termināli, ja to vērtība ir konkrēts teksts, tiek likti apostrofos. Piemēram 'namespace'.
- termināli, ja to vērtība ir teksts no saturīgas grupas, tiek attēloti kā izcelts vārds. Piemēram **NAME.**
- izveduma likuma kreisā puse tiek atdalīta ar kolu, un likums pabeigts ar semikolu. Piemēram, `<namedSet>: <nameShead> '{' <units> '}' ;`
- likuma alternatīvas tiek atdalītas ar vertikālu svītru. Piemēram,
  `<unit> : <block> | <transformation>;`
- neobligātie likuma elementi tiek likti kvadrātiekavās. Piemēram, `<a>:<b>[<c>]<d>;` ir ekvivalents ar `<a>: <b><d> |<b><c><d>;`
- likumos var lietot apaļās iekavas to tradicionālajā nozīmē.

Pielikumos sniegts gan transformācijas pieraksta paraugs (transformācija no loģiskā modeļa uz tehnoloģisko apvienojumu (join) modeli), gan pilns gramatikas teksts.

5.1. <file>

Transformāciju pieraksta vienība ir fails, kurā tiek aprakstīts patvaļīgs skaits transformāciju bloku un transformāciju (skat. [2]). Lai tehniski izmantotu C# vides iespējas, transformācijas un to atsevišķie bloki var tikt iekļauti *namespace* iekavās.

namespace vārds atbilstoši C# prasībām var sastāvēt no elementāriem vārdiem, kas atdalīti ar punktiem.

Precīzāk iepriekš teikto apraksta sekojošais formālais pieraksts:

```
<file>          : [<file> <fileBlock>];
<fileBlock>     : <namedSet>  | <unit>;
<namedSet>      : <nameShead> '{' <units> '}';
<nameShead>     : 'namespace' <namespaceName>;
<namespaceName> : [<namespaceName> "."] NAME ;
<units>         : [<units> <unit>];
<unit>          : <block>  | <transformation>;
```

5.2. <block>

Transformācijas blokam (skat. [2]) ir vārds (identifikators C# nozīmē), kas varētu tikt izmantots atsaucei, veidojot transformācijas. Pašā vienkāršākajā gadījumā transformācija var sastāvēt no viena bloka un šim tipiskajam gadījumam arī kalpo iespēja līdz ar bloku nodefinēt atbilstošo transformāciju.

Obligāta bloka definīcijas sastāvdaļa ir izmantoto modeļu tipi un vārdi (atsaucei likumu elementos). Ir speciālas īsrakstības formas gadījumiem, ja abu modeļu tipi sakrīt, vai izejas un mērķa modeļi sakrīt.

Transformācijas bloka likumi veido bloka ķermeni.

Formāli:

```
<block>     : <blockHead> '{' <rules> '}';
<blockHead> : <blockDescriptor>[ ':' <simpleTransformationName>];
<simpleTransformationName>  : NAME ;
<blockDescriptor>  : 'block' <blockName> '(' <modelType> <modelName>
                        ( ',' modelType> <modelName> |
                          '='            <modelName> |
                          '='                        )
                     ')';
<blockName>  : NAME ;
<modelType>  : NAME ;
<rules>      : [<rules> <rule>];
```

5.3. <rule>

Transformācijas likumi atbilstoši [2] 3.1. Transformācijas likums aprakstītajam dalās izpildāmos (<topRule>) un referencējamos(<subRule>):

```
<rule>  : <topRule> | <subRule>;
```

5.3.1. <topRule>

Izpildāms likums tiek atpazīts pēc tā, ka pirms likuma ķermeņa (tiks aprakstīts vēlāk) ir kols. Izpildāmam likumam var būt neobligāts vārds (C# identifikators), kas ir svarīgs orientācijas līdzeklis likumam atbilstošajā C# tekstā.

```
<topRule>  : [<ruleName>] ':' <ruleBody>;
<ruleName> : NAME ;
```

5.3.2. <subRule>

Tā kā referencējams likums nevar tikt patstāvīgi izpildīts, tad tā apraksts sākas ar obligātu likuma vārdu, aiz kura seko likuma parametru saraksts, kas varētu būt arī tukšs, bet sarakstu aptverošās iekavas ir obligātas. Likuma parametrs ir mainīgais (tiks aprakstīti vēlāk).

```
<subRule>  : <ruleName> '(' [<subRuleParamaters>] ')' <ruleBody>;
<subRuleParamaters> : [ <subRuleParamaters> ','] <variable>;
```

5.4. <ruleBody>

Transformācijas likuma ķermenis ir transformācijas elementu virkne, kas sintaktiski noslēdzas ar semikolu. Lasāmības uzlabošanai elementus drīkst atdalīt ar komatu. Transformācijas likuma elementam drīkst būt vārds, kas ir noderīgs atbilstošā C# koda izpratnē.

```
<ruleBody>      : <ruleElements> ';' ;
<ruleElements>  : [ <ruleElements> <ruleElement>] ;
<ruleElement>   : ',' | <ruleElementName> <realRuleElement>;
<ruleElementName> : [  NAME ':' ];
<realRuleElement>
  : <emptinessElement>
  | <existsElement>
  | <innerLoopElement>
  | <listElement>
  | <creatorElement>
  | <assignElement>
  | <getElement>
  | <goMirror>
  | <lightC>
  | <rawElement>
  | <frameElement>;
```

5.4.1. <emptinessElement>

Šis elements izpildās veiksmīgi tad un tikai tad, ja norādītā elementu virkne vai norādītais referencējamais likums ne reizi nevar izpildīties līdz galam (skat. *Clause_callNonLoop* paveids *empty*) vai arī var teikt, ka ir tukša objektu, kam izpildās apakšlikums, kopa.

Sintaktiski visi apakšlikumus izsaucošie elementi ir ļoti līdzīgi un atšķiras ar operāciju indicējošo simbolu - šinī gadījumā tas ir izsaucējs.

```
<emptinessElement>: '[' '!' <blockOfElements> ']' ;
<blockOfElements> : <ruleElements> | <ruleReference>;
<ruleReference>   : '@' <ruleName>;                                    {
```

5.4.2. <existsElement>

Šis elements izpildās veiksmīgi tad un tikai tad, ja norādītā elementu virkne vai norādītais referencējamais likums kaut reizi izpildās līdz galam (skat. *Clause_callNonLoop* paveids *exists*).

Kopējā šāda veida elementu sintakse aprakstīta 5.4.1. Operācijas simbols ir vertikālā svītra.

```
<existsElement>  : '[' '|' blockOfElements ']';
```

5.4.3. <innerLoopElement>

Šis elements izpildās veiksmīgi vienmēr un apstaigā pilnīgi norādīto elementu virkni vai norādīto referencēto likumu (skat. 4.8 *Clause_callNonLoop* paveids *internalLoop*).

Kopējā šāda veida elementu sisntakse aprakstīta 5.4.1. Operacijas simbols ir zvaigznīte.

```
<innerLoopElement>: '[' '*' <blockOfElements> ']' ;
```

5.4.4. <listElement>

Saraksta elements atlasa visus norādītā veida objektus un apstaigāšanas gaitā iterējas tiem cauri (*Clause_list*). Atlasīšanai ir 2 pamatveidi:

1. Absolūtais. Objekti tiek meklēti norādītajā modelī - terminālis **`MODEL`** ir modeļa vārds no transformācijas bloka definīcijas. Tālāk atkal ir 2 varianti:
   a. tiek atlasīti visi norādītā meta entītiju tipa objekti. Atlasi var ierobežot, norādot, kura objekta sastāvdaļas tās ir.
   b. specsimbols `'#'` norāda, ka tiek ņemts metamodeļa objekts.
2. Relatīvais. Tiek meklētas *transformācijas tekošā objekta* sastāvdaļas. Ja kā saistošais elements ir punkts, tad tiek atlasīti visi norādītā meta entītiju tipa objekti, pretējā gadījumā - visi objekti, uz kuriem rāda norādītā saite (MLink vārds).

```
<listElement>  : MODEL '.' ([ '[' <containerObject> ']'] <entityType> | '#')
               | '.' <entityType>
               | '-' <linkType>;
<containerObject> : <variable>;
<entityType>      : NAME;
<linkType>        : NAME;
```

5.4.5. <creatorElement>

Ar šo elementu tiek radīti jauni modeļa objekti (norādītā meta entīšu tipa). Jāņem vērā, ka objekti tiek veidoti tajā modelī, kuram **nepieder** *transformācijas tekošais objekts*. Tipiskā transformācijā, izmantojot `<listElement>`, tiek apstaigāti izejas modeļa objekti un ar `<creatorElement>` tiek radīti atbilstošie mērķa modeļa objekti.

Sastāvdaļa `<containerObject>` norāda, zem kura objekta pievienot jauno objektu. Ja šis norādījums nav, tad jaunais objekts tiek pievienots tieši modeļa objektam.

Ar mīnusa un tildes palīdzību tiek kontrolēta "spoguļošanas" saišu veidošana. Tipiskajā gadījumā (mīnuss) tiek izveidotas simetriskas saites starp transformācijas tekošo objektu un jauno objektu. Ja izmantota tilde, tad nekādas "spoguļošanas" saites netiek veidotas.

```
<creatorElement> : ':' ('-' |  '~') ['[' <containerObject> ']'] <entityType>;
```

5.4.6. <assignElement>

Transformācijas mainīgajam tiek piešķirts *transformācijas tekošais objekts*. Tipiski šī ir mainīgā pirmā izmantošana likumā un līdz ar to tas automātiski tiek izveidots transformācijas likuma izpildes kontekstā (tālākajās iterācijās tiek izmantots jau izveidotais elements). Izpildāmā likuma izpildes beigās tiek likvidēti visi transformācijas mainīgie.

```
<assignElement>  : '=' <variable>;
```

5.4.7. <getElement>

Mainīgā vērtība ir jaunais *transformācijas tekošais objekts*.

```
<getElement>  : <variable>;
```

5.4.8. <goMirror>

Jaunais *transformācijas tekošais objekts* tiek iegūts pa "spoguļošanas" saiti no vecā.

```
<goMirror>  : '~';
```

5.4.9. <lightC>

Šis elements paredzēts situācijām, kad predefinētās valodas konstrukcijas nedod iespēju veikt gribētās darbības (piemēram, izveidot saites eksemplāru). Šādos gadījumos tiek iekļauts C# paplašināts teksts (protams, rakstītājam jāpārzin atbalsta funkcionalitāte vismaz elementu līmenī).

Terminālis **TEXT** apzīmē patvaļīgu C# teksta fragmentu.

C# teksta paplašinājumi ir atsauces uz transformācijas mainīgajiem. Tās ir 2 veidu:

1. Ar vienu $ norāda, ka šajā vietā tiks iekļauts C# kods, kas iegūst transformācijas mainīgā vērtību.
2. Ar diviem $$ norāda, ka šajā vietā tiks iekļauts C# kods, kas iegūst transformācijas mainīgā unikālo identifikatoru.

```
<lightC>     : '%(' <lightBody> ')%' ;
```

```
<lightBody>  : [ <lightBody> <plus> ] ;
<plus>       : TEXT
             | '$' NAME
             | '$$' NAME ;
```

5.4.10. <rawElement>

Šis sintaktiskais elements paredzēts manuālai transformācijas elementu ievadīšanai un savā sarežģītībā ir samērojams ar <lightC>.

Kā pirmais saturīgais konstrukcijas parametrs ir izveidojamā elementa veids, ja tas nav norādīts, tad tiek veidots *Clause_external*.

Savukārt paplašinātais C# teksts nonāks transformācijas elementa metodes *init* definīcijā. Būtībā paplašinātais C# teksts ir līdzīgs tam, kas aprakstīts pie <lightC> (kas ir speciāla forma tikai priekš *Clause_external*) un atšķiras sintaktiski. Arī paplašinājumu semantika sakrīt, tikai šeit identifikatora pieprasīšanai tiek izmantota zvaigznīte.

```
<rawElement> : '<#' [<rawElementKind>] '+' <extendedCtext> '#>' ;
<rawElementKind>  : NAME;
<extendedCtext>   : [ <extendedCtext> (TEXT |  <implant>)];
<implant>         : '<#=' <implantBody> '#>';
<implantBody>     : ['*'] <variable>;
```

5.4.11. <frameElement>

Šis elements paredzēts "tukša" elementa iekļaušanai likumā. Tiks izveidots transformācijas elements ar norādīto tipu (ja tips nav norādīts, tiek izveidots elements *Clause_any*). Tālākā elementa definēšana veicama ar C# līdzekļiem atbilstošajā kodā.

```
<frameElement>  : '*' [ '(' NAME ')' ];
```

5.5. <variable>

Transformācijas mainīgie un to definīcijas tiek izmantotas dažādos kontekstos. Šeit ir sintaktiskais kopsavilkums saistībā ar mainīgo vārdu formu un pilnu to kvalifikāciju. To, vai var iztikt ar vārdu, vai arī nepieciešama pilna kvalifikācija, nosaka konkrēto likumu konteksts (šī gramatika to neprecizē).

Mainīgā tips var būt gan meta entītes vārds (jākvalificē ar modeļa vārdu no bloka virsraksta), gan C# tips, ko norāda ar simbolu `'#'`.

```
<variable>: <variableName> ['(' <variableDescriptor> ')'];
<variableDescriptor> : <variableType>;
<variableName> : '$' NAME ;
<variableType> : MODEL '.' <entityType>
```

```
            | '#' NAME;
```

5.6. <transformation>

Šis pieraksts paredzēts netriviālu (vairāki modeļi vai vairāki transformācijas bloki) transformāciju definēšanai.

Transformācijai ir obligāts vārds (C# identifikators), neobligāti parametri un saturīgais ķermenis.

```
<transformation>    : 'transformation' <transformationName>
              [<transformationParameters>] '{' <transformationBody> '}';
<transformationName>: NAME;
<transformationBody>: <transformationVariables> <blockCalls>;
```

Transformācijas parametri ir transformācijai nododamo/atgriežamo modeļu vārdi. Tā kā visiem parametriem ir viens un tas pats tips - modelis, tad tas nav norādāms. Parametru vārdi tiks izmantoti, lai regulētu to izmantošanu dažādos transformācijas blokos.

```
<transformationParameters>  : '(' <trmParameters> ')';
<trmParameters>     : [<trmParameters> ','] <modelName>;
<modelName>         : NAME;
```

Transformācijas mainīgo nodaļa tiek izmantota, lai definētu vārdus iekšēji izmantojamajiem modeļiem. Vārdi parametros un mainīgo definīcijas kopā apraksta modeļus, kas tiks izmantoti transformācijā. Protams, parametri un lokālās definīcijas nedrīkst šķelties.

```
<transformationVariables>  : [<transformationVariables> <trVarDefinition> ];
<trVarDefinition>          : 'var' <trmParameters> ';';
```

Transformācijas bloku izsaukumi tiek rakstīti atbilstoši nepieciešamajai secībai, parametros norādot apstrādājamo modeļu vārdus (izejas un mērķa). Modeļu vārdus drīkst nenorādīt, ja transformācija ir tieši ar diviem parametriem, kas arī ir bloka izsaukuma parametri.

```
<blockCalls> : [ <blockCalls> <blockCall>];
<blockCall>  : <fullBlockName> <callParameters> ';' ;
<fullBlockName> : <namespaceName> '.' NAME ;
<callParameters>: [ '(' <modelName> ',' <modelName> ')';
```

6. Secinājumi un rezultāti

Aktivitātes ietvaros ir izstrādāti satrurīgi elementi un transformācijas elementārā loģika, tai skaitā, apstaigāšanas loģika. Kā arī ir izstrādāta un realizēta transformācijas valoda, kas izmantota modeļa apstrādes transformāciju mašīnā.

7. Literatūras saraksts

[1] Plume J., Strods J., Karlsons I., Smilts U., Smotrovs J., Gavars G., Stasko I., Dancis M. Meta-model Based Data Conversion. In J.Barzdins(ed.),3rd International Workshop on Databases and Information Systems, Vol.1, Latvian Academic Library, 1998.
[2] 1.3.3. " Modeļa apstrādes transformāciju mašīnas izstrāde, kas ietver daudz-modeļu minipulācijas" progresa pārskats
[3] http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form

8. Pielikumi.

8.1. Transformāciju pieraksta gramatika

Meklēšanas ērtībai likumi izvietoti alfabētiskā secībā. Sākuma simbols ir `<file>`.

```
<assignElement>  : '=' <variable>;
<block>          : <blockHead> '{' <rules> '}';
<blockCall>      : <fullBlockName> <callParameters> ';' ;
<blockCalls>     : [ <blockCalls> <blockCall>];
<blockDescriptor>: 'block' <blockName> '(' <modelType> <modelName>
                        ( ',' modelType> <modelName> |
                          '='            <modelName> |
                          '='                        )
                     ')';
<blockHead>      : <blockDescriptor>[ ':' <simpleTransformationName>];
<blockName>      : NAME ;
<blockOfElements>: <ruleElements> | <ruleReference>;
<callParameters> : [ '(' <modelName> ',' <modelName> ')';
<containerObject>: <variable>;
<creatorElement> : ':' ('-' |  '~') ['[' <containerObject> ']'] <entityType>;
<emptinessElement>: '[' '!' <blockOfElements> ']' ;
<entityType>     : NAME;
<existsElement>  : '[' '|' blockOfElements ']';
<extendedCtext>  : [ <extendedCtext> (TEXT |  <implant>)];
<file>           : [<file> <fileBlock>];
<fileBlock>      : <namedSet>  | <unit>;
<frameElement>   : '*' [ '(' NAME ')' ];
<fullBlockName>  : <namespaceName> '.' NAME ;
<getElement>     : <variable>;
<goMirror>       : '~';
<implant>        : '<#=' <implantBody> '#>';
<implantBody>    : ['*'] <variable>;
<innerLoopElement>: '[' '*' <blockOfElements> ']' ;
<lightBody>      : [ <lightBody> <plus> ] ;
<lightC>         : '%(' <lightBody> ')%' ;
<linkType>       : NAME;
<listElement>    : MODEL '.' ([ '[' <containerObject> ']'] <entityType> | '#')
                 | '.' <entityType>
                 | '-' <linkType>;
<modelName>      : NAME;
<modelType>      : NAME ;
<namedSet>       : <nameShead> '{' <units> '}';
<nameShead>      : 'namespace' <namespaceName>;
<namespaceName>  : [<namespaceName> "."] NAME ;
<plus>           : TEXT | '$' NAME | '$$' NAME ;
<rawElement> : '<#' [<rawElementKind>] '+' <extendedCtext> '#>' ;
<rawElementKind>  : NAME;
<realRuleElement>
  : <emptinessElement>
  | <existsElement>
  | <innerLoopElement>
  | <listElement>
  | <creatorElement>
```

```
  | <assignElement>
  | <getElement>
  | <goMirror>
  | <lightC>
  | <rawElement>
  | <frameElement>;
<rule>           : <topRule> | <subRule>;
<ruleBody>       : <ruleElements> ';' ;
<ruleElements>   : [ <ruleElements> <ruleElement>] ;
<ruleElement>    : ',' | <ruleElementName> <realRuleElement>;
<ruleElementName>: [  NAME ':' ];
<ruleName>       : NAME ;
<ruleReference>  : '@' <ruleName>;                                    {
<rules>          : [<rules> <rule>];
<simpleTransformationName>: NAME ;
<subRule>        : <ruleName> '(' [<subRuleParamaters>] ')' <ruleBody>;
<subRuleParamaters>: [ <subRuleParamaters> ','] <variable>;
<topRule>        : [<ruleName>] ':' <ruleBody>;
<transformation> : 'transformation' <transformationName>
       [<transformationParameters>] '{' <transformationBody> '}';
<transformationBody>: <transformationVariables> <blockCalls>;
<transformationName>: NAME;
<transformationParameters>: '(' <trmParameters> ')';
<transformationVariables> : [<transformationVariables> <trVarDefinition> ];
<trmParameters>  : [<trmParameters> ','] <modelName>;
<trVarDefinition>: 'var' <trmParameters> ';';
<unit>           : <block>  | <transformation>;
<units>          : [<units> <unit>];
<variable>       : <variableName> ['(' <variableDescriptor> ')'];
<variableDescriptor> : <variableType>;
<variableName>   : '$' NAME ;
<variableType>   : MODEL '.' <entityType>
                 | '#' NAME;
```

8.2. Transformācijas pieraksta paraugs.

```
namespace LogicalJoin2 {
 block techno(LogicalJoin FR, Joins TO) :Ttechno {
 model_dataBase:
  %( var x=myBlockT.onDataBase;
          myBlockT.tModelTyped.dataBase=
                (string.IsNullOrWhiteSpace(x))?myBlockT.sModelTyped.dataBase:x;
)%;

 base:
  FR.Join=$j
  %( var x=$j; if(x.MDS__isSelected(myBlockT.onDataBase)==false) return false; )%
       $j :- Join=$jT %( $jT.name=$j.name; )%
       $j.Jpart=$p :- [$jT]JPart=$pT
       %(if($p.basedOn.isHardDelete==true)
          $pT.isHardDelete=true;)%
```

```
        $p-basedOn=$e
        %( var x=$pT; x.name=$p.name; x.tableName=$e.name; )%

 main_next:
  FR.Join=$j
  %( var x=$j; if(x.MDS__isSelected(myBlockT.onDataBase)==false) return false; )%
        $j~=$jT(TO.Join)
        $j-main=$p~=$pT(TO.JPart)
        %( $jT.MDS_pp_main.addValue($$pT); )%
  $j
 orderedP: .Jpart=$p1
  %(if($$p==$$p1) return false;)%
        $p~=$pT(TO.JPart)
        $p1~=$p1T(TO.Jpart)
        %( $pT.MDS_pp_next.addValue($$p1T); )%
        $p1=$p
        ;

 link:
  FR.Join=$j
  %( var x=$j; if(x.MDS__isSelected(myBlockT.onDataBase)==false) return false; )%
        $j.Jpart.Jlink=$l:-JCondition=$lT
        $l-to=$pt~=$ptT(TO.JPart)
        $l-basedOn=$lb
        %(if($lb.isBase)
           $lT.isBase=true;
                )%
        $pt-basedOn=$e
        %( myBlock.variables["tPK"]=new MEDUS.UniversalObject(); )%
        [| @findPK]
        $tPK(#string)
        %( myBlock.variables["pf"]=new
MEDUS.UniversalObject($l.MDSbox.myContainer, false); )%
        $pf(FR.Jpart)~=$pfT(TO.JPart)
        %( var l=$l;
          var ex=l.extend;
          var pf=$pf;
          var pt=$pt;
          var tPK=$tPK;
          var lT=$lT;
          if(pf.position==pt.position){
            var m=string.Format("{0}[{1}].{2}->{3}[{4}].{5}{6}",
              pf.name, pf.position, l.basedOn.name, pt.name, pt.position, tPK,
ex==null?"":"("+ex+")");
        //    lT.sourceField=m;
                  return true;
```

```
            }
      if(pf.position<pt.position){
            $ptT.MDS_pp_condition.addValue($$lT);
            lT.MDS_pp_sourceJPart.addValue($$pfT);
//          lT.sourceField=l.basedOn.name;
//          lT.targetField=tPK;
            switch(ex){
              case null: break;
              case "target": lT.isLeftJoin=true; break;
              case "source": lT.isRightJoin=true; break;
              case "both":   lT.isLeftJoin=true; lT.isRightJoin=true; break;
              }
            } else {
            $pfT.MDS_pp_condition.addValue($$lT);
            lT.MDS_pp_sourceJPart.addValue($$ptT);
//          lT.sourceField=tPK;
//          lT.targetField=l.basedOn.name;
            switch(ex){
              case null: break;
              case "target": lT.isRightJoin=true; break;
              case "source": lT.isLeftJoin=true; break;
              case "both":   lT.isLeftJoin=true; lT.isRightJoin=true; break;
              }
             }
            )%
      $lb.SubField=$sf :~ [$lT]JoiningPair=$jpT
      $sf-pkType=$fts
      %( var x=$sf; var y=$jpT;
        y.type=$fts.type;
        if($pf.position<$pt.position){
              y.sourceField=x.name;
              y.targetField=x.targetUnitName;
                    if($lb.isMandatory)
                      y.sourceFieldIsMandatory=true;
                    y.targetFieldIsMandatory=true;
              } else {
              y.sourceField=x.targetUnitName;
              y.targetField=x.name;
                    if($lb.isMandatory)
                      y.targetFieldIsMandatory=true;
                    y.sourceFieldIsMandatory=true;
              }
       )%;

findPK($e(FR.Entity))
  $e.Unit=$u
```

```
        %(if($u.isPK!=true) return false;
          myBlock.variables["tPK"]=new MEDUS.UniversalObject($u.name); )%
  ;
  }
}
```