Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Kiberfizikai rendszerek gráfmintaillesztés alapú elosztott ellenőrzése futási időben

TDK dolgozat

Készítette:

Tóth Krisztián
Gábor Szilágyi

Konzulens:

Vörös András
Búr Márton
Szárnyas Gábor

2016

Tartalomjegyzék

Kivonat

A technológia fejlődésével rohamosan jelennek meg a kiberfizikai rendszerek egyre több tradicionális területen is: a vasúti rendszerek, robot rendszerek megfigyelését egyre több szenzor végzi, autók egymással kommunikálnak, és gyakran már önvezető funkcióval is rendelkeznek. Jellemzője ezen rendszereknek, hogy nagy mennyiségű szenzor adatot kell feldolgozniuk, és a rendelkezésre álló információk alapján gyorsan kell reagálniuk a környezet változásaira.

A kiberfizikai rendszerek gyakran kritikus feladatokat látnak el, ahol elengedhetetlen a helyes működés. Ennek biztosítására azonban nem mindig elegendőek a tradicionális, biztonságkritikus rendszerek esetén alkalmazott megközelítések, hiszen a gyorsan változó környezet, az alkalmazott intelligens megoldások és az elosztottság nem teszi lehetővé a tervezési idejű ellenőrzést. Erre nyújthat megoldást a futásidejű ellenőrzés, amelyre többféle megközelítés is létezik. Az időbeli viselkedéseket jellemzően automata formalizmusok segítségével és temporális nyelvekkel, míg a strukturális felépítést és adat jellegű viselkedést gráfminták segítségével tudjuk specifikálni és gráfmintaillesztés segítségével ellenőrizni. Az irodalomban is több megközelítés ismert, mi ezeket továbbfejlesztve egy olyan gráfmintaillesztés alapú elosztott ellenőrzést megvalósító keretrendszert terveztünk, amely képes egyrészt az elosztott rendszer lokális tulajdonságait vizsgálni, továbbá ezek alapján a rendszer állapotára következtetni és a lehetséges hibákat jelezni.

A keretrendszer egy nyílt forráskódú gráfmintaillesztő rendszerre épül. Munkánk során ezt egészítettük ki új algoritmusokkal, hogy a gráfminta specifikációk bővebb körét tudjuk támogatni. Amennyiben az ellenőrzéshez szükséges, az elosztottan futtatott ellenőrzések eredményeiből automatikusan számítjuk a rendszer új állapotát. Nyelvi támogatást adunk a szenzor adatok feldolgozására és a tudásbázisba való integrálásának támogatására. Az általunk fejlesztett gráfmintaillesztés alapú keretrendszerrel támogatjuk az elosztott kiberfizikai rendszerek ellenőrzését, és a megközelítésünk működését egy esettanulmány segítségével demonstráljuk.

Abstract

The rapid development of technology leads to the rise of Cyber-Physical Systems even in the field of safety critical systems like railway, robot, and self-driving car systems. Cyber-physical systems process a huge amount of data coming from sensors and other information sources and it often has to provide real-time feedback and reaction.

Cyber-Physical Systems are often critical, which means that their failure can lead to serious injuries or even loss of human lives. Ensuring correctness is an important issue, however traditional design-time verification approaches can not be applied due to the complex interaction with the environment, the distributed behavior and the intelligent controller solutions. Runtime analysis provides a solution where graph-based specification languages and analysis algorithms are the proper means to analyze the behavior of cyber-physical systems at runtime. Existing approaches from the literature formed the basis of our work: we developed a distributed runtime verification framework to analyze the local behavior of the components and ensure the global correctness of the systems.

We developed a framework on top of an existing open-source solution. We extended the framework to support a richer set of possible specifications and we implemented the corresponding algorithms. We provided a language to support the integration of sensor information to the knowledge base and we also developed a simple language to decompose the graph specifications and distribute the analysis algorithm to the components. In case of complex global specifications, the algorithm collects the information from the various analysis components and infers the state of the system. The introduced framework was evaluated in a research project of the department and proved its usefulness.

1. fejezet

Bevezetés

Biztonságkritikus kiberfizikai rendszerek működése során kiemelten fontos a helyes működés biztosítása. A rendszertervezés elején el kell döntenünk, hogyan fogjuk biztosítani a specifikációban elvárt működést, mivel ez határozza meg a felhasználható eszközöket. Egy általános megoldást fogunk bemutatni, mely robusztus és megbízható rendszerek fejlesztését teszi lehetővé.

A helyes működés biztosításához nem hagyatkozhatunk csupán a rendszertervezők és szoftverfejlesztők szakértelmére, más módszert kell találnunk rendszerünk ellenőrzésére. Ez azért nehéz feladat, mert rengeteg változó befolyásolja az elkészült rendszer működését.

Az iparban alkalmazott megoldás, hogy szoftverkomponensek tervezési idejű helyességét tesztesetekkel ellenőrzik. Ebben az esetben csak egy szimulációs környezetben ellenőrzik a rendszer helyességét, de ezzel nehezen lehet egy dinamikusan változó rendszer minden lehetséges lefutását megvizsgálni. Erre nyújthat megoldást a futásidejű ellenőrzés, amelyre többféle megközelítés is létezik. Az időbeli viselkedéseket jellemzően automata formalizmusok segítségével és temporális nyelvekkel, míg a strukturális felépítést és adat jellegű viselkedést gráfminták segítségével tudjuk specifikálni és gráfmintaillesztés segítségével ellenőrizni.

Az irodalomban is több megközelítés ismert, mi ezeket továbbfejlesztve egy olyan gráfmintaillesztés alapú elosztott ellenőrzést megvalósító keretrendszert terveztünk, amely képes egyrészt az elosztott rendszer lokális tulajdonságait vizsgálni, továbbá ezek alapján a rendszer állapotára következtetni és a lehetséges hibákat jelezni.

A keretrendszer egy nyílt forráskódú gráfmintaillesztő rendszerre épül. Munkánk során ezt egészítettük ki új algoritmusokkal és terveztünk egy elosztott lekérdezés végrehajtó rendszert, mellyel komplex rendszereket is egyszerűen tudunk modellezni és azon lekérdezéseket futtatni. Amennyiben az ellenőrzéshez szükséges, az elosztottan futtatott ellenőrzések eredményeiből automatikusan számítjuk a rendszer új állapotát. Nyelvi támogatást adunk a szenzor adatok feldolgozására és a tudásbázisba való integrálásának támogatására. A megközelítésünk működését egy esettanulmány segítségével demonstráljuk.

A 2. fejezetben bevezetjük a dolgozatban használt alapvető fogalmakat és technológiákat, melyek ismerete elengedhetetlen a munkánk megismeréséhez. A 3. fejezetben bemutatjuk munkánk első részét, a lekérdezés végrehajtó keretrendszert, majd a lokális kereső algoritmus kiterjesztését mutatjuk be, kitérve az implementációs kihívásokra és megoldásokra. A 4. fejezetben prezentálunk egy megközelítést szenzoradat modellalapú integrációjára, majd egy egyszerű példa segítségével bemutatjuk a mérnöki tervezési folyamatot. Az 5. fejezetben egy megoldást kínálunk komplex lekérdezések végrehajtására elosztott modelleken, majd egy példán keresztül bemutatjuk rendszerünk működését. A 6. fejezetben egy létező kiberfizikai rendszeren fogalmazunk meg egy problémát és arra egy megoldást kínálunk, végül a 7. fejezetben összefoglaljuk az eredményeket.

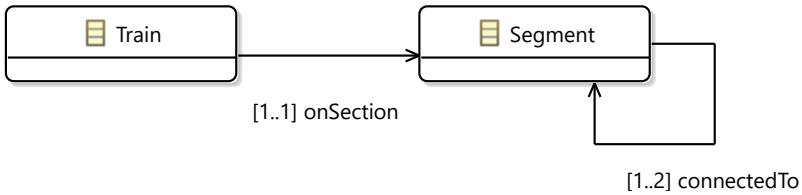
2. fejezet

Háttérismeretek

Dolgozatunk során kiberfizikai rendszerek futásidejű ellenőrzésével foglalkoztunk. Az ehhez szükséges háttérismereteket és fogalmakat mutatjuk be ebben a fejezetben.

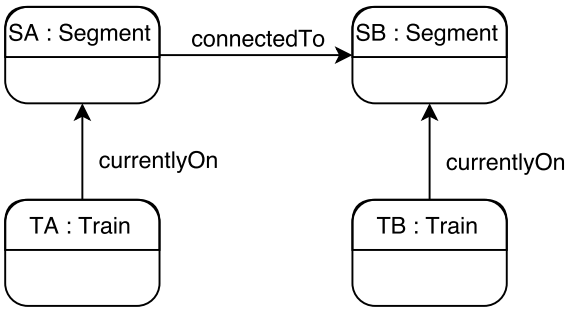
2.1. Példa

A fogalmak ismertetéséhez vizsgáljuk meg a következő példát. Egy vasúti rendszer ellenőrzéséhez egy egyszerű metamodellt készítünk, amely leírja, hogy milyen elemekkel modellezzük a rendszer állapotát (ld. 2.1. ábra). A modell vasútszakaszokból (`Segment`) épül fel, melyek további vasútszakaszokhoz kapcsolódhatnak (`connectedTo`). A vasúti rendszerben közlekedő vonatok (`Train`) mindig egy adott vasútszakaszon tartózkodnak (`currentlyOn`).



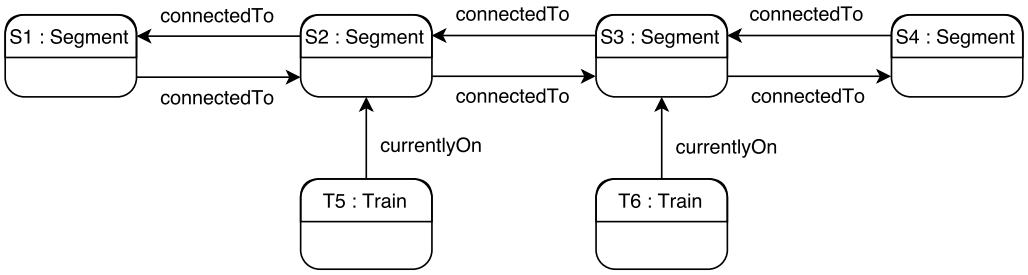
2.1. ábra. Egy egyszerű vasúthálózat metamodellje

A rendszer biztonságos üzemeltetéséhez fontos jelezni, ha két szomszédos szakaszon is vonat helyezkedik el. Ezt egy ún. gráfmintával adhatjuk meg, amelyet grafikusan a 2.1. ábrán látható módon szemléltetünk. (ld. 2.2. bekezdés)



2.2. ábra. Egy keresett gráfminta grafikus ábrázolása

A továbbiakban vizsgáljuk azt az esetet, amikor a rendsezr állapotát a 2.3. ábrán látható példánymodell írja le.



2.3. ábra. Egy lehetséges példánymodell, ami a rendszer egy állapotát írja le.

2.2. Fogalmak

A megoldandó kihívások részletes tárgyalása előtt definiáljuk azokat alapfogalmakat, melyek elengedhetetlenek a modellalapú szoftvertervezés és kiberfizikai rendszerek témakörében.

Modellalapú tervezés Modellalapú fejlesztés (*model-driven engineering*) esetén a tervezés fókusza különböző mérnöki modellek pl. strukturális és viselkedési modellek elkészítése. A modellek alkalmasak arra, hogy azokból a rendszer egy komponenseinek interfészeit, ill. bizonyos esetekben megvalósításukat származtassuk (pl. kódgenerálási technikák segítségével).

Tervezési idejű ellenőrzés Modellalapú tervezés alkalmazása esetén a tervezés fázisában a rendszer modelljeinek elkészítésekor lehetőségünk nyílik a rendszertervek ellenőrzésére (*design-time verification*). Validáció során ellenőrizhetjük, hogy a tervezési szabályoknak megfelel-e a modellünk, vagy végezhetünk verifikációt, amely során a rendszer lehetséges viselkedéseit vizsgálva tudunk meggyőződni a tervek helyességéről.

Futásidejű ellenőrzés Futásidejű ellenőrzés (*runtime verification*) során a rendszer helyességét működés közben, az éles rendszer futásait vizsgálva ellenőrizzük [11]. Nagy előnye a tervezési idejű ellenőrzéssel ellentétben, hogy ezzel a módszerrel elosztott, komplex interakciókat is nyomon lehet követni és ellenőrizni azok helyességét. Abban az esetben, amikor a komplexitás miatt nem lehetséges a tervezési idejű ellenőrzés, ez a módszer jelenthet megoldást.

Kiberfizikai rendszerek Kiberfizikai rendszereknek (*cyber-physical systems*, CPS) nevezzük azokat a számítógépes rendszereket, amelyek a fizikai világgal interakcióban vannak. Alapvető elvárás ezekkel a rendszerekkel szemben, hogy alkalmazkodni tudjanak a dinamikusan változó környezethez [9, 6]. Ezekben a rendszerekben általában közös, hogy rendelkeznek valamilyen érzékelővel (*sensor*), amivel a külvilágról adatot tudnak gyűjteni, valamilyen beavatkozóval (*actuator*) tudnak a környezetre hatni és rendelkeznek valamilyen intelligenciával. Ezen rendszerek változatos felépítésűek lehetnek, a beágyazott eszközöktől a felhő alapú szolgáltatásokig terjed a lehetséges informatikai komponensek halmaza.

Modellezés, példánymodell és metamodell A modell vagy példánymodell (*instance model*) általában a fizikai világ egy leképzése, mely során absztrakcióval élve egyszerűsítjük a világról alkotott képet (2.3. ábra). A metamodell (*metamodel*) egy olyan modellezési nyelv, ami meghatározza azon elemeket, amikből felépülhet egy modell. A metamodell egy közös struktúrát fogalmaz meg modellek egy csoportjára (2.1. ábra).

Gráfminta és mintaillesztés A gráfminta lényege, hogy egy gráfban bizonyos kényszereket (*constraints*) kielégítő részgráfokat keresünk. Ezért *gráfmintának* (*graph pattern*) nevezzük változók sorozatát és a változókra vonatkozó kényszerek halmazát.

Egy adott gráfminta esetén *eredményhalmaznak* (*result set*) nevezzük azon, n-esek (*tuples*) halmazát, amelyeket a változókba helyettesítve kielégítik a gráfmintában lévő kényszereket. Az eredményhalmaz elemeit *illeszkedésnek* (*match*) nevezzük.

A gráfmintákat a jelen dokumentumban kétféleképpen adjuk meg. Egyrészt a változók és a kényszerek felsorolásával (ld. 2.4. alfejezet), másrészt grafikus ábrázolással (2.2. ábra).

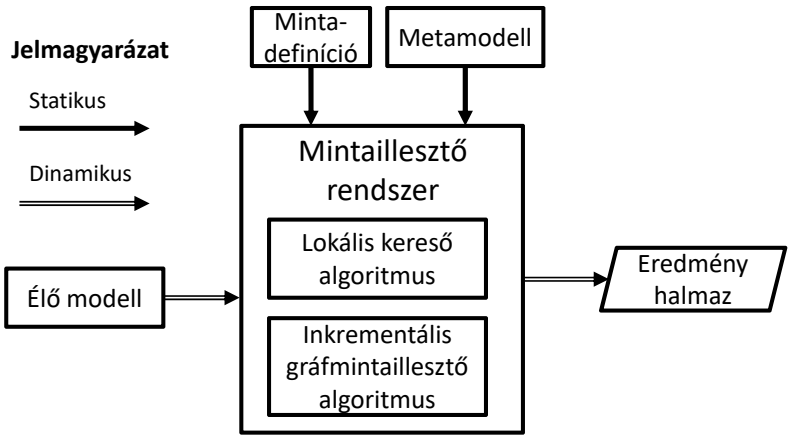
2.3. Modell-lekérdezések gráfminták segítségével

A gráfminták illesztése során tulajdonképpen modell-lekérdezéseket hajtunk végre, ahol a lekérdezés feltételeit egy gráfminta definiálja. Ez a koncepció motivált minket, hogy megoldást keressünk ilyen lekérdezések megfogalmazására.

2.3.1. VIATRA keretrendszer

A VIATRA egy eseményvezérelt, reaktív modelltranszformációs platform, mely rendelkezik egy deklaratív lekérdezési nyelvvel (VIATRA Query Language, VQL) [2]. A nyelvi elemkészletet a 2.4. fejezetben ismertetjük. Ezen nyelv segítségével megfogalmazhatunk egy gráfmintát a metamodell osztályaival, azok attribútumaival és asszociációival.

A VIATRA rendszer legfontosabb komponense egy gráfmintaillesztő komponens, melynek architektúrája a 2.4. ábrán látható.



2.4. ábra. VIATRA gráfmintaillesztő rendszere

1. **Minta definíció.** A VIATRA lekérdezőnyelvében megfogalmazott gráfminták. Részletesen a 2.4. fejezetben tárgyaljuk őket.

2. **Metamodell.** A metamodell adja meg szemantikáját a gráfminta elemeinek. Egy olyan elemkészlet, melyre gráfmintákat írhatunk fel.

3. **Mintaillesztő algoritmusok.** A VIATRA lekérdezés futtató keretrendszere két merőben eltérő algoritmust ad a fejlesztő kezébe: inkrementális (2.3.2. szakasz) és keresés-alapú (2.3.3. szakasz) algoritmusok különböző előnyökkel és hátrányokkal rendelkeznek.

4. **Mintaillesztő rendszer.** Ennek a komponensnek a feladata összegyűjteni az eredmény előállításához szükséges elemeket, kiválasztva a futtatandó algoritmust.

5. **Eredményhalmaz.** A bemeneti mintához tartozó illeszkedések halmaza.

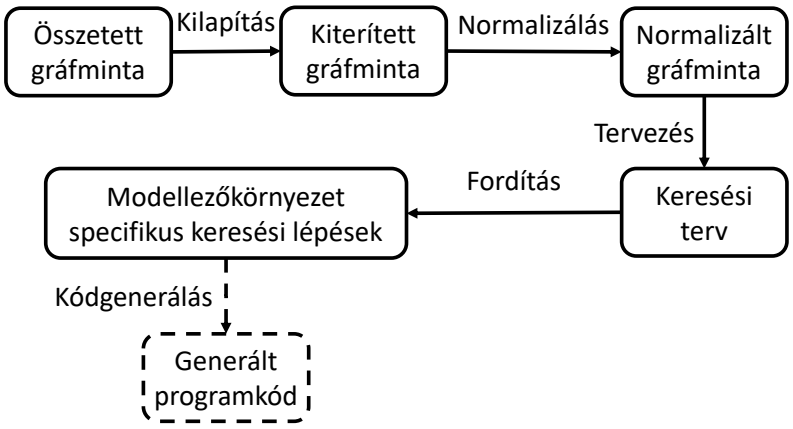
2.3.2. Inkrementális gráfmintaillesztő algoritmus

A VIATRA keretrendszer gráfmintaillesztő komponense eredetileg kizárólag egy inkrementális algoritmussal [1] valósította meg az illesztést. Az algoritmus alapelve az, hogy építsünk egy adatstruktúrát, ami képes meghatározni a lekérdezés eredményét. Ezt töltsük fel a példánymodelljeink elemeivel, megkapva a lekérdezés eredményhalmazát. Azután csak a példánymodellek változását terjesszük el az adatstruktúrán, azzal ellentétben, hogy az egész példánymodellt újra betöltenénk [3].

Ennek a megoldásnak hátránya a memória felhasználás, mivel az adatstruktúra mérete függ a modelltől és a gráfminta összetettségétől. Gyakorlati alkalmazásban, amikor a modellek önmagukban óriásiak, ennek a megoldásnak a memóriaigénye kielégíthetetlen.

2.3.3. Lokális kereső algoritmus

A lokális kereső algoritmus abból a célból készült, hogy egy alacsony memóriaigényű megoldást kínáljon, alternatívaként a gyors lekérdezéseket lehetővé tevő, de magas memóriaigényű inkrementális algoritmussal szemben.



2.5. ábra. Keresési terv előállítása és kódgenerálás

A keresési terv előállítása során egy művelet sorozatot hajtunk végre a gráfmintán, ez látható a 2.5. ábrán. A nyilak jelentik a folyamat lépéseit, a dobozok a gráfminta köztes reprezentációit.

1. **Összetett gráfminta (*composite graph pattern*).** A VIATRA Query nyelve lehetőséget ad komplex gráfminták komponálására, amelyek hivatkozhatnak más gráfmintákra is.

2. **Kiterített gráfminta (*flattened graph pattern*).** Egy, az eredetivel ekvivalens jelentésű egyszerű gráfminta.

3. **Normalizált gráfminta (*normalized graph pattern*).** A metamodellből kikövetkeztethető feltételektől és duplikált kényszerektől is mentes gráfminta.

4. **Keresési terv (*logical search plan*).** Egy szekvenciális keresési terv, amely a deklaratívan megfogalmazott gráfmintákból a metamodell jellemzőinek, illetve a keresési műveletek költségének figyelembevételével készül.

5. **Modellezőkörnyezet specifikus keresési lépések (*physical search plan*).** A keresési terv már a modellező környezetben interpretáltan végrehajtható keresési lépéseket tartalmaz.

6. **Generált kód (*generated code*).** Egy lehetséges további lépés a kódgenerálás, melynek kimenete egy olyan önálló alkalmazás forráskódját lehet, amit adott platformra le lehet fordítani és a modellező környezettől függetlenül, a lefordított bináris állományból lehet futtatni.

A lokális kereső algoritmus várhatóan lényegesen kevesebb operatív memóriát igényel, mint az inkrementális gráfmintaillesztő algoritmus, azonban ez a számítási gyorsaság rovására mehet olyan rendszerek ellenőrzésénél, amelyeknek arányaiban kis része változik és folyamatosan kell ellenőrizni a gráfmintákat [3].

2.3.4. Keresési terv

A keresési terv kétféle keresési műveletet tartalmazhat.

- Ellenőrzés (*check*) – Ellenőrzés esetén megvizsgáljuk, hogy a lekötött változók értéke megfelel-e egy adott kényszernek. Amennyiben nem, a változólekötést eldobjuk, egyéb esetben megtartjuk.
- Kiterjesztés (*extend*) – Kiterjesztés során az egyik lekötetlen változókat kötjük le egy H halmaz elemeire.

A futás során *keretnek* (*frame*) nevezzük egy minta változóinak egy adott lekötését. Ekkor a keresés során egy olyan keretből indulunk ki, amely minden változó lekötetlen, azaz minden változóhoz `null` értéket rendel. A keresési terv azt adja meg, hogy milyen szabályok szerint kell megváltoztatni egy adott keretet, hogy a végén a keret olyan változólekötéseket tartalmazzon, amelyre teljesülnek a mintában szereplő kényszerek.

2.4. A VIATRA lekérdező nyelv ismertetése

A VIATRA lekérdező nyelv számos nyelvi elemet tartalmaz, amely lehetővé teszi változatos, különböző alkalmazásokhoz megfelelő minták leírását. Ebben az alfejezetben ezeket a nyelvi elemeket ismertetjük, minden esetben egy példát adva VQL-ben az adott elem használatára.

Minta definiálása Egy mintát a `pattern` kulcszóval lehet megadni. A minta fejlécében definiáljuk a minta nevét és a minta paraméterváltozóit. Ezen kívül a mintában további változókat használhatunk, a mintához tartozó eredményhalmaz elemeit a paraméterváltozóra vetítve csak a fontos az illeszkedések paraméterekre való vetítését fogja tartalmazni. A minta törzsében adjuk meg azokat a kényszereket, amelyeknek teljesülnie kell az illeszkedésekre.

```
pattern SomePattern(TA:Train, TB:Train)
{
  Train.currentlyOn(TA, s);
  Train.currentlyOn(TB, s);
  TA != TB;
}
```

Típuskényszer Típuskényszerek segítségével tudjuk megfogalmazni a gráfminta által keresett objektumok típusát a modellben, azaz le tudjuk szűrni a gráfminta lehetséges illeszkedéseit.

```
Train(TA);
```

Útvonalkifejezés Az útvonalkifejezéssel adhatunk meg olyan kényszert, amely azt köti ki, hogy adott referenciákon való navigációk és/vagy attribútumlekérdezés adott sorrendű végrehajtásán keresztül lehet jutni egy adott változóból a másikba.

```
// A train vonat currentlyOn referenciaja a segment valtozora mutat
Train.currentlyOn(train, segment);
// Letezik olyan v1 es v2 elem, hogy train currentlyOn referenciaja a v1 valtozora mutat
// v1 connectedTo-ja a v2-re, v2 connectedTo-ja pedig a seg2-re.
Train.currentlyOn.connectedTo.connectedTo(train, seg2);
```

Ellenőrző kifejezés Az ellenőrző kifejezés[1] egy olyan kényszer egy gráfmintában, amelyet primitív típusú változóktól függő predikátummal kell megadni. A kényszer akkor teljesül egy adott változólekötésnél, amennyiben a predikátum teljesül a behelyettesített értékekkel.

```
pattern isNameCapital(e:Employee)
{
  Employee.name(e, employeeName);
  check(employeeName == employeeName.toUpper());
}
```

Kiértékelő kifejezés A kiértékelő kifejezés (`eval` kulcsszó) egy olyan kényszer egy gráfmintában, amely Xbase szintaktikájú kifejezéseket [13] tartalmaz megkötésekkel, ezekben hivatkozhatunk a minta változóira.

```
pattern capitalName(e:Employee, resultName)
{
  Employee.name(e, employeeName);
  resultName == eval(employeeName.toUpper());
}
```

Konstans értékkényszer A mintadefinícióban megadhatunk konstansokat, például számokat, felsorolt típusú értékeket és logikai értékeket. Minden a mintadefinícióban hivatkozott előre definiált érték ilyen kényszerré alakul.

- **Egyenlőség vizsgálat.** Modellelemek és/vagy konstansok egyenlőségét leíró kényszer.

```
pattern isNameCapital(e:Employee)
{
  Employee.name(e, employeeName);
  employeeName == "John";
}
```

- **Egyenlőtlenség vizsgálat.** Modellelemek és/vagy konstansok egyenlőtlenségét leíró kényszer.

```
pattern sameNameEmployee(e1:Employee, e2:Employee)
{
  Employee.name(e1, employeeName);
  Employee.name(e2, employeeName);
  e1 != e2;
}
```

[1] Az ellenőrző kifejezéseket VQL-ben a `check` kulcsszóval jelöljük. Fontos, hogy ez nem mindig egyezik meg a 2.3.4. szakaszban ismertetett ellenőrzés (*check*) művelettel, ami egy általános operátor típus- és értékkényszerek ellenőrzésére.

Gráfmintára illeszkedés vizsgálata. Előre definiált gráfminták újrahasználatára, kiterjesztésére alkalmas elem (`find` kulcsszó). Leszűkíthetjük a hivatkozott gráfmintának paraméterben megadott változók illeszkedési halmazát. Ez a nyelvi elem feldolgozás során helyettesíthető a hivatkozott gráfmintából generált keresési tervvel.

```
pattern onSamePosition(TA:Train, TB:Train)
{
  Train.currentlyOn(TA, s);
  Train.currentlyOn(TB, s);
  TA != TB;
}
pattern isDangerous(TA:Train, TB:Train)
{
  find onSamePosition(TA, TB);
}
```

Gráfmintára nem illeszkedés vizsgálata. Ez a nyelvi elem akkor fog illeszkedést eredményezni, ha a hivatkozott gráfmintára nincs illeszkedés.

```
pattern isSafe(TA:Train, TB:Train)
{
  neg find onSamePosition(TA, TB);
}
```

Gráfmintára illeszkedő eredmények számának vizsgálata. Definiált gráfmintákra illeszkedő eredmények számára tudunk ezáltal szűrést megfogalmazni.

```
pattern isSafe(TA:Train, TB:Train)
{
  0 == count find onSamePosition(TA, TB);
}
```

Tranzitív lezárt. A tranzitív lezárásra először bemutatunk az algoritmuselméletben általánosan ismert definícióját, majd annak megfelelőjét a gráfmintaillesztés területén.

- **Definíció 1 (algoritmuselmélet).** Egy G irányított gráf tranzitív lezártja az a G^+ gráf, amelyre teljesül, hogy
  - $V(G^+) = V(G)$
  - $(u, v) \in E(G^+)$ pontosan akkor, ha létezik irányított út G gráfban u-tól v-ig. ▪

- **Definíció 2 (gráfmintaillesztés).** Legyen adott Π, 2 változós gráfminta. Ekkor képezzük azt a G irányított gráfot, amelynek csúcsai a példánymodell elemei, a és b modellelem között pedig akkor fut irányított él, ha (a, b) illeszkedik a Π gráfmintára. Ekkor Π tranzitív lezártja az a Π^+ 2 változós gráfminta, amelyre egy (c, d) elempár akkor illeszkedik, ha $(c, d) \in E(G^+)$. ▪

Hasonlóan a *gráfmintára illeszkedés vizsgálatához* megadhatjuk azt is kényszerként, hogy a mintában 2 változó egy már definiált gráfminta tranzitív lezártjára illeszkedjen.

```
pattern linked(SA:Segment, SB:Segment)
{
  Segment.connectedTo(SA,SB);
}
pattern isAccessible(TA:Train, SA:Segment)
{
  Train.currentlyOn(TA, SB);
  find linked+(SA,SB);
}
```

3. fejezet

Lokális kereső algoritmus kiterjesztése

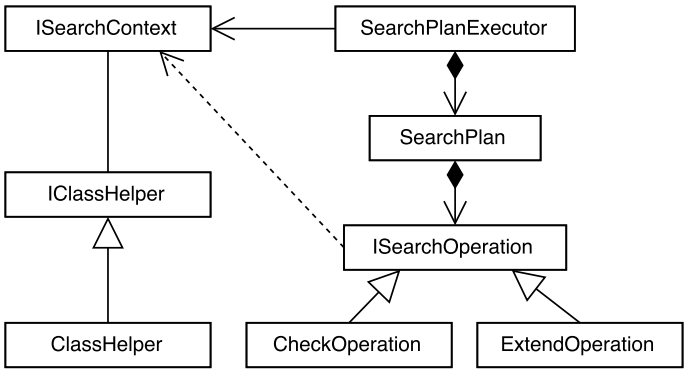
Munkánk során felhasználtunk egy korábbi, tanszéken fejlesztett prototípust, a VIATRA C++-alapú megoldás kiegészítését [4], amely egy lokális kereső algoritmust támogató eszközt ad gráflekérdezések hatékony futtatására.

Az eszköz a VQL (2.4. szakasz) változatos nyelvi elemkészletének csak egy részéhez nyújtott támogatást. Az elemkészlet ezen része – útvonal-kifejezés, típuskényszerek és a gráfmintára nem illeszkedés kényszere – önmagában nem elégséges ahhoz, hogy a lekérdezőrendszer a kiberfizikai rendszerek kontextusában is használható legyen. Emiatt munkánk első célja volt a nyelvi támogatás kiterjesztése. Ebben a fejezetben bemutatjuk a bővítés során támogatottá vált nyelvi elemeket és a megvalósított algoritmusokat.

Munkánk elméleti alapjául egy másik korábbi tanszéki kutatás szolgált [3], amelyet azonban sok helyen a környezeti sajátosságoknak megfelelően módosítani és átalakítani kellett.

3.1. Architektúra

Az alábbiakban áttekintjük az architektúrát, amely az alapját képezte a fejlesztéseknek.



3.1. ábra. Futásidejű könyvtár osztálydiagram részlete

A 3.1. ábrán a keresési terv (`SearchPlan`) tárolja a keresési műveleteket és azoknak egy előre definiált sorrendjét. A keresési terv végrehajtó (`SearchPlanExecutor`) a létrehozásakor paraméterként átadott keresési tervet hajtja végre. Egy segédosztály (`ClassHelper`) hivatott tárolni a végrehajtás során szükséges adatokat a metamodellből, az osztályok közti öröklési hierarchiát. A keresési művelet interfésze (`ISearchOperation`) definiálja a lokális kereső algoritmusban futó mélységi bejáráshoz szükséges közös, de mű-

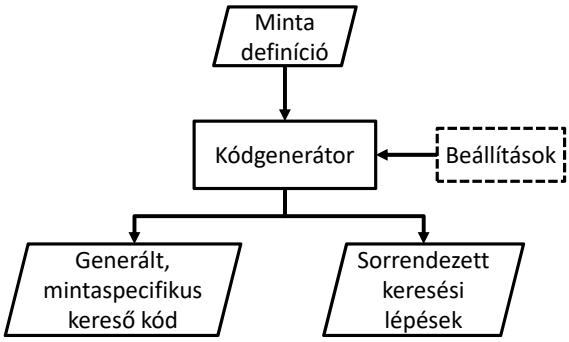
veletspecifikus függvényeket. A keresési művelet interfésze függ a segédosztálytól, mivel keresés közben a műveletek lekérdezhetik a metamodell tulajdonságait. Az ellenőrző művelet (`CheckOperation`) az absztrakt ősosztálya minden ellenőrzés típusú műveletnek. A kiterjesztő művelet (`ExtendOperation`) az absztrakt ősosztálya minden kiterjesztési műveletnek.

3.2. Modell-lekérdező eszköz C++ implementációja

A kódgenerálás során a mérnök számára két beállítás áll rendelkezésre. Mindkét megoldás a keresési terven alapul, de bizonyos szempontokban eltérnek egymástól. Az egyik egy fájlba generálja le a keresési algoritmust, a másik emellett felhasznál egy külső könyvtárat a futásához:

1. A *sorrendezett keresési lépések* a 3.1. alfejezetben bemutatott architektúrára épülő moduláris könyvtárat használja fel. A keresési terv műveleteinek megfelelő osztályok külön-külön megjelennek a generált kódban.

2. A *generált mintaspecifikus kereső kód* generálása során egy monolitikus függvényt kapunk, mely egymásba ágyazott ciklusokból és feltételekből áll. A fejlesztők számára érthetőbb kódot eredményez a mintaspecifikus kód, mivel nem kell ismerni a moduláris könyvtár működését, de komplexebb minta esetén a pontos működés visszafejtése vagy a generált kód módosítása nem triviális.



3.2. ábra. Kódgenerálás módjai

A következőkben bemutatott kiterjesztést mindkét kódgenerálási beállítással támogatjuk.

3.3. A VIATRA lekérdezőnyelvének kiterjesztése

A következőkben a 2.4. alfejezetben bevezetett nyelvi elemeket soroljuk fel, bemutatjuk melyek támogatásával egészítettük ki a keretrendszert és kitérünk azok implementációs nehézségeire és megoldásaira.

- A keretrendszerben már támogatott nyelvi elemek az alábbiak:
  - Útvonalkifejezés
  - Típuskényszer
  - Gráfmintára nem illeszkedés vizsgálata
- Nyelvi elemek, amelyek támogatásával kiegészítettük a keretrendszert:

  - Ellenőrző kifejezés
  - Konstans értékkényszer
  - Egyenlőtlenség vizsgálat
  - Gráfmintára illeszkedő eredmények számának vizsgálata
  - Tranzitív lezárt
- Nem implementáltuk az alábbi nyelvi elemeket:
  - Kiértékelő kifejezés

A következőkben részletesen tároljuk azon elemeket, amelyek támogatásával bővítettük a keretrendszert.

3.3.1. Ellenőrző kifejezés

A VQL az ellenőrző kifejezések (*check expression*) definíciójára az Xbase nyelvet [13] használja. Az Xbase kifejezései a Java nyelv egy szűk részhalmazát engedélyezik. Az Xbase nyelvi elemkészlete és erős Java specifikussága miatt nem triviális az Xbase kifejezések kiértékelése C++ környezetben, így a következő lehetőségeket vettük számításba:

- Megoldást jelentene a problémára, ha készítenénk egy Xbase kifejezés kiértékelő komponenst C++ nyelven, így feldolgozva az ellenőrző kifejezésben megadott eldöntendő kérdést. Sajnos azonban a VIATRA rendszer Xbase kifejezéseket kiértékelő moduljából futásidőben nem nyerhető ki információ a kifejezésre vonatkozóan.
- Egy másik megoldás lehetne, ha az ellenőrző kifejezésben megkötnénk, hogy csak olyan Xbase nyelvi elemekből építhetjük fel a kifejezést, amelyek C++-ban is értelmesek. Például számokból és aritmetikai vagy logikai operátorokból. Azért nem választottuk ezt a megoldást, mert ekkor vagy a fejlesztőkörnyezet szintaxis ellenőrző komponensét kellett volna módosítani vagy a kódgenerátorban kellett volna a kifejezést ellenőrzésnek alávetni.

Az implementáció során egy harmadik, kompromisszumos megoldás mellett döntöttünk: ha olyan gráfmintából generálunk kódot, ami tartalmaz ellenőrző kifejezést, akkor a generált kódban egy megadott helyen meg kell adni a logikai kifejezést C++ nyelven. Azért választottuk ezt a megoldást, mert ebben az esetben nem csökkentjük a nyelvi elem kifejezőerejét. Ezzel a megoldással nem kényszerülünk a már meglévő mintadefiníciók átírására, nem módosítjuk a VIATRA keretrendszerben már megszokott működést és támogatjuk bármilyen C++ specifikus könyvtár felhasználását a ellenőrző kifejezésben.

3.3.2. Konstans értékkényszer

A mintában hivatkozott konstans típusától függően egy (C++-ban értelmezhető) kifejezést állítunk elő, amit statikusan a keresési terv megfelelő műveletébe generálunk. A megoldásunk egyszerre támogat egész, valós, logikai érték, felsorolt típus és szöveg konstansokat.

3.3.3. Gráfmintára illeszkedő eredmények számának vizsgálata

Ennek a nyelvi elemnek megfelelő mintaillesztési művelet lehet ellenőrzés vagy kiterjesztés. Abban az esetben, ha *ellenőrzés*, akkor az aktuális keretben már van egy egész szám, aminek egyenlőnek kell lennie az illeszkedő eredmények számával. Ha *kiterjesztés*, akkor az illeszkedő eredmények számát egy megadott keretbeli változó értékébe kell írni.

3.3.4. Egyenlőtlenség vizsgálat

A megoldás egy egyszerű logikai műveletre vezettük vissza. Mivel a keretben tárolt mutató egyértelműen azonosítja az elemet elegendő ezek egyezőségét vizsgálni.

3.3.5. Tranzitív lezárt

A megoldásunk egy már létező algoritmus C++ implementációja [12].

4. fejezet

Szenzoradat modellalapú integrációja

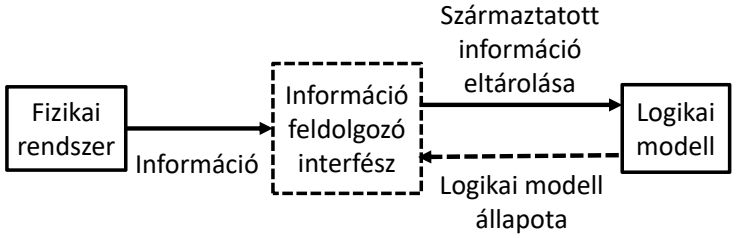
Munkánk célja az volt, hogy lehetővé tegye a szenzoradatok modellalapú feldolgozását kiberfizikai rendszerek számára. A szenzoradatok a valóságból érkező fizikai jellemzőkre vonatkozó értékek, amelyeket szeretnénk olyan formában reprezentálni, hogy azon logikai ellenőrzést tudjunk végezni. Ehhez javasolunk egy megközelítést, amely során egy gráfmintával reprezentált leképezés alapján olyan komponenst készítünk, amely a fizikai-logikai modell leképezést végrehajtja. Ezt a módszert mutatjuk be a fejezetben.

4.1. Koncepció

Ahhoz hogy a szenzoradatokon modellalapú ellenőrzéseket hajthassunk végre, fontos a fizikai világból származó adatok modellbeli integrációja. Ehhez szükséges egyrészt egy olyan nyelv, amellyel megadható a szenzoradatok modellre való leképzése, másrészt azok az algoritmusok, amelyek ezt a transzformációt elvégzik.

Tegyük fel, hogy szenzorokat helyezünk el (pl. kamerákat, sebességmérőket) az objektumainkon vagy a rendszerünk közelében. F olyamatosan érkező adatokból szeretnénk objektumok követését megvalósítani és ezek alapján dönteni a beavatkozásról. Szükség van az adatok feldolgozására és olyan formára hozására, amelyet már integrálni tudunk a modellünkbe, továbbá ezután hatékonyan tudjunk ellenőrzéseket végezni.

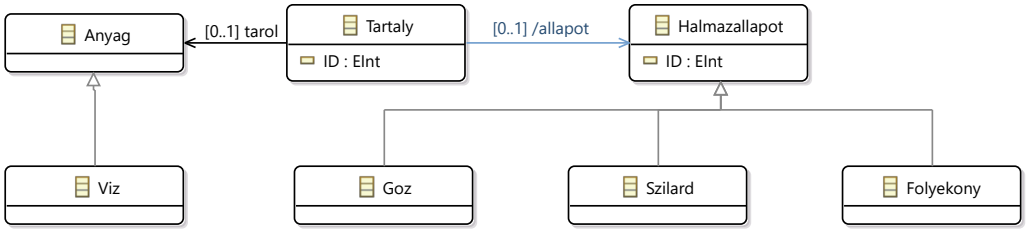
Egy megközelítést vázol fel a 4.1. ábra. A bejövő adatokat előfeldolgozásnak vetjük alá, majd csak származtatott információkat mentjük el a logikai modellbe. Csak a származtatott adatokat tárolva a modell méretét is csökkenteni tudjuk, amelyhez kevesebb memóriára van szükség. Ez különösen előnyös, ha olyan eszközökön (pl. beágyazott rendszereken) fut a futásidejű ellenőrző komponens, amelyen kevés erőforrás áll rendelkezésünkre.



4.1. ábra. Fizikai rendszer és logikai modell csatlakozása

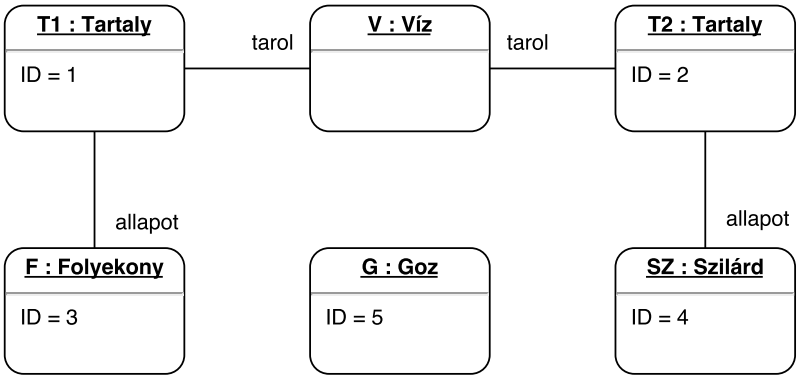
4.2. Példa

Ebben a fejezetben bevezetünk egy példát, amin szemléltetjük a szenzoradat modellalapú integrációját. Első lépés egy olyan metamodell elkészítése, amelyben megjelennek a fizikai világból származó információk és azok logikai leképzésük. A 4.2. ábrán látható osztálydiagram kémiai anyagokat tároló tartályokat modellez. Ezekről a tartályokról számon tartjuk, hogy mit tárolnak és a bennük tárolt anyag milyen halmazállapotú.



4.2. ábra. Példa metamodellje

A 4.3. ábra egy lehetséges modelljét ábrázolja az előbbi metamodellnek, két tartályt ábrázol, melyekben vizet tárolunk, az egyikben tárolt víz halmazállapota folyékony, a másikban szilárd.



4.3. ábra. Egy lehetséges példánymodellje a 4.2. ábrán látható metamodellnek

4.3. Szenzoradat integrációjának VQL nyelvű támogatása

Ebben a fejezetben bemutatjuk, hogy hogyan használhatjuk fel a VQL nyelv bizonyos elemeit a szenzor integráció hatékony leírására, továbbá bemutatjuk a leképzést, amely elkészít a modellünkhöz egy olyan interfészt, amely támogatja az egyszerű szenzoradat integrációt.

A származtatott érték tárolására ki kell jelölnünk egy metamodellbeli asszociációt. Erre a metamodellező nyelv ad támogatást a származtatott (`derived`) tulajdonság segítségével [10]. Majd ennek az asszociációnak a nevével kell hivatkoznunk, amikor a származtatott érték számítását kívánjuk leírni. Ennek a jelölésésre a `QueryBasedFeature` nyelvi elemet választottuk.

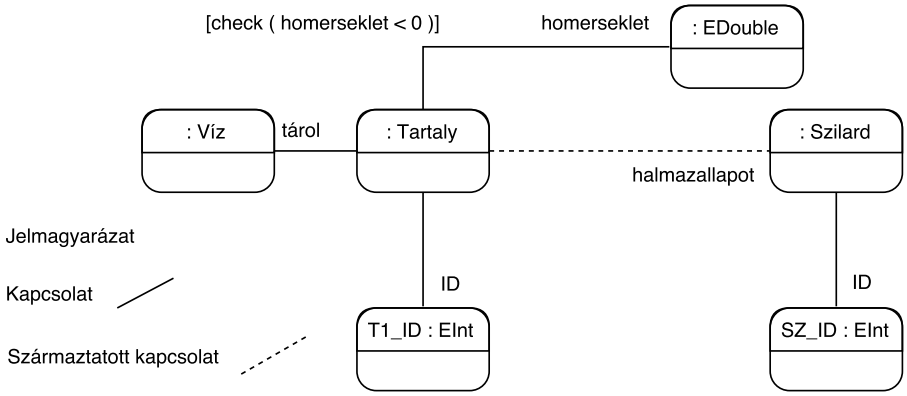
Az interfész bemeneti paramétereit is meg kell jelölnünk a származtatott érték számításának leírása során. Erre a VIATRA lekérdezőnyelvének `Bind` eleme ad támogatást, ezzel megjelölt változókra hivatkozhatunk a mintadefiníció törzsében. Az interfész generálás során ezek lesznek a bemenő paraméterek, azaz a szenzoradatok.

Az alábbi mintadefiníciót a 4.2. ábrán található metamodell elemein fogalmaz meg egy szenzoradat feldolgozó interfészt.

```
import "http://gyar.ecore/tartalyok"
import "http://www.eclipse.org/emf/2002/Ecore"

@Bind(parameters={tartalyID, allapotID, homerseklet})
@QueryBasedFeature(feature = "halmazallapot")
pattern szilard(tartaly : Tartaly, tartalyID : EInt, hAllapot : Szilard, allapotID : EInt,
    homerseklet : EDouble){
  Víz(víz);
  Tartaly.tarol(tartaly, víz);
  Tartaly.ID(tartaly, tartalyID);
  Szilard.ID(hAllapot, allapotID);
  check(homerseklet < 0);
}
```

A 4.4. ábrán látható gráfminta grafikus reprezentációjában. A `halmazallapot` származtatott értéket szeretnénk karbantartani a `homerseklet` szenzoradat értékének függvényében.



4.4. ábra. Gráfminta grafikus reprezentációja

4.4. Működés

Most bemutatjuk a szenzoradat integrációját a modellbe, kitérve a belső működésre is egy esettanulmány segítségével. A cél az, hogy modellalapon definiáljuk a fizikai adatok integrációját a modellbe majd pedig ez alapján olyan interfészt generáljunk, amelyet könnyű integrálni a meglévő rendszerekhez.

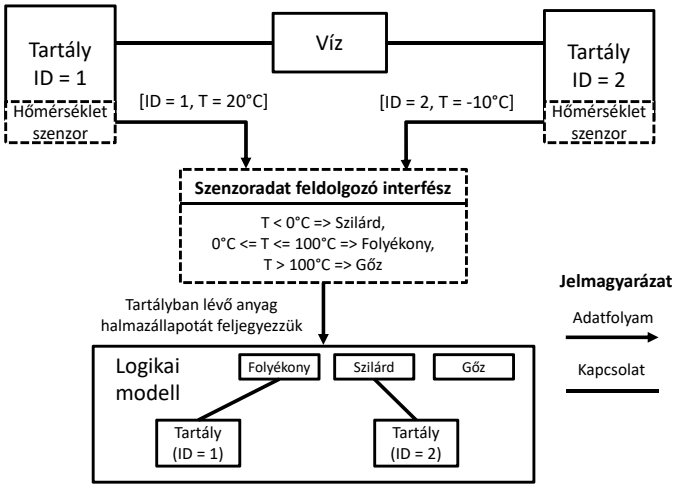
A 4.5. ábra ezt az integrációs folyamatot mutatja be a származtatott érték számítás működését az esettanulmány segítségével, ahol a fizikai rendszer tulajdonságait képezzük le a logikai modellünkre.

A szenzoradat feldolgozó interfész működése függ a tartályban tárolt anyagtól, esetünkben a víz hőtani jellemzőivel van felparaméterezve.

Az alábbi példa a fentieket használja fel egy szenzorinterfész definiálására.

```
import "http://gyar.ecore/tartalyok"
import "http://www.eclipse.org/emf/2002/Ecore"

@Bind(parameters={tartalyID, allapotID, homerseklet})
@QueryBasedFeature(feature = "halmazallapot")
pattern szilard(tartaly : Tartaly, tartalyID : EInt, hAllapot : Szilard, allapotID : EInt,
    homerseklet : EDouble){
  Víz(víz);
  Tartaly.tarol(tartaly, víz);
  Tartaly.ID(tartaly, tartalyID);
  Szilard.ID(hAllapot, allapotID);
  check(homerseklet < 0);
}
```



4.5. ábra. Szenzor adatfeldolgozás példa

Ez leírja a szilárd halmazállapotú tartályokból érkező információ leképzését a logikai modellre, ezek alapján kell elkészítenünk a többi halmazállapotra vonatkozót interfész-definíciót. Ezekből kódot generálva kapunk három statikus függvényt és keresési tervet. Mivel a mintadefiníció tartalmaz ellenőrző kifejezést (`check(homerseklet < 0)`), ezért a keresési terveket tartalmazó forráskódállományban definiálni kell a mintaillesztési lépés predikátumát.

```
searchPlan.add_operation(...
  [](double homerseklet){
    return homerseklet < 0;
  },
  ...
```

Az ellenőrző kifejezés a generált kódban egy logikai visszatérési értékű lambda kifejezésnek felel meg, melyben megfogalmazott függvény visszatérési értékét kell megadnunk. Miután ezeket a predikátumokat megadtuk a rendszerünk kész a működésre.

Minden tartálynak van egy hőmérsékletmérő szenzora, amely egy adatfolyamon keresztül továbbítja a hozzá csatlakozó tartály azonosítóját és a mért értéket. A szenzor adatokat kicsomagolva azokat átadjuk a statikus függvényeknek.

```
const int szilardID = 1;
const int folyekonyID = 2;
const int gozID = 3;

int tartalyID;
double homerseklet;
string szenzorInfo; // Pl.: 10;31.41592 (tartalyID;homerseklet)

FeldolgozHomersekletAdat(szenzorInfo, tartalyID, homersekletID);

SzilardAPIInputUpdater::update(tartalyID, szilardID, homerseklet);
FolyekonyAPIInputUpdater::update(tartalyID, folyekonyID, homerseklet);
GozAPIInputUpdater::update(tartalyID, gozID, homerseklet);
```

A *FeldolgozHomersekletAdat* függvény egyszerűen kicsomagolja a szenzorinformációban tárolt adatot. A példánkban ezt egy karakter tömb tárolja, ennél hatékonyabb megoldást is lehetne alkalmazni Egy származtatott érték számítása során a következő lépések futnak le sorban, ha bármelyik keresés vagy lekérdezés eredménye üres halmaz, akkor a származtatott érték nem frissül:

- Kikeresi a forrásobjektumot az egyedi azonosítója alapján. Esetünkben azt a váltót, amire a szenzoradat vonatkozik.

- Kikeresi a célobjektumot. Esetünkben az interfész által számított halmazállapotot reprezentáló objektumot egyedi azonosítója alapján.
- Lekérdezi a gráfmintára illeszkedő eredményeket a modellből.
- Beállítja a származtatott értéket a megfelelő halmazállapotra.

Fontos, hogy a származtatott érték számítás alatt ne fusson lekérdezés a modellen, mivel ez meghiúsíthatja a működést. A mostani megoldásunk során a lekérdezés futása közben iterátorokat tárolunk el a modellelem tárolókra. Ezen tárolók tartalmának változása esetén ezek az iterátorok érvénytelenné válnak és onnantól a lekérdezés futása nem determinisztikus.

5. fejezet

Lekérdezések végrehajtása elosztott modelleken

A rendszert leíró példánymodellt elosztva tároljuk, hiszen a modell egyes részeinek frissítéséért más-más csomópont felelős, tehát indokolt az ellenőrző komponenseket minél közelebb tenni az információ forrásához, a szükséges lekérdezéseket pedig elosztottan futtatni. Ebben a fejezetben megmutatjuk, hogy milyen módon tároljuk az elosztott modelleket és hogy hogyan valósítottuk meg az elosztott lekérdezéskiértékelést.

5.1. Modellek elosztott kezelése

Egy lehetséges megoldás volna, ha minden csomóponton tároljuk a teljes modellt. Ekkor a konzisztencia megtartásához szinkronban kellene tartanunk minden csomóponton tárolt modellt egymással.

Az általunk fejlesztett rendszerben egy modellelemet egy adott csomópont kezel, a többi csomópont csak másodlagos másolatot tárol, amellyel nem képes kezelni az objektumot, de a segítségével referenciát tudunk tárolni, ezt a proxy tervezési minta alapján készítettük el [5]. Ezen megoldás mentén kevesebb tárhely szükséges, mintha mindenhol tárolnánk a teljes modell és nincs szükség a csomópontok közti szinkronizációra, amíg betartjuk, hogy egy modellelemet csak egy adott csomópont kezel.

Minden modellelem egyedi azonosítóval rendelkezik, amely globálisan azonosítja a modellelemet az elosztott rendszerben. Így a rendszerek közötti kommunikáció során egy modellelemre mutató referenciát az egyedi azonosítója által tudunk megtalálni.

A modell felépítéséhez felhasználunk egy kezdeti statikus képet, amely leírja a kiinduló állapotot. Ezt a statikus képet egy konfigurációs állomány tartalmazza, aminek értelmezése a metamodellel együtt lehetséges. A konfigurációs állománynak tartalmaznia kell minden modellelemhez egyedi azonosítót, valamint megadhatóak a kapcsolatok és az attribútumok kezdeti állapotai.

5.2. Elosztott lekérdezés-végrehajtás

A lokális kereső algoritmus elosztott megvalósításához ki kell egészíteni az algoritmust, hogy az képes legyen elosztott modell esetén is megtalálni az adott mintára illeszkedő eredményeket, ugyanakkor ne adjon vissza olyan eredményt, amely nem illeszkedik a mintára.

Ahhoz hogy a fenti feltételek teljesülhessenek, először is meg kell vizsgálnunk, hogy mely keresési műveletek nem valósíthatóak meg helyben, egy csomóponton tárolt modellelemeken. A műveletek egy része, mint például az egyenlőség vagy egyenlőtlenség vizsgálat

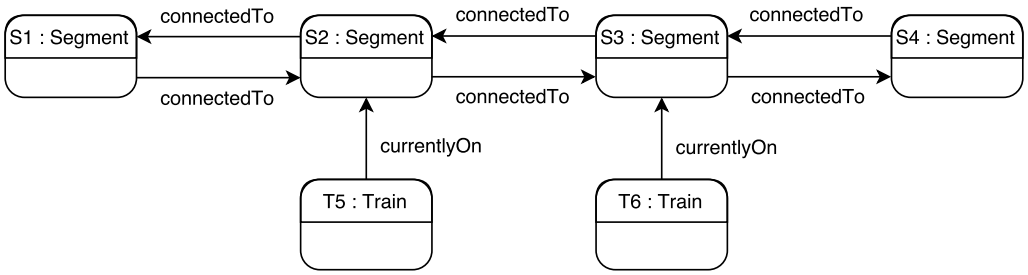
helyben is elvégezhetőek, hiszen a lekötött változókon ellenőriznek egyszerű feltételeket. Azonban vannak olyan műveletek, amelyekhez nem elég a helyben létező modellelemek ismerete. Például nem tudunk az összes Train elemen végigiterálni az 5.4. ábrán látható rendszer egy csomópontján, hiszen más csomóponton is helyezkedik el Train példány, amiről nem tudunk. Ezen műveleteket az alábbi lista tartalmazza:

- Ahhoz hogy egy osztály példányain végigiterálhassunk szükség van az összes csomópontra, hiszen a példányok bárhol elhelyezkedhetnek.
- Az asszociációk mentén való kiterjesztés nem lehetséges lokálisan, amennyiben a forrásobjektum vagy a célobjektum távoli csomóponton helyezkedik el.
- Ehhez hasonlóan azt sem tudjuk ellenőrizni, hogy egy már meglévő célobjektum felé van-e asszociációnk egy forrásobjektumból, ha a forrásobjektum nincs jelen a csomóponton.

Lokálisan nem mindig végezhető el az ellenőrző vagy kiterjesztő művelet. Ennek megfelelően ha szükséges, elosztjuk a lekérdezést a csomópontok között. Ezt a gyakorlatban új műveletek beszúrásával értük el, így expliciten meg van adva, hogy a keresés mely fázisában kell a többi csomópontra is továbbítani a keresést. Ez azért is fontos, mert bizonyos esetekben a lekérdezési tervtől függően elhagyható a művelet, például ha olyan objektumon keresztül kell navigálni, amely a lekérdezési terv adott fázisában garantálhatóan jelen van a csomóponton. Az algoritmus működését egy példán mutatjuk be a következő alfejezetben.

5.3. Példa egy lokális és egy elosztott keresésre

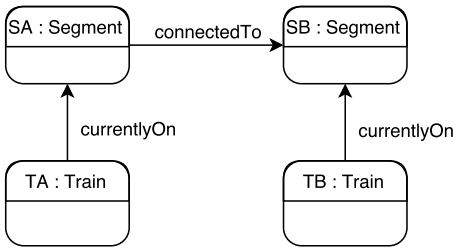
Ebben a fejezetben egy elosztott lekérdezés végrehajtását mutatjuk be. Ehhez egy példán keresztül először lokális, majd elosztott lekérdezési tervet készítünk és megmutatjuk, hogy milyen keresési fa bejárásával határozzák meg az eredményhamazt.



5.1. ábra. A példánymodell, amelyen a *lokális* vagyis elosztás nélküli verziójú lefutást szemléltetjük.

Legyen adott az 5.1. ábrán látható példánymodell, amely a már korábban példaként hozott metamodellre illeszkedik. Ebben keressünk olyan vonatpárokat, amelyek szomszédos szegmensen vannak, azaz keressük az 5.2. ábrán szemléltetett mintára illeszkedő részgráfokat. A mintát így írhatjuk le a VIATRA lekérdező nyelvén:

```
pattern NeighboringTrain(TA, TB)
{
  Train(TA);
  Train.currentlyOn(TA, SA);
  Segment.connectedTo(SA, SB);
  Train(TB);
  Train.currentlyOn(TB, SB);
}
```



5.2. ábra. A keresett gráfminta grafikus ábrázolása

Ha az erre illeszkedő részgráfokat lokális kereső algoritmus segítségével szeretnénk megkapni az egyik lehetséges lekérdezési terv a következő:

A) Iteráció az összes Train elemen (TA)

B) A TA currentlyOn referenciáján való navigáció (SA)

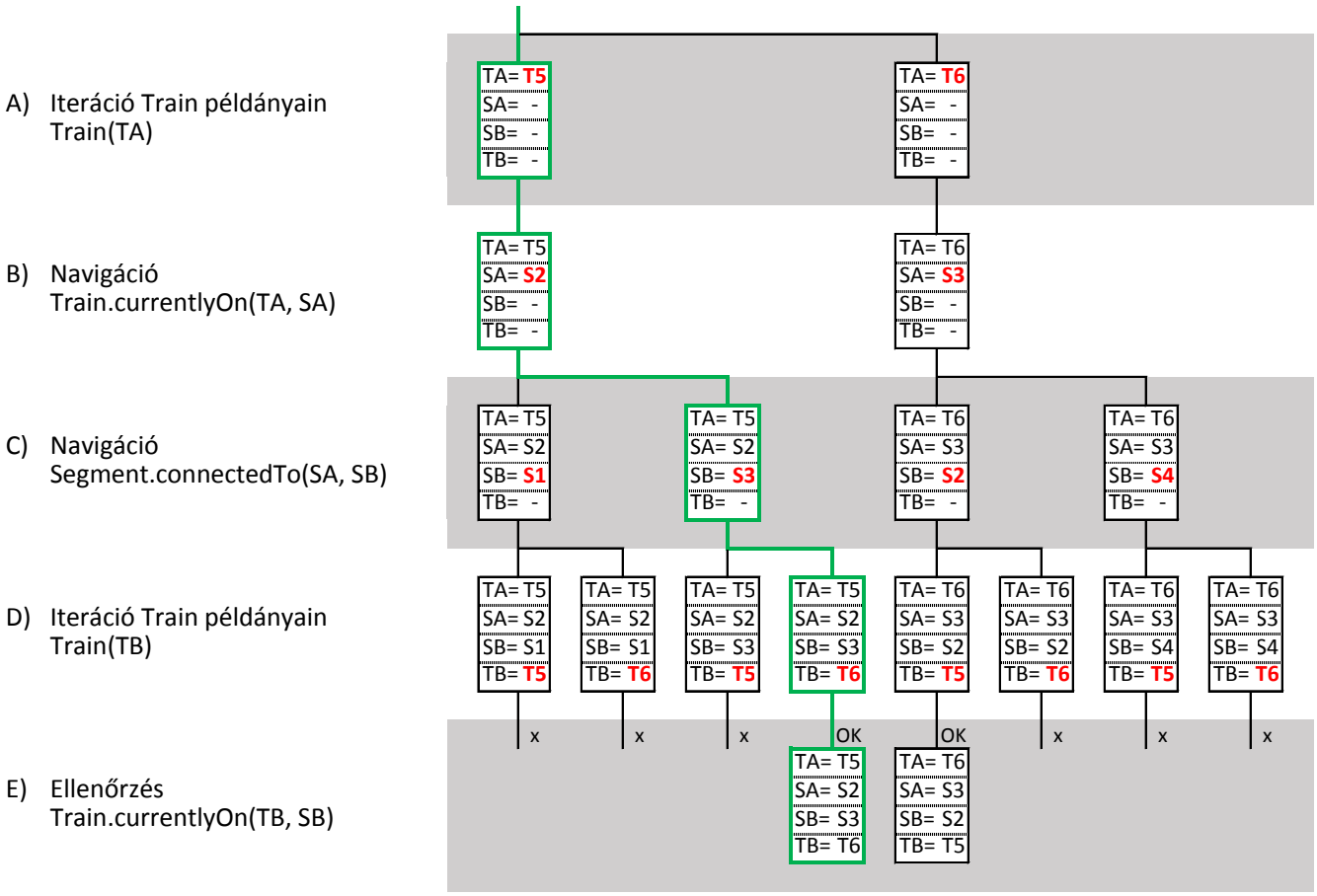
C) Az SA connectedTo referenciáján való navigáció (SB)

D) Iteráció az összes Train elemen (TB)

E) Ellenőrzés, hogy van-e TB-ből currentlyOn referencia SB felé

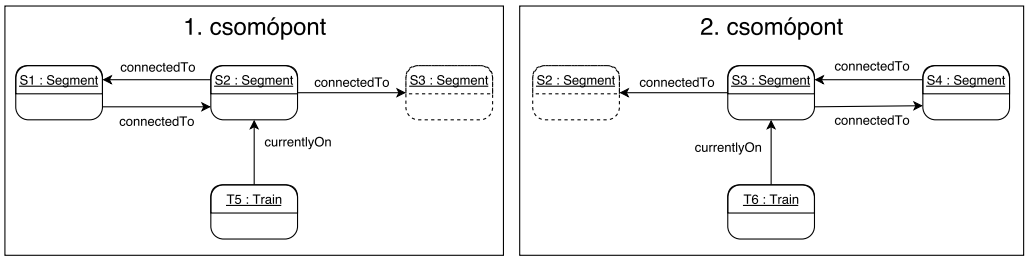
Ezen lekérdezési terv végrehajtása során az 5.3 ábrán látható keresési fát járjuk be. Az egyes sorok reprezentálják a keresési terv műveleteit, a téglalap reprezentálja a keretet, melyben tároljuk a minta változóinak aktuális lekötését. A vízszintes elágazások azt mutatják be, hogy egy keresés az adott ponton több további lekötéssel is folytatható. A keresési terv végrehajtása során egy mélységi kereső algoritmus járja be ezt a keresési fát.

Vegyük például a zölddel jelölt ág kiértékelését. Kezdetben üres keretből indulunk ki, azaz semmelyik változó sincs lekötve. Az első művelet (A) során végigiterálunk a Train példányain, azaz a T5-ön és a T6-on. Először TA=T5 lekötéssel dolgozunk tovább.A következő művelet egy navigáció (B), a Train currentlyOn referenciáján keresztül. Mivel TA=T5 currentlyOn referenciája az S2-be vezet, ezért S2-re kötjük le az SA értékét. A következő navigáció (C) elvégézése során a connectedTo referencia az SA=S2 elemből két helyre is mehet, S1-be vagy S3-ba. Az SB=S3 lekötése után elvégezhetjük a következő keresési műveletet(D), amely a T5-re majd a T6-ra köti le a TB értékét. Ha a T6-ra kötjük le, akkor a következő keresési művelet (E) az ellenőrzés során azt fogja találni, hogy a TB=T6 ból induló currentlyOn referencia elvezet az SB=S3-ba, tehát a kényszer teljesül. (Ellenben, ha például T5-re kötnénk le, nem teljesülne.)



5.3. ábra. Keresési fa az alap lekérdezési tervhez. Bal oldalon látható hogy az adott keresési lépés mely kényszer teljesülését eredményezi

Most, hogy láttuk a lokálisan futó esetet, tekintsük azt az esetet, amikor elosztott modellen kell megtalálni az egyezéseket.



5.4. ábra. Az 5.1. ábrán található gráf több rendszeren tárolva, proxykkal jelezve a más csomóponton tárolt modellelemet.

Mivel nem minden egyezés található meg csak lokálisan, ezért a lokális lekérdezési tervből az alábbi konstruálható, amely már alkalmas az elosztott egyezéseket is megtalálni:

1) **Keret továbbítása minden csomópontnak**
2) Iteráció az összes lokális Train elemen (TA)
3) A TA onSection referenciáján való navigáció (SA)
4) Az SA connectedTo referenciáján való navigáció (SB)
5) **Keret továbbítása minden csomópontnak**

6) Iteráció az összes Train elemen (TB)

7) Ellenőrzés, hogy van-e TB-ből onSection referencia SB felé

Az 5.5. ábrán egy elosztott lekérdezés lefutását figyelhetjük meg. Két csomópont szükséges a lekérdezés eredményhalmazának előállítására. A szaggatott kék vonal a két rendszer közötti kommunikációt jelzi. Észrevehető, hogy a keresési fa ugyanazon kereteken keresztül jut el az illeszkedő keretekig.

5.4. Az elosztott lekérdezésfuttató felépítése

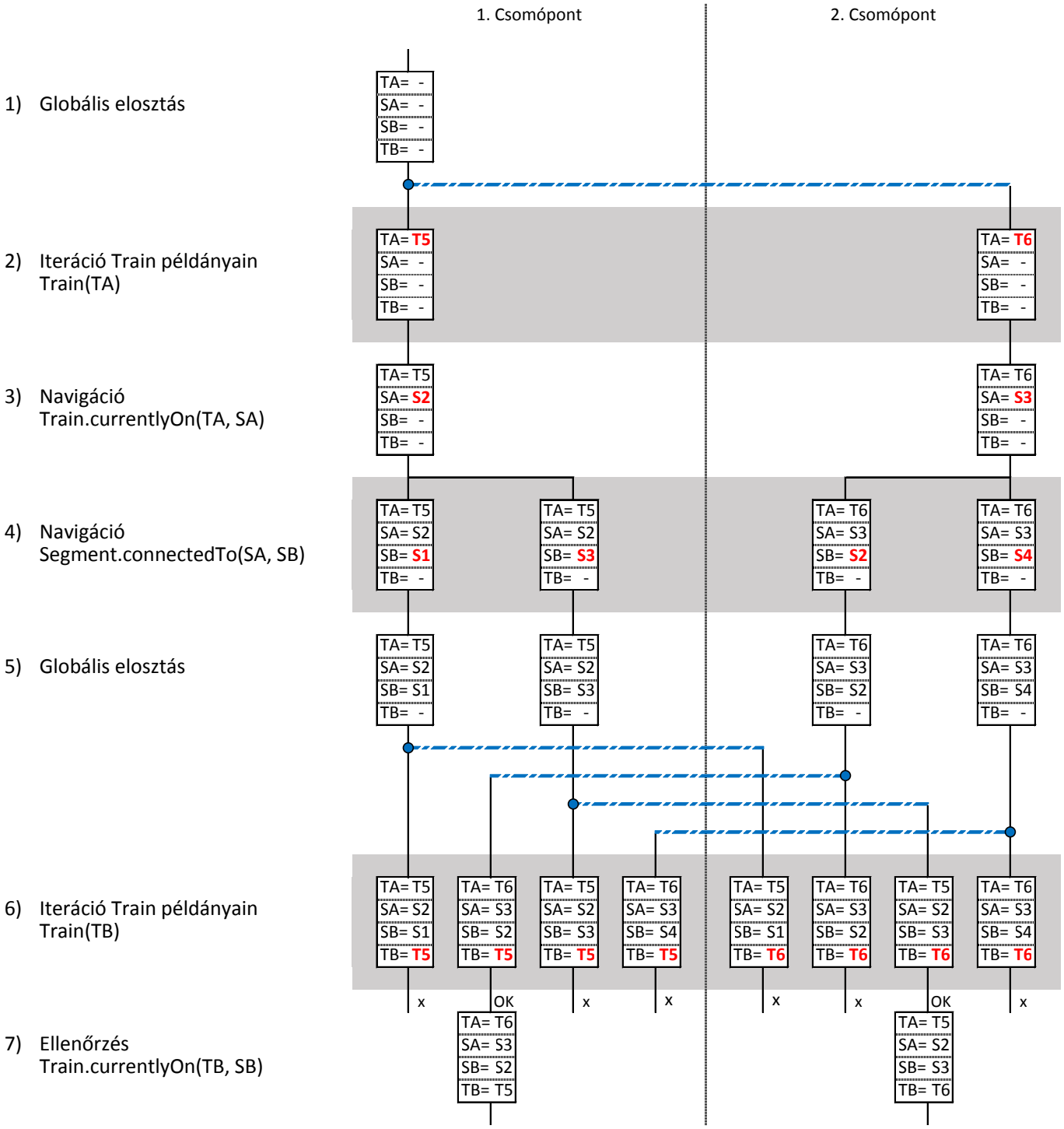
Ahhoz, hogy a kódgenerátor által előállított lekérdezéseket elosztottan futtathassuk, készítettünk egy keretrendszert, amely lehetővé teszi a csomópontok közötti kommunikációt, valamint a generált lekérdezésekhez való egységes hozzáférést. Ennek célja, hogy egy csomópont ne csak a lokálisan indított lekérdezés futtatására legyen képes, hanem máshol indított tetszőleges lekérdezésekhez kapcsolódó feladatok végrehajtására is.

A rendszer vázlatos architektúrája az elosztottság szempontjából az 5.6. ábrán látható módon épül fel. Minden csomópont megvalósít egy szolgáltatást, amely képes egy lekérdezés részfeladatát végrehajtani. Ehhez különböző lekérdezés végrehajtó komponenseket használ, amely egységes interfészt biztosít a generált lekérdezések keresési tervének eléréséhez. A generált lekérdezések képesek az elosztott modell lokálisan tárolt részén végrehajtani egy részlekérdezést.

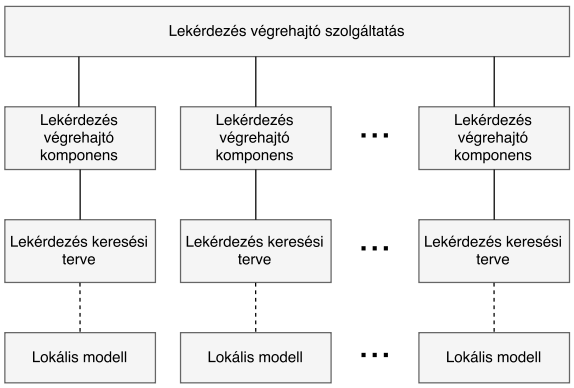
5.5. Az elosztott lekérdezésfuttató megvalósítása

Ahhoz hogy a rendszert megvalósítsuk, szükséges, hogy a kódgenerátoron is módosításokat végezzünk. Ezért minden generált lekérdezéshez egy egyedi azonosítót rendeltünk, ami egyértelműen azonosítani képes a lekérdezést. Ezen kívül generáltunk egy ún. `QueryRunnerFactory` osztályt. Ez tartalmaz egy gyártófüggvényt, amely képes azonosító alapján a lekérdezésfuttató (`QueryRunner`) objektumot gyártani, ami az adott lekérdezés végrehajtását végzi.

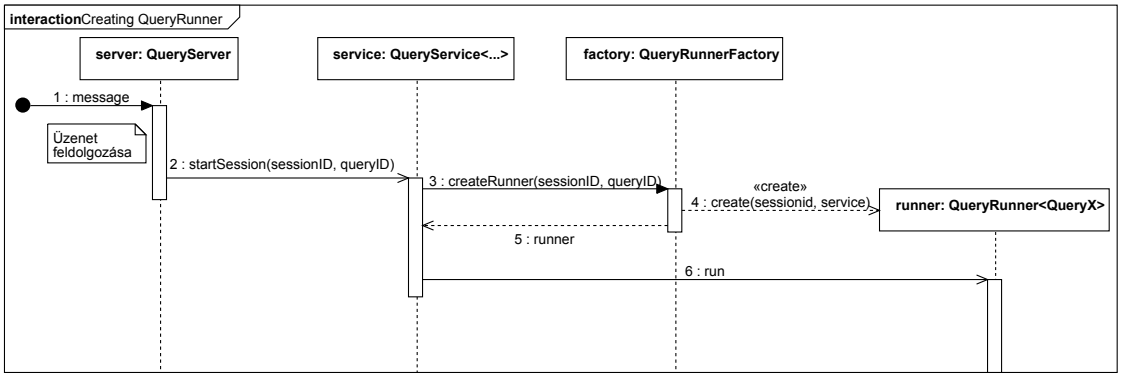
A megvalósított rendszer osztálydiagramjának egyszerűsített változata az 5.8 ábrán látható. A generált kódok sárgával, a sablon osztályok zölddel vannak jelölve. A szolgáltatás megvalósításáért két osztály felel. Az egyik, a `QueryServiceBase` lehetővé teszi a többi csomópontnak a lekérdezésfuttatáshoz szükséges szolgáltatások elérését (például lekérdezésfuttatók létrehozása a különböző csomópontokban, lekérdezési részfeladat megoldása, stb). A `QueryService` ennek egy olyan leszármazottja, amely már a generált kód alapján képes ezen szolgáltatásokat megvalósítani. A `QueryServiceBase`-hez kapcsolódik egy `QueryServer`, amely a szolgáltatás hálózati kiszolgálásáért felel (ld. 5.7. ábra, ahol a lekérdezésfuttatáshoz szükséges `QueryRunner` létrehozási kérésének kiszolgálását szemléltetjük). A `QueryServer` a bejövő kérésekhez egy `Request` objektumot hoz létre és továbbítja a `QueryServiceBase`-nek, amely ehhez létrehoz egy `TaskInfo` objektumot. Ez reprezentálja a végrehajtandó feladatot, ehhez tartozik, hogy melyik kérésre kell az eredményt továbbítani (`Request`), a rendszer hova gyűjti a lekérdezés eredményét (`ResultCollectorBase`), illetve egy Future objektum, amelyen keresztül a lokálisan indított lekérdezés eredménye elérhető (lsd. Active Object tervezési minta [8]). A `QueryService` a többi olyan csomópontról is tárolja az elérésükhöz szükséges információkat, amelyek a modell valamely részét tárolják, hiszen a lekérdezés végrehajtásához ezen csomópontok is szükségesek. Ezekhez egy-egy `QueryClient` objektum tartozik, amelyen keresztül elérhetőek a távoli csomópontok.



5.5. ábra. Keresési fa az elosztott lekérdezési tervhez. Bal oldalon látható hogy az adott keresési lépés mely kényszer teljesülését eredményezi, kékkel a csomópontok közötti kommunikációt jelöljük.



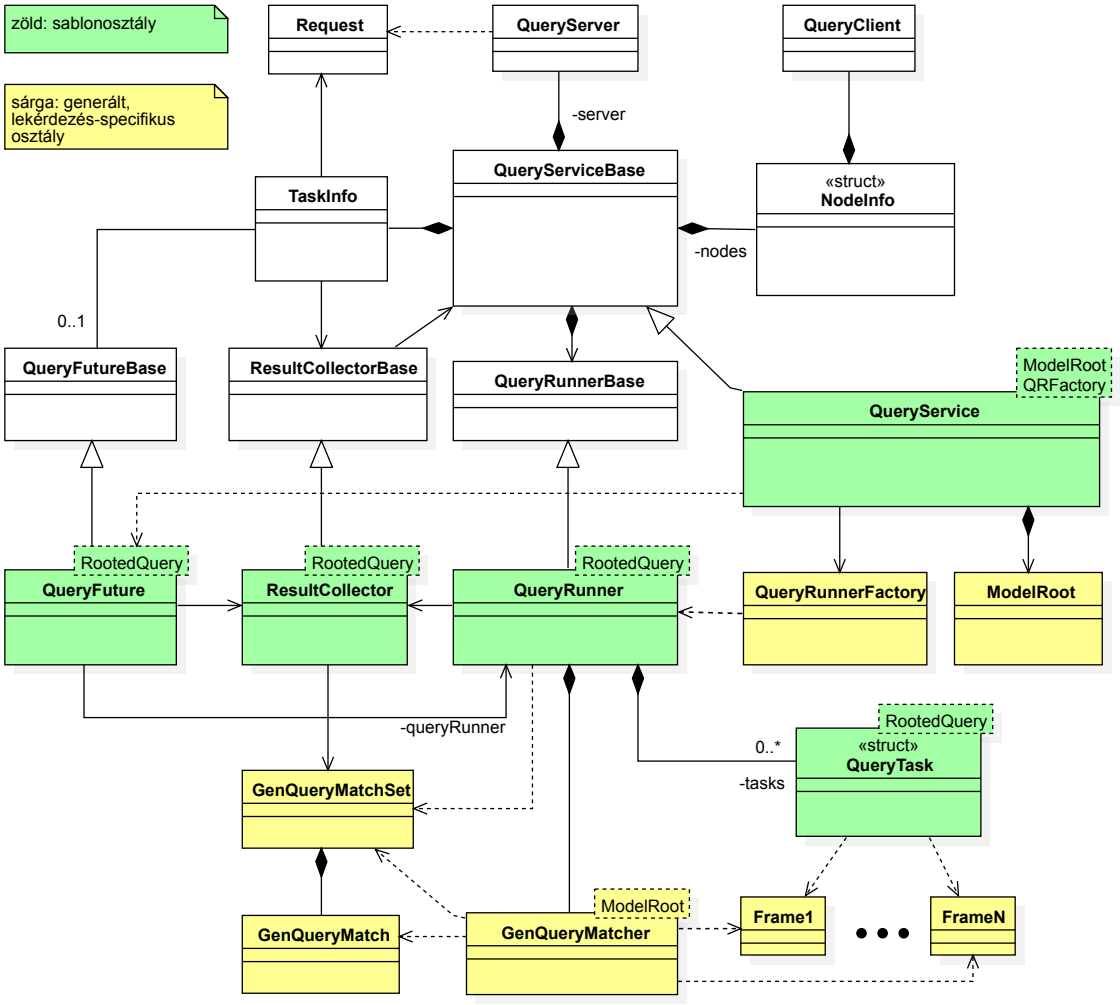
5.6. ábra. A rendszer felépítése vázlatosan



5.7. ábra. Egy kérés kiszolgálásának folyamata

A `QueryService` sablonosztályhoz tartozik egy `ModelRoot`, ami a lokális modellhez biztosít elérést, valamint egy `QueryRunnerFactory`, amely elengedhetetlen, hogy konkrét QueryRunnert hozhassunk létre lekérdezés-azonosító alapján. Mivel ez a kettő osztály generált, gráfminta-specifikus osztály, ezért ezek sablonparaméterei az osztálynak.

A `QueryRunner` a keresés végrehajtásához a generált Matcher (`GenQueryMatcher`) osztály példányán keresztül hajtja végre a keresési műveleteket. Ez a keresés eredményhalmazát egy generált halmaztípussal (`GenQueryMatchSet`) adja meg. Az eredményt a runner átadja a keresési feladathoz tartozó `ResultCollector` objektumnak.



5.8. ábra. Osztályok struktúrája metódusok és attribútumok nélkül

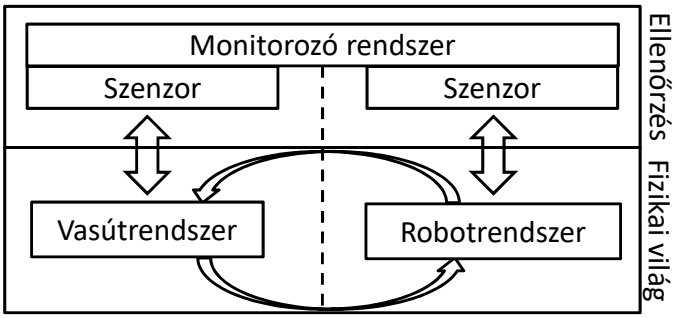
6. fejezet

Esettanulmány

Egy létező kiberfizikai rendszer ellenőrzését tűzzük ki célul az előző fejezetekben bemutatott elosztott lekérdezés végrehajtás és szenzoradat integráció segítségével. Egy mérnöki tervezési folyamatot mutatunk be egy probléma megoldására ezen eszközök segítségével.

6.1. MoDeS3 modellvasútrendszer

A MoDeS3 modellvasútrendszer egy modellalapon fejlesztett biztonságkritikus, autonóm kiberfizikai rendszer [7]. A valóságban a vasúthálózat elemei akár több száz kilométerre is lehetnek egymástól,emiatt elosztott rendszerként kell kezelnünk. Minden elemhez tartozik egy irányító áramkör, ami vezérli a helyi rendszert. A rendszerünk három fő komponensből áll, ahogy azt a 6.1. ábra szemlélteti.



6.1. ábra. MoDeS3 vasútrendszer sematikus felépítése

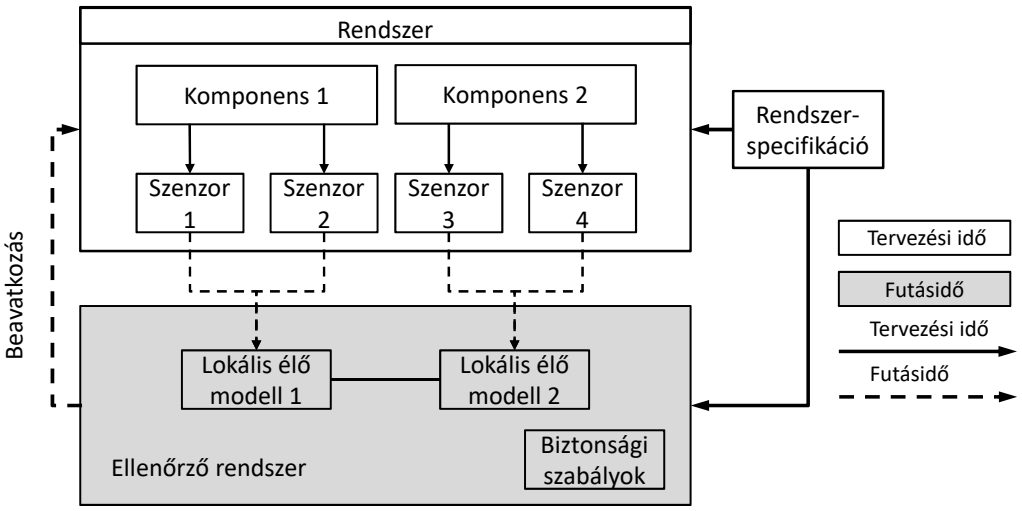
A vasútrendszer foglalja magában a különböző vasúti elemeket, közlekedési eszközöket, azok működéséhez szükséges elemeket, azaz mindent, ami egy hétköznapi vasúthálózat része. A robotrendszer vasúti állomáson működő rakodórendszer elemeit és azok működéséhez szükséges elemeket foglalja magában. Így egy komplex, autonóm rendszert kapunk, amelyben megtalálható minden ahhoz, hogy a fejlesztéseink számára jó esettanulmány legyen.

Tervezés időben nehéz problémát jelent, hogy a vonatok ne ütközzenek egymással. Az autonóm működés, a magas fokú elosztottság, a dinamikusan változó rendszer egyesével is nagy kihívást jelentenek. Ezért azt javasoljuk, hogy gráflekérdezés-alapú futásidejű módszerrel garantáljuk a biztonságkritikus követelmények teljesülését.

6.2. Megközelítés

A tervezés egy rendszerspecifikációval indul, ami leírja a mérnök számára a célokat, amiket a megvalósuló rendszernek teljesítenie kell. A specifikáció segítségével a mérnök megtervez egy rendszert, modellezi az elemeket és formalizálja a VIATRA nyelvi elemeivel a specifikációban megadott biztonsági szabályokat, amik betartásával biztosítható a rendszer helyes működése.

A rendszer futása során az ellenőrző rendszer komponensei a szenzorokból bejövő információval folyamatosan frissítik az élő modellt. Az ellenőrző rendszer a rendszerspecifikációból formalizált biztonsági szabályokat ellenőrzi az elosztott rendszeren.



6.2. ábra. A tervezési és futási folyamat menete

Azt, hogy a rendszert felépítő elemek mely komponensbe kerüljenek, a mérnök tervezési idejű feladata.

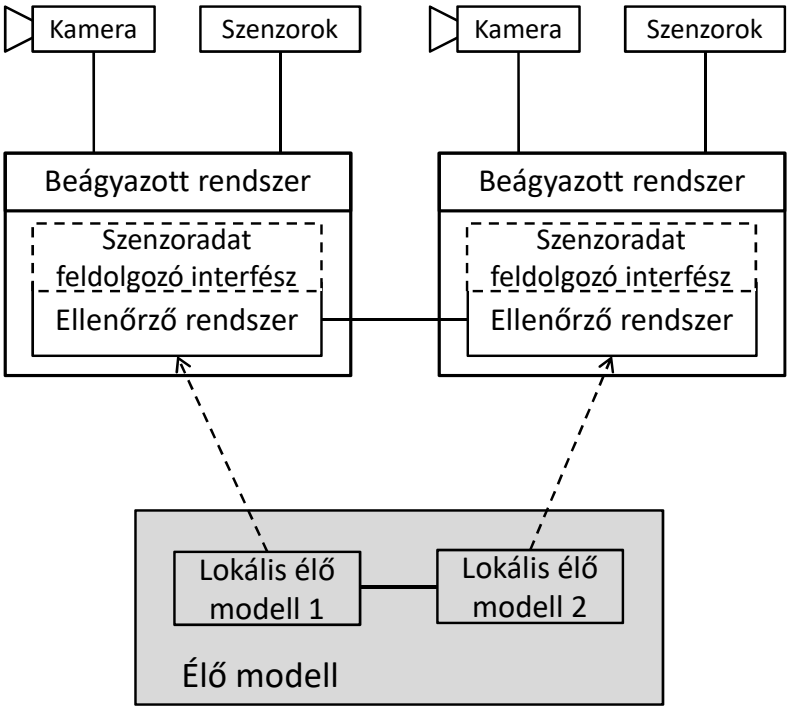
6.3. Probléma és megoldás

Ebben az alfejezetben bemutatunk egy komplex problémát, majd felvázolunk egy lehetséges megoldást az előző fejezetekben bemutatott módszerekkel.

A vasútrendszerben számos váltó berendezés üzemel, amik ki vannak téve az időjárás viszontagságainak. Ezeknek a váltóknak van egy üzemhőmérsékletük, melyen megbízható a működésük. Bizonyos hőmérséklet alatt befagyhatnak ezek a váltók, ezzel akár veszélyeztetve a közlekedést. Szeretnénk futásidejű módszerekkel vizsgálni a váltók működését és jelezni, ha egy vonat egy befagyott váltóhoz közeledik.

A vonatok pozíciójának követését egy valós vasúti rendszerben megoldhatjuk pl. GPS-szel. A modellvasút esetében ezt a vasútrendszer feletti kamerával, számítógépes képfeldolgozás segítségével helyettesítjük.

A 6.3. ábrán láthatjuk a vasútrendszer egy lehetséges futásidejű elosztását. Két részre osztjuk vasúti elemeinket a térbeli elhelyezkedés alapján, majd mindkét részhez hozzárendelünk egy-egy ellenőrző egységet. Ezen ellenőrző rendszerek kezelik az elosztott modellt, futtatnak lokális vagy szolgálnak ki globális gráflekérdezéseket és a szenzorokból érkező adatokat feldolgozzák és azokból leszűrt következtetéseket elmentik a lokális modellbe.



6.3. ábra. A konkrét vasúti rendszer felépítése.

6.4. Működés

Az eszközünk működését az esettanulmányon megfogalmazott biztonsági kritériumokból felírt gráfmintákkal és azok futtatásával demonstráljuk. Emellett párhuzamosan szenzoradat feldolgozást végző rendszereket is üzembe helyeztünk minden csomóponton, amik a hozzájuk rendelt modellelemektől érkező adatokat leképzik logikai modelljükre.

A vasúthálózat 7 darab váltóból és 24 darab szegmensből épül fel. Két csomópontot telepítettünk a vasúthálózat bal és jobb oldalára és elosztottuk köztük a váltókat és szegmenseket. Minden elemhez csak egy csomópontot rendeltünk. A rendszerünk helyes működésének ellenőrzésére felírtuk az alábbi gráfmintát, amit elosztottan futtattunk.

```
pattern isDangerous(seg:Segment, turn:Turnout, tr:Train){
  Train.currentlyOn(tr, seg);
  Segment.connectedTo(seg, turn);
  Frozen(fro);
  Turnout.currentState(turn, fro);
}
```

A gráfminta egy olyan váltó, szegmens, vonat hármast keres, melyre igaz, hogy a vonat pillanatnyi helyzetét meghatározó szegmens közvetlen környezetében egy befagyott váltó található. Ha illeszkedést kapunk erre a mintára, akkor jelezhetünk az ellenőrző rendszeren keresztül a vonatnak, hogy álljon meg.

Az előző gráfmintában kihasználtuk a modellben tárolt `Turnout.currentState(...)` származtatott értéket. Ennek karbantartásához a váltókon elhelyezett hőmérsékletmérő szenzorból érkező adat feldolgozására van szükségünk. Az alábbi interfészdefiníciókat fogalmaztuk meg a szenzoradatok integrációja céljából.

```
@Bind(parameters={turnID, stateID, turnTemp})
@QueryBasedFeature(feature = "currentState")
pattern frozenStateAPI(turn:Turnout, turnID:EInt, fro:Frozen, stateID:EInt, turnTemp:
    EDouble){
  Turnout.id(turn, turnID);
  Frozen.id(fro, stateID);
  check(turnTemp < -20);
}

@Bind(parameters={turnID, stateID, turnTemp})
@QueryBasedFeature(feature = "currentState")
pattern operationalStateAPI(turn:Turnout, turnID:EInt, oper:Operational, stateID:EInt,
    turnTemp:EDouble){
  Turnout.id(turn, turnID);
  Operational.id(oper, stateID);
  check(turnTemp >= -20);
}
```

Ezen definíciókból generáltunk interfész implementációt és integráltuk a már meglévő rendszerünkbe. Minden csomóponton létrehoztunk helyi állapot objektumokat (`Frozen`, `Operational`) egyedi azonosítóval, amikkel paramétereztük a statikus függvényeket.

```
const int operationalID = 100;
const int frozenID = 101;
int turnoutId;
double turnoutTemp;
std::string info; // Pl.: 22;0.3141592 (turnoutID;turnoutTemp)

ParseTempInfo(info, turnoutId, turnoutTemp);

FrozenStateAPIInputUpdater::update(turnoutId, frozenID, turnoutTemp);
OperationalStateAPIInputUpdater::update(turnoutId, operationalID, turnoutTemp);
```

A `ParseTempInfo` egy olyan függvény mely egy karakterláncból álló szenzorinformációt részekre bont az interfész számára feldolgozható formában.

6.5. Eredmények

A 6.4. alfejezetben bemutatott rendszert felépítettük és kipróbáltuk. A futás során az elvárt eredményt produkálta. A szenzoradat integráció folytonosan zajlott, a lekérdezések lefutottak és megtalálták azon vonatokat, melyek veszélyes helyzetbe kerültek.

7. fejezet

Konklúzió

Dolgozatunkban kiberfizikai rendszerek futásidejű ellenőrzésével foglalkoztunk, célunk egy olyan keretrendszer készítése volt, amely kritikus, elosztottan futó, összetett kiberfizikai rendszerek ellenőrzését tudja hatékonyan támogatni.

7.1. Elért eredmények

Munkánk során először kiegészítettünk egy nyílt forráskódú, alacsony szintű programozási környezetben fejlesztett modell lekérdező keretrendszert, hogy a specifikációs minták (pl. a rendszer specifikációját leíró gráfminták) bővebb halmazát tudja támogatni, vizsgálni. Építve a keretrendszer adta gazdag nyelvi elemkészletre, kidolgoztunk egy módszert, amely során támogatni tudjuk egyszerűen használható API, azaz interfész generálását a fizikai világból érkező adatok modellbeli integrációjának támogatására. Ezt az interfészt használva egyszerűen kapcsolhatóak szenzor adatok a modellhez, továbbá az adatokat azonnal le is képezzük a logikai modell nyelvére.

Annak támogatására, hogy az információ forráshoz minél közelebb történjen meg az ellenőrzés, elosztott modelltároló és lekérdezéskiértékelő rendszert készítettünk. A keretrendszer támogatja, hogy a szenzorokhoz és a beavatkozókhoz minél közelebb, lokálisan értékeljük ki az információkat, így biztosítva a gyors beavatkozás lehetőségét.

Megközelítésünk alkalmazását egy esettanulmány segítségével mutatjuk be, amelyben egy elosztott, több rendszer kompozíciójaként előálló biztonságkritikus rendszer ellenőrzését végeztük el.

7.2. Gyakorlati eredmények

A megtervezett algoritmusokat és leképezéseket egy keretrendszerbe implementáltuk, felhasználva és kibővítve a korábban is rendelkezésre álló gráflekérdező keretrendszert. Integráltuk az új nyelvi elemek támogatását és a szenzorintegrációs interfészt generáló algoritmusokat. A keretrendszer modell tároló és gráflekérdező motorjának kibővítésével pedig egy elosztott, lokális keresés alapú, kis erőforrásigényű ellenőrző keretrendszert valósítottunk meg.

Az elkészített algoritmusokat és fejlesztéseket szisztematikusan teszteltük, hogy biztosítsuk azok helyességét.

7.3. Korlátozások és jövőbeli munka

Fejlesztéseink prototípusként állnak rendelkezésre, amelyet a jövőben is tervezünk bővíteni. Támogatni fogjuk bővebb nyelvi elemkészlet elosztott kiértékelését és jobb konfigu-

rálhatóságát. Tervezzük az allokáció modell alapú támogatását, amely segítségével akár a lekérdező rendszer átkonfigurálása is megoldható lenne.

A jövőben több kereső algoritmus implementálása és összehasonlítása is segítené a még hatékonyabb futásidejű ellenőrzés megvalósítását.

Köszönetnyilvánítás

Szeretnénk megköszönni konzulenseinknek Vörös Andrásnak, Búr Mártonnak és Szárnyas Gábornak, az iránymutatást és a közös munkát. A dolgozat elkészülését támogatta az IncQuery Labs Kft. és az MTA-BME Lendület Kiberfizikai Rendszerek Kutatócsoport.

Irodalomjegyzék

[1] Gábor Bergmann: Incremental model queries in model-driven design. PhD dissertation (Budapest University of Technology and Economics). Budapest, 2013.

[2] Gábor Bergmann – István Dávid – Ábel Hegedüs – Ákos Horváth – István Ráth – Zoltán Ujhelyi – Dániel Varró: VIATRA 3: A reactive model transformation platform. In *Theory and Practice of Model Transformations - 8th International Conference, ICMT 2015, Held as Part of STAF 2015. Proceedings* (konferenciaanyag). 2015, 101–110. p.

[3] Márton Búr: A general purpose local search-based pattern matching framework. Master's thesis (Budapest University of Technology and Economics). 2015.

[4] Róbert Dóczi: Search-based query evaluation over object hierarchies. Diplomaterv (Budapest University of Technology and Economics). 2016.

[5] Erich Gamma – Richard Helm – Ralph Johnson – John Vlissides: *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA, 1995, Addison-Wesley Longman Publishing Co., Inc. ISBN 0-201-63361-2.

[6] Hibatűrő Rendszerek Kutatócsoport: Kiberfizikai rendszerek. `https://inf.mit.bme.hu/edu/individual/taskgroups/cps`.

[7] Hibatűrő Rendszerek Kutatócsoport: Model-based demonstrator for smart and safe systems. `https://inf.mit.bme.hu/research/projects/modes3`.

[8] R. Greg Lavender – Douglas C. Schmidt: Active Object – An Object Behavioral Pattern for Concurrent Programming, 1995.

[9] Edward Ashford Lee – Sanjit Arunkumar Seshia: *Introduction to embedded systems: A cyber-physical systems approach*. 2011, Lee & Seshia.

[10] István Ráth – Ábel Hegedüs – Dániel Varró: *Derived Features for EMF by Integrating Advanced Model Queries*. Berlin, Heidelberg, 2012, Springer Berlin Heidelberg, 102–117. p. ISBN 978-3-642-31491-9.
URL `http://dx.doi.org/10.1007/978-3-642-31491-9_10`.

[11] Runtime Verification Conference: Introduction. `https://rv2016.imag.fr/`.

[12] Tamás Szabó: Transitive reachability for efficient event-driven model transformations. Master's thesis (Budapest University of Technology and Economics). 2012.

[13] Xtext Documentation: Xbase language reference. `https://www.eclipse.org/Xtext/documentation/305_xbase.html#xbase-language-ref-introduction`.