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

Adatelemzési folyamatok diagnosztikája és adatminőségérzékenység-elemzése

TDK-dolgozat

Készítette:

Szilvásy Noémi
Urbán Balázs

Konzulens:

Gönczy László
Kocsis Imre

2015.

Tartalomjegyzék

Kivonat

Napjainkban egyre több az informatikától és a statisztikától korábban jórészt független szakma alkalmaz adatelemzést. Az adattárolás és feldolgozás fajlagos költségének csökkenésével egyre nagyobb mennyiségű és egyre nagyobb sokféleségű adaton végeznek elemzéseket, sokszor olyan szakemberek, akiknek fő szakterülete az adatminőség értékeléstől és javítástól igen messze esik. Mindeközben a heterogén forrásból származó, jellemzően tisztítatlan adatok által jelentett kockázatok jelentősége nem csökkent. A feldolgozási és elemzési folyamat érzékeny lehet a bemeneti adatok hibáira, valamint a szakértői feltételezések ellenőrzését is igényli. Ezt a feladatot nehezíti, hogy az adatok szerkezete és adott esetben az adatok forrása is időben változhat.

Az adatok rendszerezése, tisztítása és elemzése ma már gyakran adatelemző munkafolyamatok segítségével történik, melyek kezelését grafikus eszközök (pl. Rapid Miner, Knime) támogatják az adatelemzésben kevésbé jártas felhasználóknak is.

Amennyiben a feldolgozás / adattisztítás lépései során nem sikerül kiküszöbölni az összes adathibát, akkor ezek torzíthatják az adatelemzési lépések (pl. interaktív vizuális analízis, statisztikai módszerek) kimenetét. Ugyanakkor ezek a hibák sokszor kivédhetőek további adattisztító és konzisztencia-ellenőrző lépések beiktatásával, amelyek megakadályozhatják a hibás értékek továbbterjedését az adathibákra érzékeny lépésekig.

A dolgozat keretein belül megterveztünk egy ontológia alapú metamodellt, mely általános adatfeldolgozó folyamatokat ír le. Létrehoztunk reprezentatív példa adattisztító és adatelemző folyamatokat egy erre alkalmas grafikus eszközben (Rapid Miner) és biztosítottuk a folyamatokból (ontológia alapú) példánymodellek generálását. A téma szakirodalmának tanulmányozása alapján megalkottunk egy adathibákat leíró taxonómiát. Az adathibák terjedését leíró szabályokat definiáltunk az adatfeldolgozási, -tisztítási és -elemzési folyamatok különböző típusú lépéseire, és megvizsgáltuk, hogy mely lépés mely adathibákra érzékeny és hogyan tehető robusztussá. Példát adtunk arra, hogy a vizsgált környezet modelljének ismerete hogyan segítheti az adatok konzisztencia- és teljsségellenőrzését.

Hibaterjedés alapú eszközt és módszert dolgoztunk ki, melyhez a fenti folyamatokból automatikusan komponens alapú hibaterjedési modelleket állítottunk elő. A vizsgálathoz megalkottunk egy általános komponens leíró modellt, amellyel más típusú rendszerek is leírhatóak. Megvalósítottunk egy eszközt, amely a generált modellen korlátkielégítési programozás alapú hibaterjedés vizsgálatot hajt végre, és képes felderíteni a lehetséges hibaokokat és jelenségeket a folyamatban, visszavezetve ezeket az eredeti modell szintjére. Az elkészült rendszerünk ezáltal képes rámutatni a folyamat azon lépéseire, ahol további ellenőrzésekre vagy adattisztításra van szükség.

A dolgozatban egy összetett felhő alapú alkalmazás teljesítmény és szolgáltatásbiztonsági mérési adatainak feldolgozásán és kezdeti elemzésén keresztül mutatjuk be módszerünk gyakorlati alkalmazhatóságát.

Eredményeink közvetlenül segíthetik adatelemzési projektek hatékony tervezését azáltal, hogy szisztematikus módon javaslatot teszünk bemeneti adatok és a köztes számítások hibáinak kiszűrésére a mért rendszer modelljének figyelembevételével. Ezzel időigényes és szakértői tudást igénylő munkát váltunk ki és segítjük, hogy az elemző a lényegi problémák felderítésére koncentráljon. A megközelítésünk független az analízis során használt eszközöktől.

Abstract

Data analysis is an important activity which is performed by a growing number of non-technical users as well.

Data cleaning is necessary in most in data analysis processes. A data analysis process can be very sensitive to faults in the input data and often needs human check by experts. The structure and the sources of the data may also change over time, which makinges data cleaning more difficult. In data analysis projects, data processing, data cleaning and data mining areis often implemented by workflows. Graphical data analysis workflow tools (eg. Rapid Miner, Knime) support these activities for non-technical users as well who may not have deep experience in data cleaning and analysis.

If the data errors were are not eliminated completely during data processing / data cleaning, these errors can seriously distort the output result of the actual data analysis steps (eg. interactive visual analysis, statistical methods). However, these errors can be "caught" during the processavoided by inserting additional data cleaning and consistency checks. This way the further propagation of data errors can be stopped.steps to avoid propagating the wrong value across the process to the sensitive steps.

In our paper we describe an ontology-based metamodel to represent general data analysis processes. We created data cleaning and analysis processes in a popular graphical tool (Rapid Miner) and implemented the automatic generation of (ontology) instance models. We created a data fault-error-failure taxonomy based on literature research. For the various types of data cleaning, analysis and processing operations, we defined error propagation rules and analysed the data quality sensitivity and robustness of the steps. We show how system models can help the analysis of data consistency and completeness.

We examined how methods of dependable computing can be used within this field. We developed a method and a tool for error propagation analysis of the above mentioned processes which are transformed to component based system model. We created a method and a tool for the error propagation analysis based on constraint programming. Potential faults and failures of data. management workflows are traced back to the originall modemodel so our system can reveal weak points in the process where data cleaning steps should be inserted to prevent failures caused by poor data quality. In this paper we also introduce the practical applicability of our method in the case study of performance and dependability analysis of metrics measured in a complex cloud based application.

Our results can help to perform data analysis project more effectively by giving recommendations on how to eliminate input data errors and the effects of wrong calculations, also considering the model of the system under analysis. These recommendations can save a lot of expert effort and help data analysts to concentrate on the investigation of the business problem. Our methodology is independent from the tools being used for analysis.

1. fejezet

Bevezető: Adathibák hatása az adatelemzésre

Napjainkban az adatelemzés egyre több, az informatikától és a statisztikától független szakmában megjelenik. Ezeken a területeken grafikus, folyamat alapú eszközökkel olyan szakemberek is könnyen létre tudnak hozni adatelemző folyamatokat, akik nem jártasak az adatanalízisben. Ugyanakkor, ezek a feldolgozási és elemzési folyamatok érzékenyek a bemeneti adatok hibáira. Az adathibákat kiszűrésével sok elemzői munkát és tévedést meg tudunk előzni.

Jelen fejezet egy minta adatelemző folyamat segítségével demonstrálja, hogy a folyamat bemeneti adatainak különböző típusú hibái minőségileg különböző hibákat tudnak okozni a folyamat kimenetén. Elemzési céltól és alkalmazási területtől függően, az egyes „kimeneti" hibakategóriák jelentősége természetesen más és más, de mindenképp igaz az, hogy ezek a hibák nem feltétlenül számszerűen, hanem minőségi kategóriákként is kezelhetőek. Például, ha egy adatelemző folyamat objektumkategóriák felett értelmezett statisztikai modelleket szolgáltat, akkor minőségi különbség van aközött, hogy egy vagy néhány modell parametrikusan enyhén hibás, illetve hogy teljes kategóriák modelljei hiányoznak.

Az úgynevezett kvalitatív hibamodellezésnek - amikor egy rendszer komponenseinek bemenő és kimenő hibáit a típusukkal jellemezzük - a biztonságkritikus és kritikus rendszerek tervezési gyakorlatában komoly szakirodalma van. Kvalitatív hibaterjedés modellezéssel ezeken a területeken jó hatékonysággal becsülhetőek egy rendszer „legrosszabb" viselkedési módusai. Dolgozatunk alapvető célja megmutatni, hogy a kvalitatív hibaterjedés-elemzés hatékony eszköz lehet az adatelemzési folyamatok bemeneti hibaérzékenységének jellemzésére és konstruktív javítására is.

Célunk egy olyan megközelítés és módszer kidolgozása volt, amely közvetlenül segítheti adatelemzési projektek hatékony tervezését azáltal, hogy rámutatunk a folyamat azon lépéseire, ahol további ellenőrzésekre vagy adattisztításra van szükség, és szisztematikus módon javaslatot tesz a bemeneti adatok és a köztes számítások hibáinak kiszűrésére a vizsgált rendszer modelljének figyelembevételével. Fontos volt, hogy a megközelítés független legyen az analízis során használt eszközöktől.

A megvalósítás során megterveztünk egy ontológia alapú metamodellt, mely általános adatfeldolgozó folyamatokat ír le. Biztosítottuk a folyamatokból (ontológia alapú) példánymodellek generálását. Megalkottunk egy adathibákat leíró taxonómiát és adathiba-terjedést leíró szabályokat definiáltunk az adatfeldolgozási, -tisztítási és -elemzési folyamatok különböző típusú lépéseire. Hibaterjedés alapú eszközt és

módszert dolgoztunk ki, melyhez a folyamatokból automatikusan komponens alapú hibaterjedési modelleket állítottunk elő. A vizsgálathoz megalkottunk egy általános komponens leíró modellt, amellyel más típusú rendszerek is leírhatóak. Megvalósítottunk egy eszközt, amely a generált modellen korlátkielégítési programozás alapú hibaterjedés vizsgálatot hajt végre, és képes felderíteni a lehetséges hibaokokat és jelenségeket a folyamatban, visszavezetve ezeket az eredeti modell szintjére. Az elkészült rendszerünk ezáltal képes rámutatni, hogy a folyamat belül hol van szükség további ellenőrző vagy adattisztító lépés beszúrására.

A 2. fejezetben motivációs példán keresztül szemléltetjük módszerünk jelentőségét, szerepét és fontosságát. A 3. fejezetben bemutatjuk azokat a szakirodalmakat és eszközöket, amelyek formálták szemléletünket a módszer kidolgozása során, valamint itt definiáljuk az általunk megalkotott adathibákat leíró taxonómiát. A 4. fejezetben bemutatjuk a megalkotott módszert, az 5. fejezetben pedig ismertetjük a megvalósított eszközöket. A kidolgozott módszer továbbfejlesztési lehetőségeit a 6. fejezet tekinti át. Végül munkánkat a 7. fejezetben foglaljuk össze.

2. fejezet

Motivációs mintapélda

Motivációs példánkat egy jelenleg is futó kutatási projektből (*Cloud Performance Management*) merítjük, ahol egy összetett, felhő alapú alkalmazás teljesítmény és szolgáltatásbiztonsági mérési adatainak kiértékelését végezzük. Ebben a projektben több adatelemző folyamat is felépítettünk a *RapidMinder* [32] eszközben (melyet később bemutatunk). Motivációs példánk számítógépklaszter terheléseloszlás vizualizációját állítja elő nyers bemeneti adatokból. (Ez esetben monitorozóeszköz által előállított CSV állományokból.) További folyamatokat is bemutatunk a 4.1. fejezetben.

2.1. Bemenő adatok forrása

A felhő alapú környezet jellemzője, hogy fizikai és virtuális gépekből épül fel, amelyek szolgáltatásokat nyújtanak. Ezeket a gépeket teljesítményszempontból terheljük és mérjük a *Cloud Performance Management* projekt keretein belül, majd az összegyűjtött mérési eredményeket kiértékeljük.

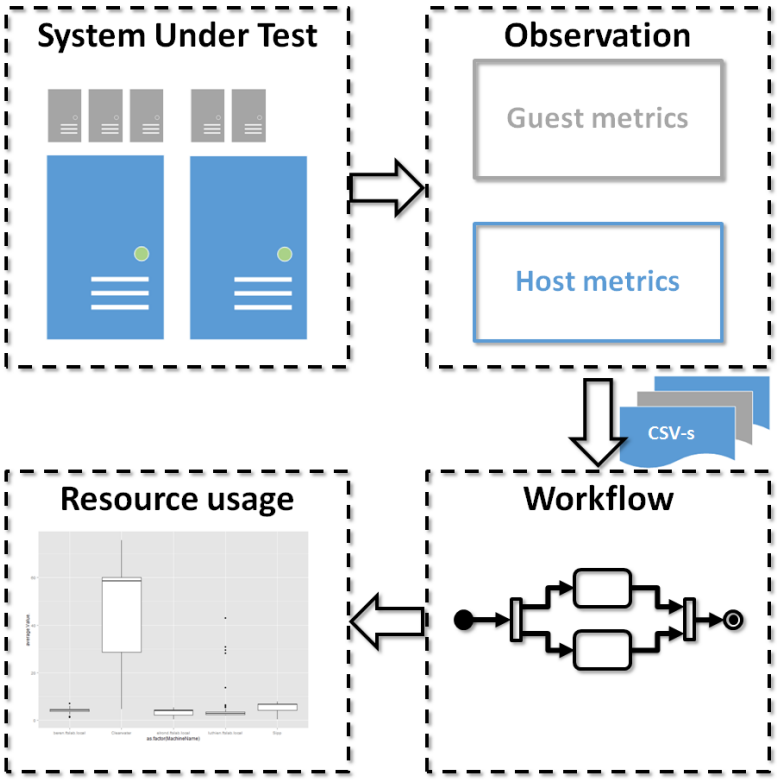
Az említett projekt egyik célja, hogy meghatározza egy felhőbe helyezett alkalmazás hogyan fog viselkedni a felhasználói terhelés és a rendszer konfigurációjának függvényében. Ennek meghatározásához fontos tudni azt, hogy az egyes gépek a mérések során hogyan működtek, azaz mi volt az összefüggés a felhasználók által generált terhelés és az erőforrások terheltsége között.

A mérési és méréskiértékelési folyamatot a 2.1. ábra szemlélteti. A fizikai és virtuális gépekről az *ESXi hypervisor* által periodikusan mért és riportolt teljesítményjellemzőkből adatfeldolgozó folyamat segítségével meghatározzuk az adatanalízis szempontjából fontos kimenetet.

2.2. Az adatelemző folyamat bemenete

A folyamat bemenete CSV állományokból áll, amelyek mérési eredményeket tartalmaznak. A CSV bemenetek felépítése a 2.2. ábrán látható.

A megfigyelésekhez tartozik egy időbélyeg (*Timestamp*), ami a mintavételezés időpontját rögzíti. Megjelenik a mintavételezett metrika (*MetricId*) a mért értékkel (*Value*) együtt. Látható emellett, hogy a gép milyen nevű erőforrását (*Instance*) mértük, hiszen egy gépen egy adott típusú erőforrásból (CPU, diszk, hálózati interfész) több is lehet, így azonosítani kell, hogy melyikhez tartozik az adott megfigyelés.



2.1. ábra. Mérésfeldolgozás menete

Timestamp	Value	MetricId	Instance
1429201380	0	disk.devicelatency.average	naa.50026b7246028893
1429201380	14	net.packetstx.summation	vmnic0
1429201380	14	disk.read.average	naa.50026b7246028893
1429201380	20	net.broadcastrx.summation	vmnic0
1429201380	0,18	cpu.usage.average	8
1429201380	0,01	cpu.usage.average	13

2.2. ábra. Mérési eredmények a folyamat bemenetén

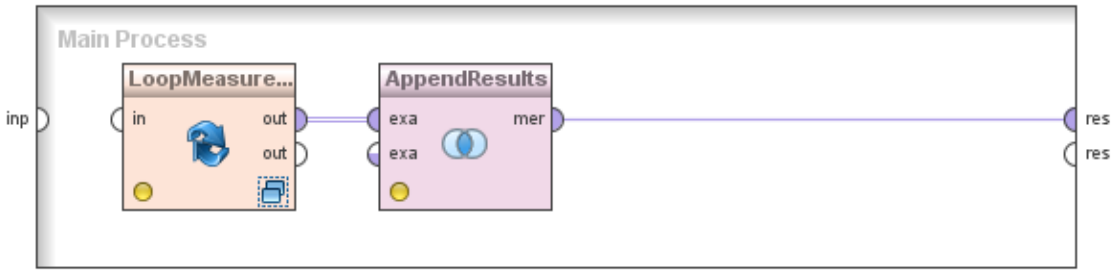
Annyi ilyen CSV állományunk van, ahány gépről rögzítettünk mérést, tehát minden géphez tartozó mérési eredmény külön állományban található. (Az állománynévből derül ki, hogy melyik gép méréseit tartalmazza, ez a mérési környezet sajátossága.)

2.3. Az adatelemző folyamat feladata

A következőkben ismertetett *RapidMiner* [32] folyamat célja, hogy gépenként megjelenítsük a CPU terhelésének eloszlását. Ennek egyik módja az, hogy minden géphez *boxplot* diagramon megjelenítjük a *cpu.usage.average* metrika átlagos értékét, amit az egy időpillanathoz tartozó megfigyelések átlagából számít. Egy gép esetében, ugyanahhoz az időpillanathoz és metrikanévhez akkor tartozik két megfigyelés, ha annak több példánya van az adott erőforrásból. Ebben az esetben átlagoljuk a különböző erőforrásokon mért értékeket. (Például több maggal rendelkező processzorra a magok időablakra vonatkozó kihasználtságának átlagát számoljuk.)

2.4. Az adatelemző folyamat felépítése

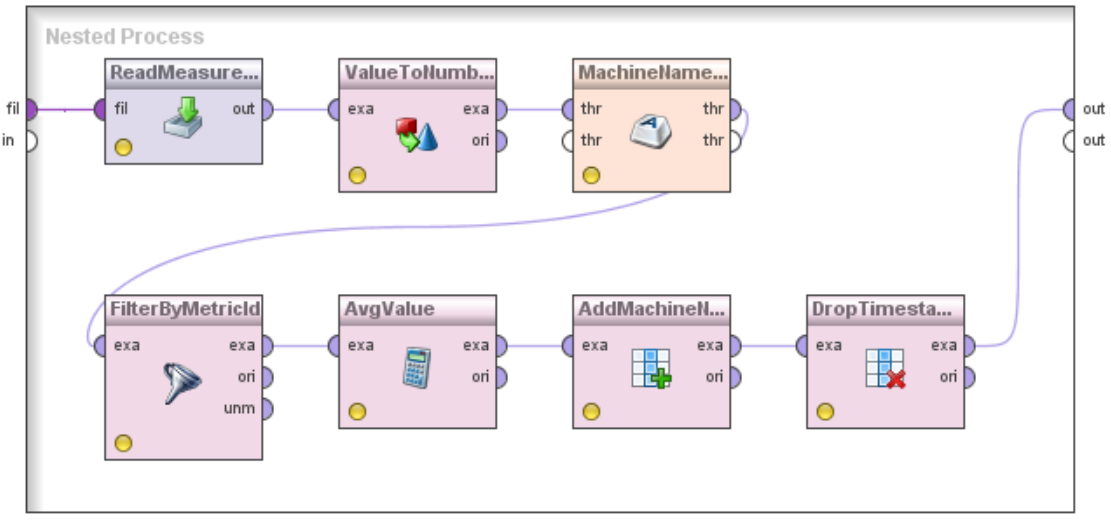
A hierarchikus adatelemző folyamat legfelső szintjét a 2.3. ábra szemlélteti.



2.3. ábra. A példafolyamat legfelső hierarchiaszintje

A legfelső hierarchiaszint a következő két lépésből áll.

- *LoopMeasurements*: Ciklussal végigmegy a paraméterként megadott könyvtár állományain, jelen esetben a különböző gépek mérési eredményeit leíró CSV állományokon. (A RapidMiner eszköz beépített *Loop Files* operátora.)
- *AppendResults*: Összefűzi a ciklus részeredményeit, azaz az egyes gépekhez tartozó eredményeket (amelyeket a 2.4. ábrán látható alfolyamat állít elő). (*Append* operátor)



2.4. ábra. LoopMeasurements alfolyamat

- *ReadMeasurementCSV*: Beolvassa a ciklus által aktuálisan meghatározott, mérési eredményeket tartalmazó CSV állományt. (*Read CSV* operátor)
- *ValueToNumber*: A szövegként tárolt, tizedesvesszővel (nem ponttal) tagolt értékeket számmá alakítja. (*Parse Numbers* operátor)
- *MachineNameMacro*: Az állománynévből kinyeri a gépnevet, és eltárolja. (*Generate Macro* operátor)

- *FilterByMetricId*: Azokra a megfigyelésekre szűr, amelyek metrikaneve (MetricId) megegyezik cpu.usage.average metrikával. (*Filter Examples* operátor)
- *AvgValue*: Időbélyegenként átlagolja a mért értékeket (Value). Ez végzi a különböző erőforrásokon (Instance) mért értékek átlagolását. (*Aggregate* operátor)
- *AddMachineName*: Hozzáad egy új oszlopot az adatszerkezethez. Az új oszlop a gépnevet tartalmazza, amelyet az operátor a korábban mentett macro értékéből nyer. (*Generate Attributes* operátor)
- *DropTimestamp*: Törli a Timestamp oszlopot az adatszerkezetből. (*Select Attributes* operátor)

2.5. Az adatelemző folyamat kimenete

A folyamat eredményeként egy két oszlopból álló táblázatot (a 2.5. ábra) kapunk. Az első leírja, hogy melyik géphez tartozik a második oszlopban rögzített átlagos érték.

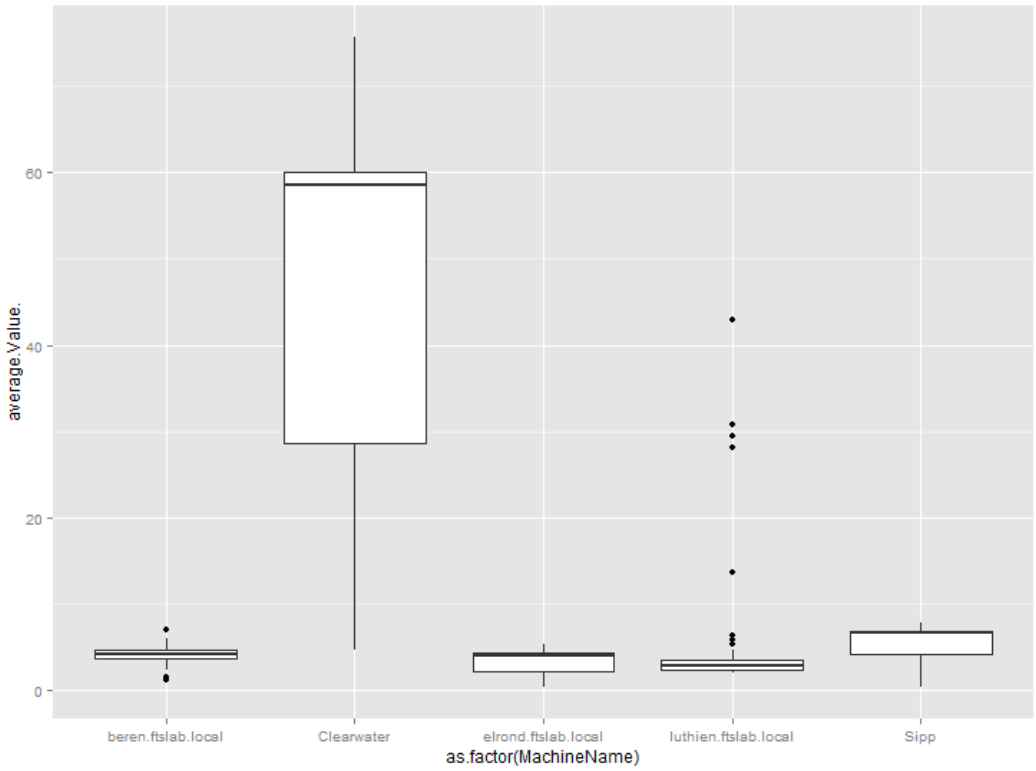
average(Value)	MachineName
2.620	beren.ftslab.local
1.198	beren.ftslab.local
2.600	beren.ftslab.local
15.910	Clearwater
13.470	Clearwater

2.5. ábra. A adatelemző folyamat kimenete

A 2.6. ábra jeleníti meg az egyes gépekhez tartozó diagramokat. Jól olvashatóak a minimum-maximum értékek, a medián és a kvartilisek. A boxplot diagram legfelső pontja jelöli a maximum értéket, a legalsó pedig a minimumot. A „doboz” belsejében lévő vízszintes vonal a mediánt, a „doboz” felső határa a $Q3$ kvartilis, az alsó határa pedig a $Q1$ kvartilis értékét jelzi. Jelen esetben, ez azt mutatja, hogy mekkora volt a cpu.usage.average metrika maximális/minimális értéke, mediánja. Ebből láthatjuk és összehasonlíthatjuk, hogy az egyes gépek mennyire voltak terheltek a mérések során. Az is leolvasható, hogy pl. a Clearwater gép esetében a CPU terhelés biztosan nem követi a normál eloszlást, hiszen a a mediánja nem a $(Q3 - Q1)/2$ határon helyezkedik el, azonban luthien.ftslab.local gép terhelése már szimmetrikus (de nem biztos, hogy normál) eloszlású, mivel teljesül rá, hogy mediánjának értéke $(Q3 - Q1)/2$.

2.6. Bemeneti adathibák és hatásuk

Motivációs céllal felsorolunk néhány ismert (és a projektmunka során is tapasztalt) bemeneti adathibatípust, melynek az adott folyamat és problématerület esetén ha-



2.6. ábra. Boxplot diagram

tása van a kimenetre. Az adathibák egy bő kategória-rendszerét a 3.2.2. fejezetben ismertetjük.

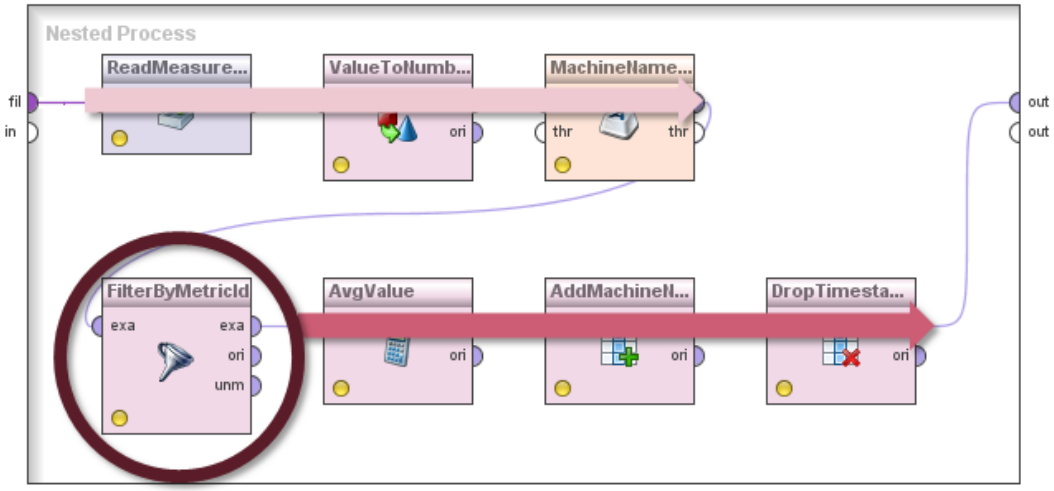
2.6.1. Elgépelés a metrikanévben

Előfordulhat, hogy a *cpu.usage.average* metrika más néven lett rögzítve a mérést vezérlő script hibája miatt. Például nem követtünk le egy időszakos változást a mért rendszerben. Ilyenkor a *FilterByMetricId* lépés (2.7. ábra) szűrőfeltételének nem felel meg egyik metrikanév sem, így az adott géphez tartozó box hiányozni fog a boxplot diagramról. Ugyanezt a hibát kapjuk a kimeneten, ha hiányzik minden olyan sor a mérési eredményekből, ami az adott metrikát rögzítette.

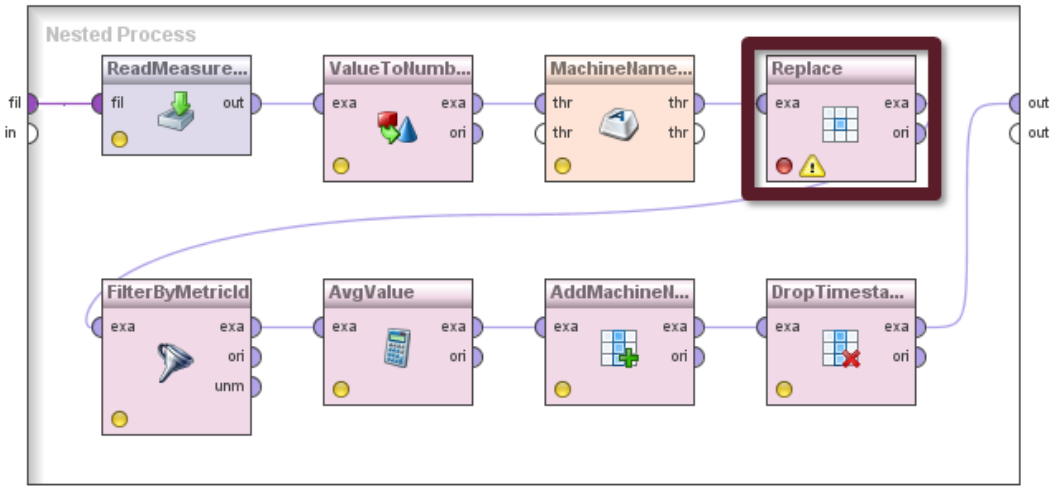
Amennyiben a hibás metrikanév elgépelésből adódott, felderíthető a hiba egy metrikanév szótárnak történő megfeleltetéssel. Ha tudjuk, hogy mit mire szeretnénk javítani, akkor egy *Replace* vagy egy *Replace (Dictionary)* operátor beiktatásával kiküszöbölhetjük az adathiba kimenetre gyakorolt hatását.

2.6.2. Hiányzó időbélyeg

Elképzelhető, hogy hiányzik egy időbélyeg az egyik megfigyelésnél (2.9. ábra). Ennek az lesz a következménye, hogy az időbélyegenkénti átlagszámításnál (2.10. ábra) megjelenik egy üres cellához tartozó átlag. Noha a Timestamp oszlopot a folyamat egy későbbi lépésénél eldobjuk, az „üres” időbélyeghez tartozó plusz átlag megjelenik a kimeneten. Ez akár módosíthatja is a minimum/maximum értéket (2.11. ábra), hiszen nem vettük figyelembe az átlagszámítás során.



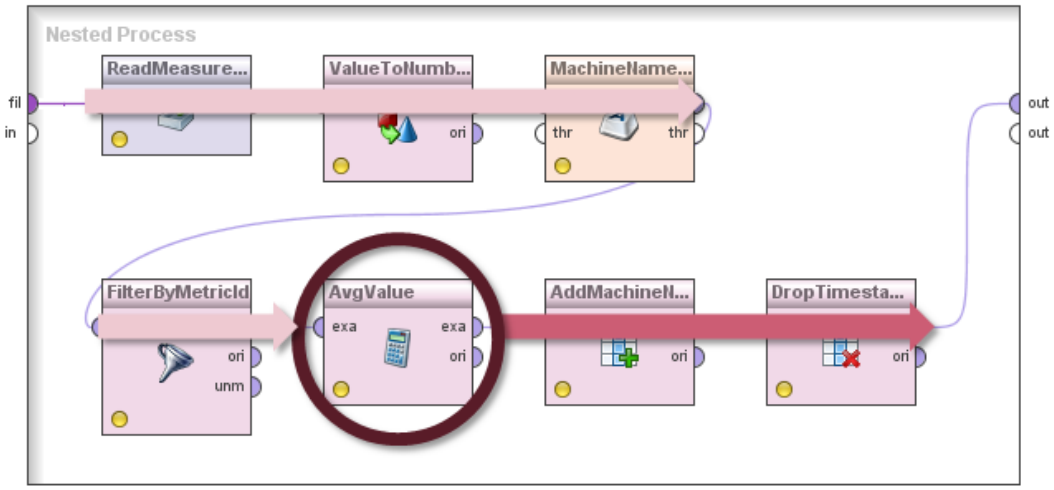
2.7. ábra. Filter nem engedi tovább a szűrőfeltételben szereplő attribútumban adathibás megfigyeléseket



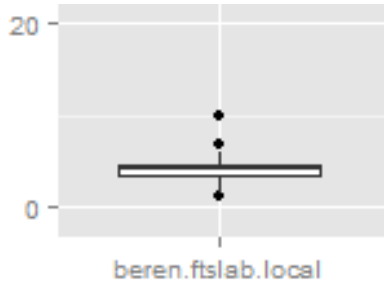
2.8. ábra. Hiányzó értékek helyettesítése Replace operátorral

Timestamp	Value	MetricId	Instance
1430418780	0	net.multicasttx.summation	vusb0
	10	cpu.usage.average	2
1430418780	0	disk.queuelatency.average	naa.5000cca00a2a07b8
1430418780	73	net.transmitted.average	vmnic1

2.9. ábra. Hiányzó időbélyeg



2.10. ábra. Aggregate operátor Timestamp alapján végzett groupBy művelete hibás kimenetet ad



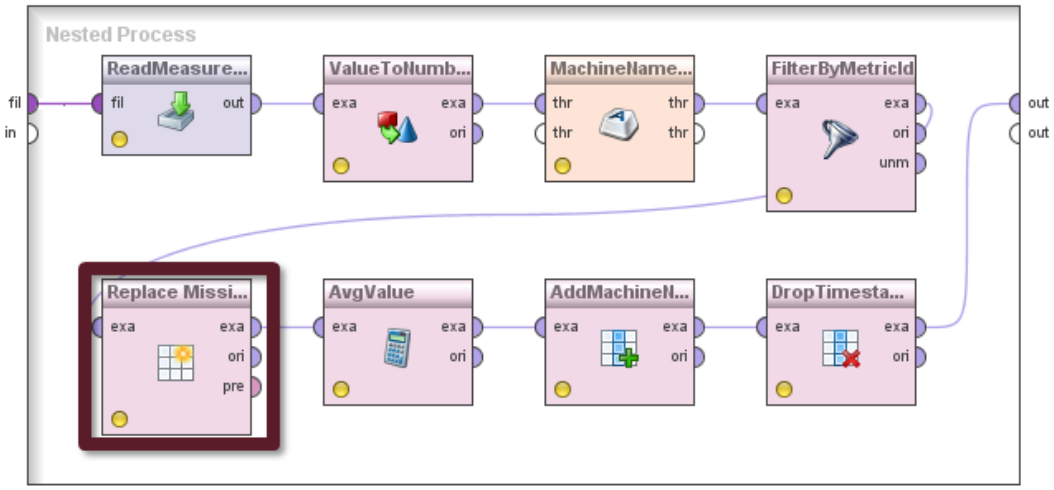
2.11. ábra. Hibás maximumérték

Ez észrevehető, ha tudjuk, hogy hányszor mintavételeztünk, hiszen így eggyel több átlagunk lesz. A hiányzó érték (*Missing Value*) kiküszöbölhető a *Replace Missing Values* operátor (2.12. ábra) segítségével. A hiányzó értékeket helyettesíthetjük az adott megfigyelést megelőző megfigyelés időbélyegével, hiszen Timestamp szerint rendezett adatokat kap a folyamat.

2.6.3. Illegális érték, értéktartományra vonatkozó kényszer megsértése

Lehetséges olyan eset is, hogy néhány mért érték a megengedett értéktartományon kívül esik. Például a *cpu.usage.average* metrika értéke százalékban van kifejezve, így lehetséges értékei 0-100 közé esik. Ha e tartományon kívül eső érték szerepel a mérési eredményeket leíró adatszerkezetben (2.13. ábra), akkor az értékekre vonatkozó kényszert sértő adathibáról (*Min-Max*) beszélünk. Az ilyen adathiba akkor okozza a boxplot diagram minimum/maximum értékének érvénytelen tartományba esését, ha kellően nagy az eltérés a megengedettől vagy kellően sokszor előfordul ahhoz, hogy az átlagot (2.10. ábra) módosítsa annyira, hogy az érvénytelen értéket vegyen fel (2.14. ábra). Ennek az adathibának a hatása elkerülhető, ha a folyamatban lévő *Filter Example* típusú operátort (2.15. ábra) további (az értéktartományra vonatkozó) szűrőfeltételekkel egészítjük ki.

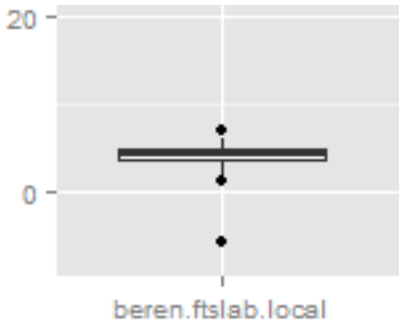
Látható, hogy egy rövidebb folyamatban is milyen sok hibalehetőség rejlik. Ezért fontos ezekkel a hibalehetőséggel tervezni, hiszen így megspórolhatjuk a hibás adatok elemzését.



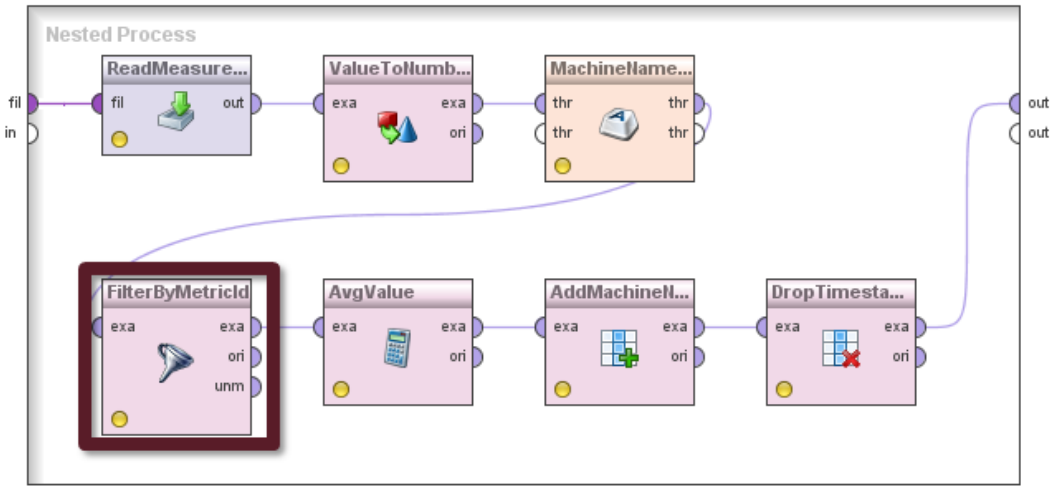
2.12. ábra. Hiányzó értékek megszüntetése Replace Missing Values operátorral

Timestamp	Value	MetricId	Instance
1430418780	0	net.multicasttx.summation	vusb0
1430418780	-40	cpu.usage.average	2
1430418780	0	disk.queuelatency.average	naa.5000cca00a2a07b8
1430418780	73	net.transmitted.average	vmnic1
1430418780	20	disk.write.average	
1430418780	6	disk.totallatency.average	naa.5000cca00a2a07b8

2.13. ábra. Értéktartományon kívüli Value érték



2.14. ábra. Hibás minimumérték



2.15. ábra. Értéktartományon kívül eső értékek kiszűrése új szűrése féltétel beállításával

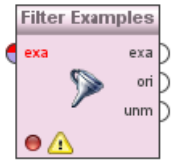
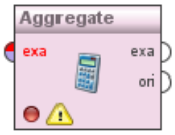
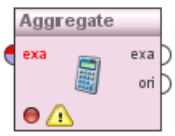
2.7. Folyamatdiagnosztika és -tervezés, hibaterjedés kvalitatív jellemzésével

Munkánk során ismertetett mintapélda estén, a folyamat szintjén felállíthatóak olyan relációk, melyek bemeneti hibatípusokat kimeneti hibatípusokhoz rendelnek (és viszont). A 2.16. táblázat az előző alfejezet hibaterjedésének folyamatszintű tulajdonságait foglalja össze.

Bemeneti hibatípus (CSV)	**Hibatípus az eredményben (boxplot)**
Elgépelés egy géphez tartozó mérések metricId attribútumában (*Typos*)	Hiányzó box
Hiányzó CSV a bemenetek között	Hiányzó box
Felesleges CSV a bemenetben	Felesleges box
Értékhiba (*Cell value subtle error*)	Hiba a Q1/Q2/Q3 kvartilisben
Hiányzó adatok (*Missing values*)	Hibás a minimum/maximum értéket jelölő pont
Értéktartományra vonatkozó kényszer megsértése (*Min-Max*)	Hibás a minimum/maximum értéket jelölő pont

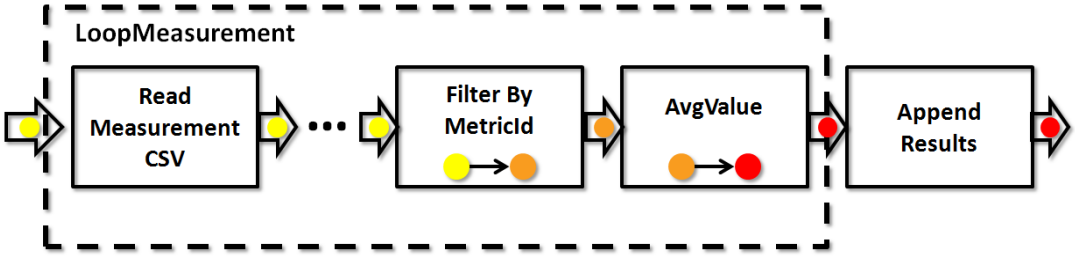
2.16. ábra. A folyamat hibatranszformációi a bemeneti állományoktól a *boxplot*ig

Azt is érdemes megfigyelnünk, hogy az operátorról operátorra „átadott" hibatípus valójában transzformálódnak a folyamat végrehajtása *közben*. Azaz nemcsak a végső lépés érzékenységét szükséges vizsgálni a bemenő hibatípusokra, hiszen azok a folyamat futtatása közben, bizonyos lépéseknél alapvetően megváltoznak. A mintafolyamatban bemutatott érdemi, egy operátor működéséből következő hibatípustranszformációkat a 2.17. táblázat foglalja össze.

Hibatípus a bemeneten	**Operátor**	**Hibatípus a kimeneten**
Elgépelés következtében keletkezett hibák (*Typos*)		Hiányzó sor (*Missing* observation)
Hiányzó adatok (*Missing Value*)		Felesleges sor (*Superfluous* observation)
Értéktartományra vonatkozó kényszer megsértésére (*Min-Max*)		Hibás cellaérték (*Cell value subtle error*)

2.17. ábra. Operátorok hibatranszformációi a motivációs példában

A 2.18. ábrán a motivációs példában bemutatott példafolyamaton mutatjuk be egy hiba útját, miközben több operátor is hatással van annak típusára. A folyamat bemenetként egy olyan megfigyelést kap, amelyben elírták a metricId értékét, így az erre az attribútumra szűrő filter lépés hibázni fog, lehet kiszűri a mérést, pedig nem kellett volna, ami később az átlagolás lépésnél hibás átlagot fog eredményezni, hiszen hiányzik az egyik adat az átlagból.



2.18. ábra. Hibatranszformáció a példafolyamatban

Amennyiben a feldolgozás/adattisztítás lépései során nem sikerül kiküszöbölni az összes adathibát, akkor ezek torzíthatják az adatelemzési lépések (pl. interaktív vizuális analízis, statisztikai módszerek) kimenetét. A hibás kimenet okának felderítése többnyire szakterület specialistákat és adatelemzőket igényel, s rengeteg időt és pénzt emészt fel. A kvalitatív hibamodellezés és hibaterjedés-analízis segítségével olyan módszert javaslunk, ami lehetőséget biztosít ezen költségek csökkentésére, azaz segít abban, hogy a lehető legkevesebb bemeneti adathiba legyen hatással az adatelemzés kimenetére.

Ugyanakkor ezek a hibák sokszor kivédhetőek további adattisztító és konzisztencia-ellenőrző lépések beiktatásával, amelyek megakadályozhatják a hibás értékek továbbterjedését az adathibákra érzékeny lépésekig.

3. fejezet

Adatminőség leírása és adathibák elemzési folyamatokban

Jelen fejezet áttekintést nyújt a néhány fontos, általános célú, folyamat alapú adatelemzési eszközről és az adatminőség jellemzőinek szabványos megközelítéséről. Az adatminőséget alapvetően befolyásoló adathibák szabványos, továbbá széles körben ismert és általános kategorizálást nem találtunk, így ez a fejezet önálló eredményként felállít egy taxonómiát, mely adatelemzési munkafolyamatokban hibák terjedésének leírására alkalmas. Az adathiba taxonómiát jelen munkában nem kötjük össze az adatminőségi szabványokkal, ennek legfőbb oka az, hogy az adatminőség fogalmainak számszerűsítéséhez (metrikák felállításához) szükség van jellemzően szakterületi információkat is felhasználó vizsgálatok elvégzésére, így ezzel a problémával csak a jövőben tervezünk foglalkozni. Megvizsgáltuk az adatelemző folyamatok általános felépítését, a tipikus operátorokat, majd ez alapján megalkottunk egy ontológiát, mely általános adatelemző folyamatokat ír le.

3.1. Adatelemző és adatfeldolgozó folyamatok

Napjainkban az adatelemzésnek egyre nagyobb jelentősége van. Az informatikai rendszerek rohamos terjedésével nap mint nap rengeteg adat keletkezik. Ezek feldolgozásával és elemzésével leírhatjuk rendszerek viselkedését, javaslatot tehetünk vállalatok profitnövelési lehetőségeire és detektálhatjuk szoftveres rendszerek kiskapuinak kártékony kihasználását. Egy adatelemző folyamat végrehajtása után riportokkal leírhatjuk, hogy mi történt rendszerünkben, feltérképezhetjük az események közötti összefüggéseket valamint előrejelzést adhatunk arra, hogy hogyan fog alakulni a rendszer további működése. Mobilszolgáltatók növelhetik ügyfélkörüket, ha meg tudják határozni kik a kiterjedt kapcsolati hálóval rendelkező ügyfelek. Nyilván ők fontosak a véleményterjesztésben, így az ő elégedettségi szintjük növelése elengedhetetlen. Azaz a hívási adatok elemzésével meghatározhatják, kik azok az emberek, akik sokat sms-eznek, telefonálnak más mobilszolgáltatóhoz tartozó hálózatba, és különböző kedvezményezek ajánlásával növelhetik bizalmukat. Biztosítók adatelemzés segítségével csalásokat detektálhatnak, például felderíthetik a feltűnő összeghatár alatti, rendszeres kárköveteléseket.

Az adatelemzésnek sokfajta megközelítése létezik. Léteznek alapvetően vizuális feltáró adatanalízist (exploratory data analysis) támogató eszközök, programoz-

ható, scriptelhető statisztikai környezetek, de mi azt az esetet vizsgáljuk, amikor az adatelemző folyamatokat specializált munkafolyamat alapú eszközökkel hozzuk létre. Ilyenkor ezek a folyamatok elemi lépésekből (import/export, adattranszformáció, típuskonverzió, értékmódosítás, adattisztítás, statisztikai jellegű vizsgálatok) épülnek fel. A lépések végrehajtási sorrendjét a lépéseket összekötő kapcsolatok határozzák meg. Az egyes lépések a módosított adatszerkezetet adják tovább az őket követő lépésnek. Így a folyamat kimenete a végrehajtott lépések által átalakított bemeneti lesz.

3.1.1. Adatelemző eszközök

Az adatelemző/adattisztító/adatfeldolgozó eszközök tanulmányozásánál olyan programokat néztünk meg, amelyek támogatják a megismételhető, szakterületi tudást felhasználó elemzést. A következő szempontokat vettük figyelembe a vizsgálatuk során:

- *Támogatja az alapvető adatelemzési lépéseket?* Az adatszerkezet szűrését, új attribútummal való kiegészítését, egy-egy plot megjelenítését, külső szkript meghívását és egyéb elemi műveleteket nevezzük alapvető adatelemzési lépésnek. Ez azért fontos, hogy az ilyen elemi operátorokat ne kelljen definiálni és implementálni, amennyiben létezik olyan eszköz, amely ezt biztosítja.

- *Bővíthető saját operátorokkal, új lépéstípusokkal?* Fontos szempont, hogy az adott felhasználási területen, az elemzés során rendszeresen ismétlődő feladatok elvégzésére saját lépéstípusokat/operátorokat lehessen írni. Például, ha miden adatelemzési folyamat esetén meg szeretnénk vizsgálni, hogy a mérési adatok azonos időközönként lettek-e mintavételezve (azaz „jitter mentes”-e a mérés), akkor célszerű lehet erre a feladatra egy saját operátort definiálni. Noha ez a feladat megoldható lenne az eszközbe beépített alapvető adatelemzési lépésekkel is, mégis áttekinthetőbb és adott esetben teljesítmény szempontjából kedvezőbb folyamatot kapunk, ha külön operátort definiálunk a művelet elvégzésére.

- *Létezik hozzá API?* A később részletezett metamodellnek megfelelő példány generálásához lényeges, hogy az adatelemzési eszköz által létrehozott folyamatokhoz kívülről is hozzá lehessen férni. Így folyamat szintű fogalmakkal lehet foglalkozni, és nem XML feldolgozással kell foglalkozni.

- *Könnyen használható?* Elengedhetetlen, hogy azok a felhasználók is könnyedén tudjanak adatelemzési folyamatokat létrehozni és futtatni, akik nem rendelkeznek informatikai/programozói végzettséggel.

A RapidMiner, a Knime, a Wings, az OpenRefine és az IBM Watson Analytics eszközöket vizsgáltuk meg a fent említett szempontok alapján.

- A *RapidMiner* [32] egységes környezetet biztosít az adatbányászat, a szövegbányászat, a prediktív analízis és az üzleti analízis számára. Széles körben elterjedt, üzleti és ipari környezetben is használják kutatásra, oktatásra, alkalmazások bővítésére, prototípusok bemutatására. Szintén lehetővé teszi a

folyamat alapú adattisztítást és elemzést. Rengeteg adatelemző műveletet ismer, támogatja a bemeneti adatok validációját és optimalizálását. Új operátorokkal bővíthető, a folyamatok XML formátumban exportálhatóak. Grafikus felület segítségével biztosítja az adatelemző folyamatok egyszerű létrehozását. Képes ontológia alapú adatbázisokhoz kapcsolódni, ott lekérdezéseket futtatni.

- A *Knime* [18] nyílt forráskódú, folyamat alapú, grafikus eszköz. Komponenseiben ötvözi a gépi tanulást az adatbányászattal. Lehetőséget biztosít ETL folyamatok összeállítására, adatelemzésre és adatvizualizációra is. Ismeri az alapvető adatelemzési lépéseket, bővíthető, API-n keresztül elérhető, könnyen használható, noha felhasználói felülete elavultabb a RapidMiner által nyújtottnál.

- A *Wings* [39] egy kutatási céllal készült, folyamat alapú adatelemzést támogató eszköz. Bővíthető új lépésekkel, de nem ismeri az alapvető adattisztítási/adatelemzési lépéseket, így használatával minden szükséges operáció implementálása nagyon időigényes feladat. Ebből következik, hogy elég bonyolult, nem könnyen alkalmazható programról van szó.

- Az *OpenRefine* [24] adattisztításra alkalmas, különböző szűrők, transzformációk, rendezések alkalmazhatóak az adatokon. Clusterek szerinti szerkesztés is lehetséges. Képes scatter plot (pont-pont diagram) megjelenítésére, de bonyolultabb, adatelemzést támogató lépések elvégzésére nem alkalmas. Lehetővé teszi webservice szolgáltatások meghívását. A mérések eredményein végrehajtott műveletek scenario formájában rögzíthetőek, más bemeneten is lejátszhatóak és JSON formátumban exportálhatóak. Elérhető API csak új formátumokhoz importerek/exporterek írására létezik, tehát új művelet írása nem lehetséges.

- Az *IBM Watson Analytics* [15] softver as a service lehetőséggel biztosítja az alapvető adattisztítási lépéseket, de a végrehajtott műveletek nem naplózhatóak, tehát ugyanaz a folyamat nem hajtható végre más adatokon csak úgy, hogy a korábbi operációkat újra végrehajtjuk a felhasználói felületen. Így maga a végrehajtott folyamat nem exportálható és külső API sem létezik az eszközhöz. Ugyanakkor laikusok számára is érthetővé teszi az automatikus adatelemzés során észlelt összefüggéseket, valamint a bemeneti adat minőségét is meghatározza több jellemző figyelembevételével (pl. hiányzó és konstans értékek, outlier, skewness). Funkcionalitása részben beépíthető más eszközökbe.

A 3.1. táblázatban láthatjuk a különböző adatelemző eszközök összehasonlítását. A RapidMiner és a Knime felelt meg a fent megfogalmazott szempontoknak, így ezek képezték az adatelemző folyamatok modellezésének alapját. A konkrét folyamatokat RapidMinerben hoztuk létre.

3.1.2. Adattisztítás

Az adattisztítás az a folyamat, ami feloldja a forrás rendszerektől érkező adatokban levő inkonzisztenciát és esetleges anomáliákat. Ezek eltávolításával növelhető az adatminőség. Az adattisztító folyamat három fázisra bontható. Az első az adatok

	Alap adatelemzés?	*Bővíthető?*	*API?*	*Felhasználóbarát?*
OpenRefine	igen	nem	igen	igen
IBM Watson Analytics	igen	nem	nem	igen
Wings	nem	igen	igen	nem
RapidMiner	igen	igen	igen	igen
Knime	igen	igen	igen	igen

3.1. ábra. Adatelemző eszközök összehasonlítása

típusának vizsgálata (*Screening Phase*), a második a hihetőségvizsgálat (*Diagnostic Phase*), a harmadik pedig a hibás adatok kijavítása (*Treatment Phase*) [36].

Adattisztításra különösen nagy szükség van akkor, amikor különböző adatforrásokból származó adatokat integrálunk az együttes feldolgozás érdekében. Például adattárházak esetében megvalósul a több forrásból származó adat együttes kezelése, így fontosabb lesz az adattisztítás, hiszen különböző reprezentációjú adatok tárolása redundáns [31]. Tehát nagyon sok erőforrást fordítunk adattisztításra azért, hogy az adatokat előkészítsük a feldolgozásra. Az adatanalízis folyamatának 80%-át tipikusan az adattisztítás és az adatok előkészítése teszi ki [38]. Adattisztítási feladatok közé tartozik többek között az adatban szereplő hibák felmérése, az adatállomány szerkezeti épségének ellenőrzése, a hiányzó értékek kezelése, az eloszlások szélein található extrém értékek (*outlier*) vizsgálata és esetleges szűrése, a duplikátumok eltávolítása, egy adott értékkészleten kívül eső adat kezelése és a beolvasási/értelemezési feladatok elvégzése.

Adatelemző folyamatok futtatásakor nem helyes eredményt/outputot kaphatunk, amennyiben a bemeneti adatok nem tiszták. Két módszer közül választhatunk, hogy elkerüljük a mérési hibák, pontatlanságok és hiányosságok kimenetre gyakorolt hatását. Az egyik lehetőség, hogy tiszta adatokat tárolunk. Azaz adattárolás előtt ellenőrizzük az adatok minőségét, amennyiben ez nem felel meg a követelményeknek, akkor javítjuk a mérési eredményeket és csak ezek után tároljuk el őket. Így az adatelemző folyamat tiszta adatokat kap, tehát az adattisztítással ott már nem kell foglalkozni. A másik lehetőség, hogy a piszkosan tárolt adatokat közvetlen az adatelemzés előtt és közben tisztítjuk meg, a folyamat részeként. Ebben az esetben erőforrást spórolunk, hiszen csak azokat az adatokat tisztítjuk meg, amelyek valóban feldolgozásra kerülnek.

A dolgozatban azzal az egyébként tipikus esettel foglalkozunk, amikor az adattisztítás az adatfeldolgozó/adatelemző folyamat részeként jelenik meg.

3.1.3. Robosztusság

Egy folyamat alapú rendszer hibatűrővé (robusztussá) tételéhez szükség lehet az egyes lépések hibatűrésének biztosítása vagy a vezérlés hibatűrési tulajdonságainak fejlesztése. Ez az adatelemző folyamatok esetében is így van.

Adatelemző folyamatokban a lépések hibatűrésének biztosítása robusztus operátorok használatával érhető el. (Egy operátort/elemi lépést akkor nevezünk robusztusnak, ha érzéketlen a bemenetén megjelenő adat hibáira.) Tehát a folyamatot felépítő nem robusztus operátorokat lecseréljük azonos funkciót ellátó, statisztikai szempontból hibatűrőbb (lsd. lejjebb robusztus statisztika) operátorokra, azaz oly módon implementáljuk a lépést, hogy fel legyen készítve az adathibákra. A másik

megoldás, hogy a nem robusztus operátorok elé olyan adattisztító/adathiba-javító lépéseket szúrunk be (wrapper objektumhoz hasonlóan), amik kiküszöbölik azokat a hibákat, amelyekre az adott operátor érzékeny.

A vezérlés hibatűrési tulajdonságainak fejlesztéséről többek között a [8] cikk is ír. Üzleti folyamatokban a különböző hibák hatásának kivédését hibakezelő ágak beiktatásával lehet biztosítani, ahol hibajavító jellegű művelet elvégzésével elkerülhető a hiba. Általános célú üzleti folyamatokban a BPMN [34] és a BPEL [33] szabványban leírt módon van lehetőség elemi hibáktól mentes folyamatot létrehozni, kivételkezelést támogatni. Ezek a szabványok azonban nem definiálják, hogy hogyan használjuk ezeket a hibakezelési mechanizmusokat, azaz nem határozzák meg, hogy hogyan és hol tehető robusztussá egy folyamat. Módszerünkkel erre nyújtunk támogatást, adattisztító/adathiba-javító lépések wrapper objektumként való beszúrásával.

Ahogy korábban említettük, folyamatleíró nyelvekben az egyes lépések is robusztussá tehetőek. Ebben a kontextusban a statisztikai értelemben vett robusztusság az érdekes, ami meghatározza, hogy egy statisztikai metódus (pl. átlag, medián) mennyire érzéketlen a feltételezésektől való kis eltérésre. Azaz kifejezi, hogy például a hiányzó adatok, a kiugró értékek, a duplikátumok mennyire torzítják az adott statisztikai metódus eredményét. A robusztus statisztika ellenáll azoknak a hibáknak, amiket feltételezéstől való eltérés okoz [14]. Egy statisztikai metódus robusztusságát több jellemzővel értékelhetjük. Az egyik ilyen a letörési pont, ami meghatározza, hogy az adatok hány százalékát (hányadát) lehet elrontani úgy, hogy az adott művelet eredményét ne befolyásolja a hiba. Ez az átlag esetében 0, hiszen egy bemeneti érték megváltoztatásával jelentős mértékben változtatható a kimenet. Az adatok felének elrontásával már biztosan módosul a medián értéke, így ennek letörési pontja 0,5. Ebből következik, hogy a medián robuszt.

3.2. Adatminőség és adathibák

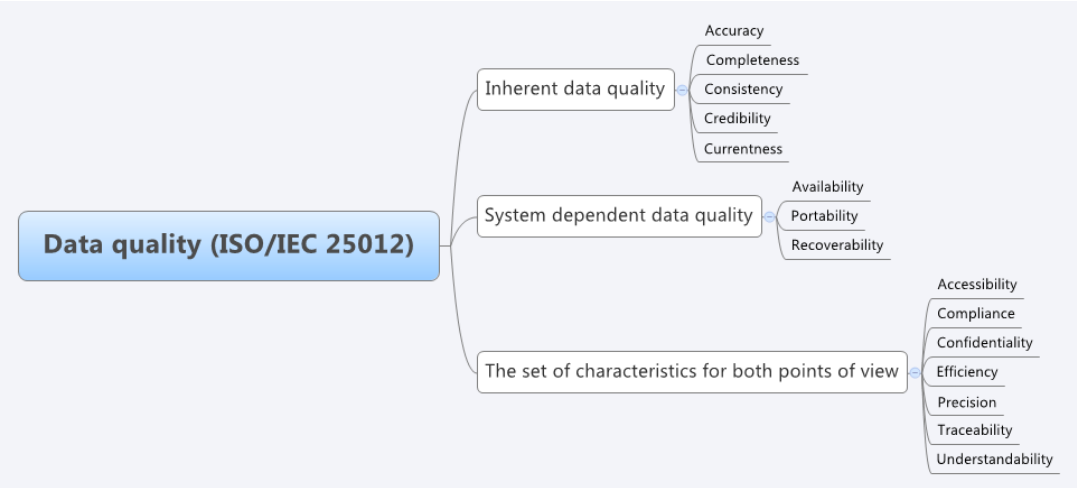
Egy adatszerkezet minőségét egyértelműen meghatározzák az adatszerkezetben előforduló adathibák. Így mi az adatminőség kontextusán értelmezhető adathibák terjedéséről beszélünk. A szakirodalomban létezik adatminőséget leíró szabvány [16], ezt a továbbiakban röviden összefoglaljuk. Adathibák általános modellezésére elfogadott a [7] cikkben leírt detektálás-orientált klasszikus hibamodell, de ebből néhány nem alkalmazható, néhány pedig nem elég specifikus ahhoz, hogy a motivációban látott adathibákat és adathiba-terjedést modellezzük. Ezért javasoltunk egy önálló adathiba-taxonómiát, amelyet ebben a fejezet fogunk részletesen bemutatni.

3.2.1. Adatminőség

A szakirodalom az adatminőség fogalmát úgy közelíti meg, hogy felsorolja azokat a kritériumokat, aminek egy jó adatminőségű adatnak meg kell felelnie. Amennyiben az adat megfelel minden ilyen elvárásnak, akkor kiváló adatminőségről beszélünk. Ha csak néhány kikötést teljesít, akkor az adatminőség rosszabb.

Az *ISO/IEC 25012* (Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Data quality model) szabvány [16] egy általános adatminőség modellt definiál a strukturált formában tárolt adatokra. A minőség leírására alkalmas 15 jellemzőt kettő szempont alapján kategorizálja

(3.2. ábra). Az első leírja, hogy milyen adatminőség karakterisztikák fakadnak magából az adatból (*Inherent data quality*), a második meghatározza, hogy milyen adatminőséget leíró jellemzők érhetőek el és őrizhetőek meg számítógépes rendszerekben (*System dependent data quality).* Az egyes adatminőség karakterisztikáknak más jelentőségük és fontosságuk van különböző területek esetén [21] [22] [30]. Azonban a gyakorlatban nem sok alkalmazását láttuk ezeknek a szabványoknak.

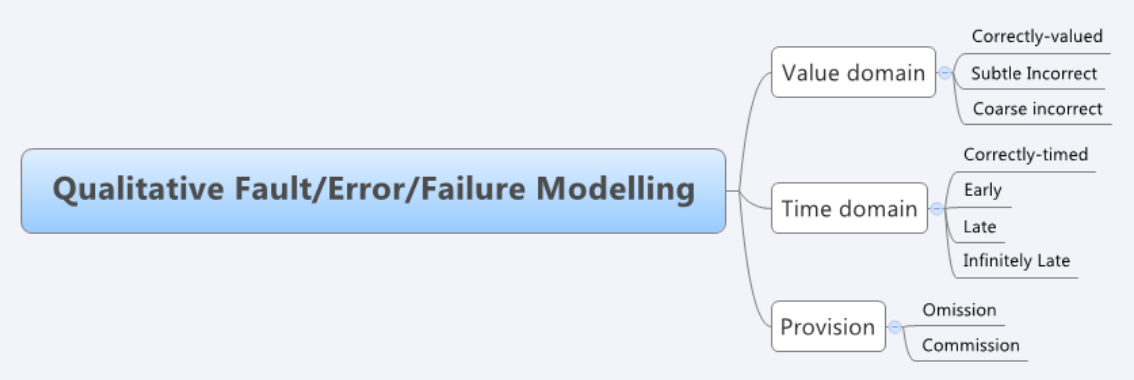


3.2. ábra. ISO/IEC 25012 szabvány adatminőség karakterisztikái

Az adatminőség praktikus számszerű jellemzésére egy jó példa az *IBM Watson Analytics* [15] által adott megoldás. Egy adott adatszerkezet adatminőséget egy százalékban kifejezett értékkel jellemzi. Ezt a mérőszámot meghatározzák az adatszerkezetben előforduló konstans értékű attribútumok, hiányzó adatok, outlier értékek és a ferdeség. A ferdeség egy eloszlás asszimmetriáját méri, ahol a szimmetria az értékek medián körülötti eloszlását jelenti. A mérőszámot befolyásolják még *influential category* típusok, azazhogy melyek azok a kategóriák, amelyek nagy frekvenciával fordulnak elő. A frekvenciát χ^2 próba alapján határozza meg. Az adatminőség kalkulálásához használt jellemzőket a módszer továbbfejlesztésénél figyelembe lehet venni, ugyanis ezek változásai is végigkövethetőek az adatelemzési folyamaton.

3.2.2. Adathibák

Az adat és vezérlési hibák kategorizálására a hibatűrő számítástechnika rendelkezik kvázi szabványos megközelítéssel, ezek közül az egyik leggyakrabban használt a 3.3. ábrán látható detektálás-orientált klasszikus hibamodell [7]. Az 5.2.1. résznél idézünk további olyan megközelítéseket, amelyek részlegesen más hibamodellt alkalmaznak. Bondavalli és Simoncini ezt a hibamodellt komponenshálózatként modellezett számítógépes rendszerek komponenshibáinak leírására alkotta meg, így a mi kontextusunkba nem alkalmazható közvetlenül. Ezek a kategóriák alapvetően szinkron és aszinkron kommunikációk hibakategóriáit adják meg. Egy adatelemzési folyamatnál jellemző nem nagyon foglalkozunk azzal, hogy két operátor közt átadott adat, milyen időzítéssel kerül átadásra, az átadott adaton belül található időbélyeggel ellátott megfigyeléseknél pedig az időzítés hibája szemantikailag inkább adathiba, mint időzítési hiba, tehát valójában nekünk az időzítési kategóriára



3.3. ábra. Qualitative Fault/Error/Failure Modelling [7] alapján

nincs szükségünk. A jelenléti[1] problémák jelentése is egy kicsit más adatelemzésnél, mert alapvetően nem az érdekes, hogy két kommunikáló komponens interakciója megtörténik-e vagy feleslegesen történik, hanem a két operátor közt átadott adatban vannak-e hiányzó, illetve felesleges adatelemek. Abból következően, hogy egy adatelemzési folyamatnak a végrehajtási logikája más, mint egy általános komponens alapú architektúráé.

Így a klasszikus hibamodell alapján, a szakirodalomban létező adathibakategorizálások [38] [31] ismeretében, valamint a korábbi adatelemzések során gyűjtött tapasztalataink segítségével megalkottunk egy *adathibákat leíró taxonómiát.*

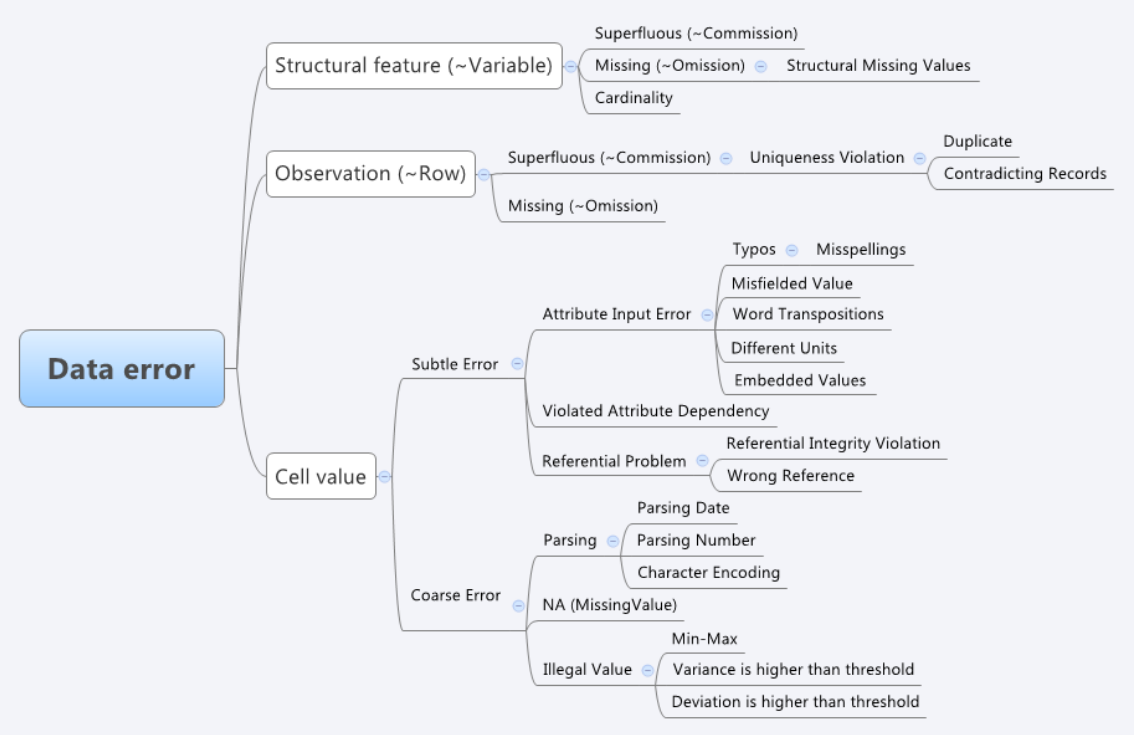
Modellünkben (3.4. ábra) ötvöztük a hibaterjedés analízis során megjelenő kvalitatív hibakategóriákat (3.3. ábra) az általános adathibákkal. Az adathibák kategorizálását szakterülettől függetlenül végeztük, így az ezeket leíró taxonómia tovább finomítható szakterület-specifikus tudás figyelembevételével.

Először megvizsgáltuk, hogy az adathiba az adatszerkezet mely részét érinti. Így megkülönböztettünk attribútumokat (*Structural feature - Variable*), sorokat (*Observation - Row*) és cellákat (*Cell value*) érintő hibákat.

Attribútumok hibalehetőségei

Attribútumok esetében előfordulhat, hogy felesleges attribútum (*Superfluous - Commission*) jelenik meg az adatszerkezetben, illetve hogy hiányzik egy elvárt attribútum (*Missing - Omission*). Ezek a hibák megjelenhetnek például úgy, hogy a mérés során mintavételező script megváltozik, és más metrikákat rögzít, mint korábban. Így az adatok együttes kezelésekor előfordulhatnak strukturális hibák. Ilyen jellegű hibák észrevehetőek az attribútumszám ellenőrzésével, illetve egy referenciasémának való megfeleltetéssel. Kardinalitásból fakadó problémáról (*Cardinality*) beszélünk akkor, amikor egy attribútum több értéket vesz fel, mint amennyi megengedett. Például tudjuk, hogy egy cella érték férfi/nő lehet, de azt tartalmazza, hogy lány. Detektálható az adathiba, ha tudjuk, hogy milyen értéket vehet fel az adott attribútum.

[1] A Provision jellegű hibákat jelenléti hibáknak fordítottuk, mivel az angol szó magyar megfelelői között nem találtunk olyat, ami a hiba jellegét valóban kifejezné.



3.4. ábra. Adathiba taxonómia

Sorok hibalehetőségei

Sorok esetében is beszélhetünk hiányzó (*Missing - Omission*) és felesleges (*Superfluous - Commission*)sorról. Felesleg esetén könnyen sérülhet az egyediségre vonatkozó kényszer (*Uniqueness Violation*). Ez adódhat sorok duplikálásából (*Duplicate*) vagy abból, hogy ugyanaz a valós entitás különböző értékekkel lett leírva (*Contradicting Records*). A felesleges sorokat detektálhatjuk a duplikátumok ellenőrzésével, a hiányzó sorok észlelésére fent már említettünk egy megoldást.

Cellák hibalehetőségei

A cellákat érintő adathibákat két nagy csoportba sorolhatjuk. Léteznek olyan hibás cellák, amelyek értékéből nem derül ki, hogy adathiba van az adott helyen (*Subtle Error*), és léteznek olyan adathibák, melyek a cella értékéből egyértelműen meghatározhatók (*Coarse Error*).

Subtle Error típusba soroltuk a következő hibákat:

- A cellaérték hibás beviteléből származó hibákat (*Attribute Input Error*). Ennek több oka lehet:
  - Elgépelés következtében keletkezett hibák (*Typos*), helyesírási hibák (*Misspellings*). Ezek detektálására megoldás lehet a szöveget tartalmazó cellák szavainak helyesírási szótárban való kikeresése.
  - Helytelenül kitöltött értékek (*Misfielded Value*), azaz nem a megfelelő dolog került a mezőbe. Például országnév helyett város. Ennek azonosításához szakterületi tudás is szükséges. Ha tudjuk, hogy egy cellában

csak országnév lehet, és nem találjuk az összes országot tartalmazó listában, akkor az adott cellában adathiba van.

  - Hibás szórend (*Word Transpositions*). Például a vezetéknév és a keresztnév helytelen sorrendben történő bevitele. Itt is hasonló megoldással lehetünk rá a problémára, mint az előbbi esetben. Ha tudjuk, hogy az első szónak a keresztnévnek kell lennie, akkor ellenőrizhetjük, hogy létezik-e ilyen keresztnév.
  - Különböző mértékegységekben rögzített adatok (*Different Units*). A cellaérték helyes nagyságrendbeesésének ellenőrzés az ilyen típusú hibák felderítésére alkalmas lehet.
  - Egy cellába több érték bevitele (*Embedded Values*). Például name="J. Smith 12.02.70 New York". Az ilyen esetek a cella tartalmára vonatkozó hierarchikus szabály kiértékelésével felderíthetőek. Jelen esetben mondhatjuk, hogy a name attribútum csak betűket tartalmazhat, számokat nem.

- Attribútumok közötti függőségi kényszer megsértéséből származó adathiba (*Violated Attribute Dependency*). Például nem teljesül az egyes cellák között a következő kényszer: age =current date – birth date. Az ilyen típusú adathibákra könnyen ráakadhatunk, amennyiben az összes függőségi kényszert ellenőrizzük a cellákon.

- Adathibát okozhat a hibás hivatkozás is (*Referential Problem*). Ezt több dolog is okozhatja:

  - Nem létező objektumra történő hivatkozás (*Referential Integrity Violation*). Például van egy adatszerkezetünk, amely a hallgatók Neptunkódját és annak a szaknak az azonosítóját tartalmazza, amelyre járnak(TABLE1(neptun,szakID)). Illetve van egy táblázatunk, ami az egyes szakok jellemzőit írja le (TABLE2(szakID,szaknev,kepzeshossza,..)). Ebben az esetben (neptun="ABC", szakID=17) a 17-es azonosítójú szakra hivatkozik. Amennyiben nem létezik ilyen azonosítójú szak, akkor nem létező objektumra történő hivatkozásról beszélünk. Az ilyen esetek megtalálására egy lehetőség az, ha ellenőrizzük, hogy az idegenkulcs (TABLE1.szakID), szerepel-e a hivatkozott tábla kulcsai között (TABLE2.szakID).
  - Hibás objektumra történő hivatkozás (*Wrong Reference*). Az előző példánál maradva, ha létezik a 17-es azonosítójú szak, de helytelenül van definiálva (pl. hibás szaknév), akkor hibás objektumra történő hivatkozás történt. Ez egy formailag nehezen észrevehető adathiba.

Coarse Error típusba a következők kerültek:

- Helytelen értelmezéséből/beolvasásból származó hibák (Parsing). Ide sorolhatóak a dátum és a számok beolvasásából, valamint a karakterkódolásból adódó hibák. Eszköztől függően okozhat futási idejű kivételt vagy nehezen észrevehető adathibát.

- Hiányzó adatok (*Missing Value*). Az ilyen esetek felismerése nem mindig könnyű, hiszen, attól, hogy egy cella értéke üres, nem feltétlenül hiányzik. Lehet, hogy valaminek az alapértelmezett értéke az, hogy nincsen kitöltve. Ugyanakkor az is előfordulhat, hogy valamilyen okból hiányzik egy adott cella értéke.

- Illegális érték szerepel a cellában (*Illegal Value*). Ennek több oka lehet:

  - Értéktartományra vonatkozó kényszer megsértésére (*Min-Max*). Százalékban mért érték 0 és 100 közötti legyen. Felismerhető a hiba a megengedett értéktartományba esés ellenőrzésével.
  - Variancia nagyobb az értéktartománynál (*Variance is higher than threshold*).Felismerhető a hiba a megengedett értéktartományba esés ellenőrzésével.
  - Szórás nagyobb az értéktartománynál (*Deviation is higher than threshold*). Felismerhető a hiba a megengedett értéktartományba esés ellenőrzésével.

Felmerülhet a kérdés, hogy mi a helyes besorolás akkor, ha egy hiba több kategóriába is tartozhat. Például egy teljes sor hiányára úgy is tekinthetünk, hogy az adott megfigyelést reprezentáló összes cellában hiányzik az érték. Ilyen esetekben mindig a legtágabb kategóriát vesszük figyelembe, azaz jelen példában ez egy sorhiány (*Observation/Missing*) lesz, és nem annyi hiányzó érték (*Cell value/Coarse Error/Missing Value*), ahány cella üres. Ez a fajta besorolás azért is kedvező, mert egy sorról nem feltétlenül vesszük észre, hogy hiányzik. (Ha személyigazolvány számok - nevek vannak egy táblázatban, akkor egy sor hiányát nehéz detektálni. Amennyiben adott időközönként rögzített mérési eredményekről beszélünk, akkor egy időbélyeg kimaradásából észrevehető a megfigyelés hiánya.) A hiányzó cellaérték (*Missing Value*) *Coarse Error* (lsd. később) kategóriába került, de a fent említett esetben ez nem lenne helyes besorolás, hiszen ha az összes cella hiányzik, az már nem feltétlen észrevehető.

Az általunk megalkotott adathibamodellt a későbbiekben fel fogjuk használni arra, hogy az adatelelemző folyamatok lehetséges lépéstípusainak adathibatranszformációs szabályait felírjuk. Azaz meghatározzuk, hogy az egyes lépéstípusok esetében a lépés bemenetén megjelenő adathibákból, a lépés végrehajtása után, milyen típusú adathibák jelenhetnek meg kimeneten.

3.3. Adattípusok

A szakirodalomban az adattípusoknak többféle felosztása létezik. Mi a [17] könyvben definiált adattípusokkal dolgoztunk, melyekre a 3.5. ábrán adtunk példákat. Két szakterületről gyűjtöttünk példákat az egyes adattípusukra. Az egyik a 2.1. fejezetben ismertetett *Cloud Performance Management* projekt mérési környezete, a másik pedig illusztrációs céllal a mindannyiunk számára valamennyire ismert főzés területe.

Az adattípusoknak fontos szerepük van a hibaterjedés szempontjából. Vannak olyan operátorok, amelyek egy-egy típusú változó hibájára jobban érzékenyek. Például az adatok szűrésekor (filter), egy *binary* típusú változó hibája nagy valószí-

Adattípus	Leírás	Cloud Performance Management	Főzés
Indexing	Általában nevek, tagek, ügyszámok, sorozatszámok, amelyek azonosítanak valamit. (Primary key, foreign key)	Mérhető metrikák azonosítói.	Élelmiszer vonalkódja.
Binary	Olyan kategória típusú attribútum, ahol az attribútum csak két értéket vehet fel.	A mérési környezetet leírásánál, egy gép host vagy guest.	Az adott étel sós vagy édes?
Boolean	Csak két értéke lehet, de az is csak a TRUE, vagy a FALSE (esetleg az UNKNOWN).	Minden szükséges metrikát mintavételeztünk a mérés során?	Van benne tojás?
Nominal	Ide tartoznak a *character* és a *string* adattípusok. A *binary* változó általános esete. Meghatározott számú értéket vehet fel az ilyen típusú változó. Az attribútum értékei között csak azonosság vizsgálható, sorba nem rendezhetőek.	Az adott metrikát hol mérjük? (Disk, memory, cpu, intarface)	Liszt típusa? (Rétes liszt/finom liszt/tönköly liszt...)
Ordinal	A sorrend típusú attribútumoknál a *character* és *string* értékeket sorba tudjuk rendezni, azaz az attribútum értéken teljes rendezést tudunk megadni. Ha tehát a!=a', akkor még azt is tudjuk, hogy a > a' és a < a' közül melyik igaz. (Pl. iskolai végzettség, termékek minősítés értékei, tanulmányi versenyen kialakult eredmény)	A lineáris regresszió számításánál kategorizálhatjuk a mérési eredményeket, hogy jók (illeszkednek az egyenesre), gyanúsak (az egyenestől csak bizonyos százalékban térnek el), vagy rosszak.	Kis-, közepes-, vagy nagylángon lángon kell főzni?
Integer	Általában nem negatív egész számok, amelyek gyakran számosságot fejeznek ki.	Az egyes mérések mintavételezésének pillanatai epochban mérve.	Hány tojás kell hozzá?
Continuos	Mért érték, amelyről feltételezzük a folytonosságot. Az adatbázis-kezelő rendszerekben *numeric* vagy *decimal* értékként jelennek meg.	A mérés során, az egyes időpillanatokban mintavételezett értékek.	Hány kg vaj kell hozzá?

3.5. ábra. Adattípusok példákkal

nűséggel a kimenetre is hibát gyakorol (nem jelenik meg az adott mintavételezés), míg egy *continous* változó hibája lehet, hogy nem okoz hibát. *Binary* típusnál, ha a host gépeken rögzített méréseket akarjuk megjeleníteni, és valahol a host helyett guest szerepel, akkor az a mérési eredmény nem jelenik meg a szűrés után. Azonban, ha a mintavételezett értékre rögzítünk filtert (50-nél nagyobb értékekre vagyunk kíváncsiak), akkor az, hogy egy helyen 60 helyett 65 szerepel, nem okoz hibát, a mintavételezés megjelenik a szűrés kimenetén. Ez a megközelítés az általunk megalkotott módszer továbbfejlesztését teszi lehetővé.

4. fejezet

Modellezési és elemzési módszer

Jelen fejezetben bemutatjuk az adattisztítási folyamatok felépítését és az ontológítával leírt modelleket. Módszert adunk, amellyel vizsgálható az adatelemző folyamatok bemeneti hibákra való érzékenysége.

4.1. Adattisztítási folyamatok felépítése

Az általunk megvalósított rendszer adatelemző/adattisztító folyamatokon végez hibaterjedés-analízist. Így az első feladat az adatelemző folyamatok felépítése volt. Ezeket a folyamatokat a RapidMiner eszközzel hoztuk létre, ahol az adatelemző folyamat különböző típusú lépések összességéből épül fel. Minden lépés egy-egy elemi feladat elvégzéséért felelős. Ezek lehetnek például az adatszerkezet beolvasásáért/exportálásáért felelős, adattranszformációs, típuskonverziós, értékmódosító, adattisztító és statisztikai jellegű lépések. A lépések működése paraméterek megadásával specializálható. Ezek a lépések kapcsolatokkal vannak összekötve, amik továbbítják az adatszerkezetet (mérési eredményeket) a rákövetkező lépésnek.

Ismertetünk néhány példát az elkészült folyamatainkból. A 2. fejezetben bemutatott adatelemzésen kívül megalkottunk egy teljességellenőrző folyamatot, ami azt vizsgálja, hogy minden mintavételezési pillanatban mértük-e az összes szükséges metrikát. A vizsgálat kimenetén azok az időbélyeg-erőforrás-metrikanév hármasok jelennek meg, amelyeket a mérési eredményeket leíró adatszerkezet nem tartalmaz, pedig rögzíteni kellett volna. Munkánk során ez a folyamat fontos volt számunkra, mert komplexitása miatt jó példa volt mérések futtatására. Emellett a folyamat kimenete segíti a mérésvezérlésben rejlő hibák detektálását (ezzel elkerülve a felesleges analízis elvégzését). A példát az F.1. függelékben részletesebben ismertetjük.

Összeállítottunk egy adattisztítást végző eljárást is, ami eltávolítja a nem informatív attribútumokat a bemeneten kapott adatszerkezetből. Azt vizsgálja, hogy egy oszlop cellái több értéket is felvesznek-e. Amennyiben ugyanaz szerepel az attribútum összes cellájában, akkor az adott oszlopot eldobja, így a nem informatív attribútumoktól mentes, keskenyebb adatszerkezetet kapunk a kimeneten.

Elkészült egy lineáris regressziót illesztő folyamat is, ami megvizsgálja, hogy a mért értékek illeszkednek-e az egyenesre. Ez alapján három címkével jelöli meg az értékeket: helyes, gyanús és rossz. Helyes egy mért érték, ha illeszkedik az egyenesre, gyanús, ha maximum 10%-kal tér el az elvárttól, egyébként rossz.

Az ismertetett adattisztító/adatelemző folyamatok nem csak külön-külön hasznosak, hanem egymás után kapcsolva, összetett folyamatként is megállják a helyüket.

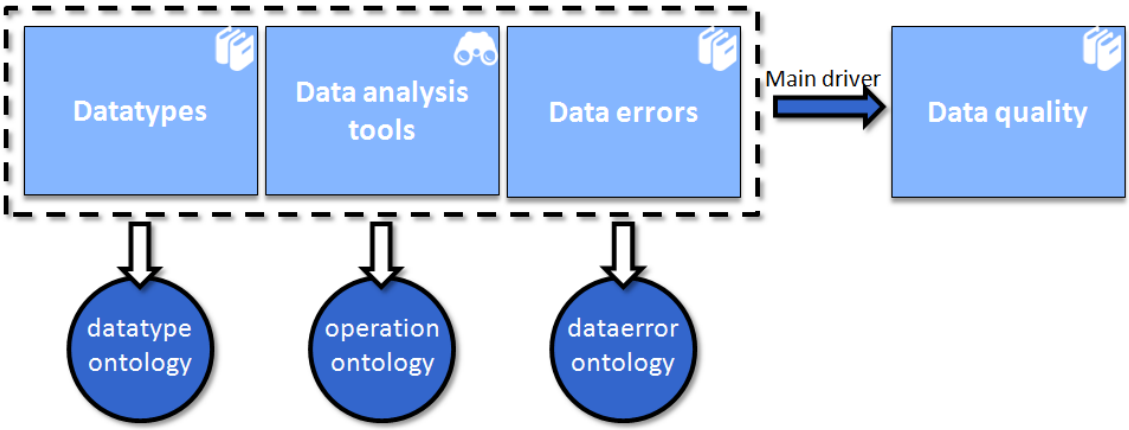
4.2. Metamodell leírása ontológiával

Az adattisztító/adatelemző folyamatok leírására, az adathibák kiküszöbölési lehetőségeinek megfogalmazására illetve az egyes adatelemző lépések adattulajdonságmódosító hatásának kifejezésérére ontológiát használtunk. Azért esett az ontológiával történő modellezésre a választás, mert jól támogatja a különböző bonyolultságú lekérdezések megfogalmazását, illetve könnyedén hozzákapcsolhatóak a már létező (pl. mérési környezetet leíró) ontológiákhoz.

Először azok az ontológiák kerülnek bemutatásra, amelyekből a fent említett három terület modellezését biztosító ontológiák építkeznek (4.2.1. fejezet), majd ezek után következnek az összetettek.

4.2.1. Adattípusok, adathibatípusok, adattulajdonságok és műveletek

A 4.1. ábrán láthatjuk, hogy melyik alapvető ontológia mely, a 3. fejezetben bemutatott ismeretre építkezik, vagyis mi szolgált alapul a modell felépítésese során. (A RapidMiner operátorok kategorizálása a 4.4. ábrán látható.)



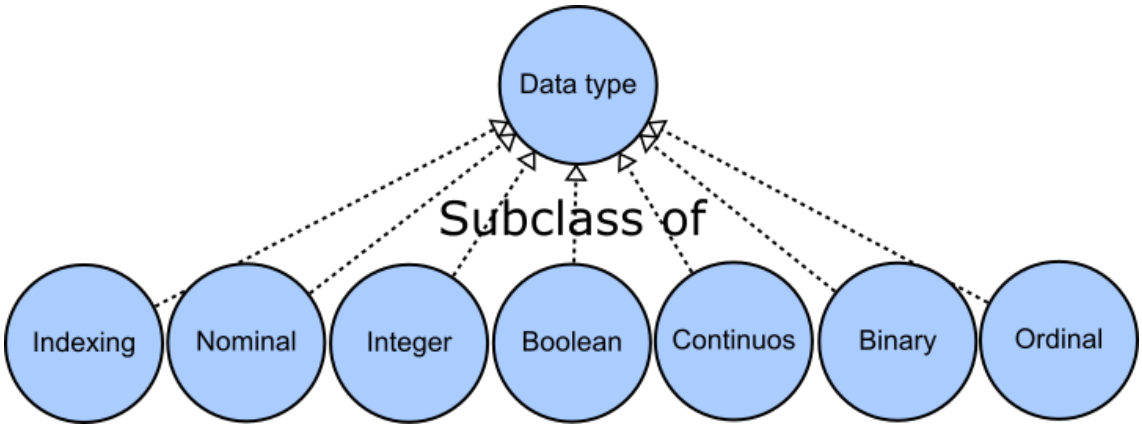
4.1. ábra. Adattípus, műveletek, adathiba

Adattípusok

A 3.3. fejezetben ismertetett adattípusok egy-egy osztályt alkotnak a *DataType* ontológiában (4.2. ábra). Az adattípusok modellezésére azért volt szükség, hogy az adatszerkezet attribútumaihoz és az operátorok paramétereihez adattípusokat rendelhessünk.

Adathibatípusok

A 3.4. ábrán látható, korábban ismertetett adathiba taxonómia került modellezésre a *DataError* ontológiában. Az adathibák modellezésére az adathibahatások kikü-

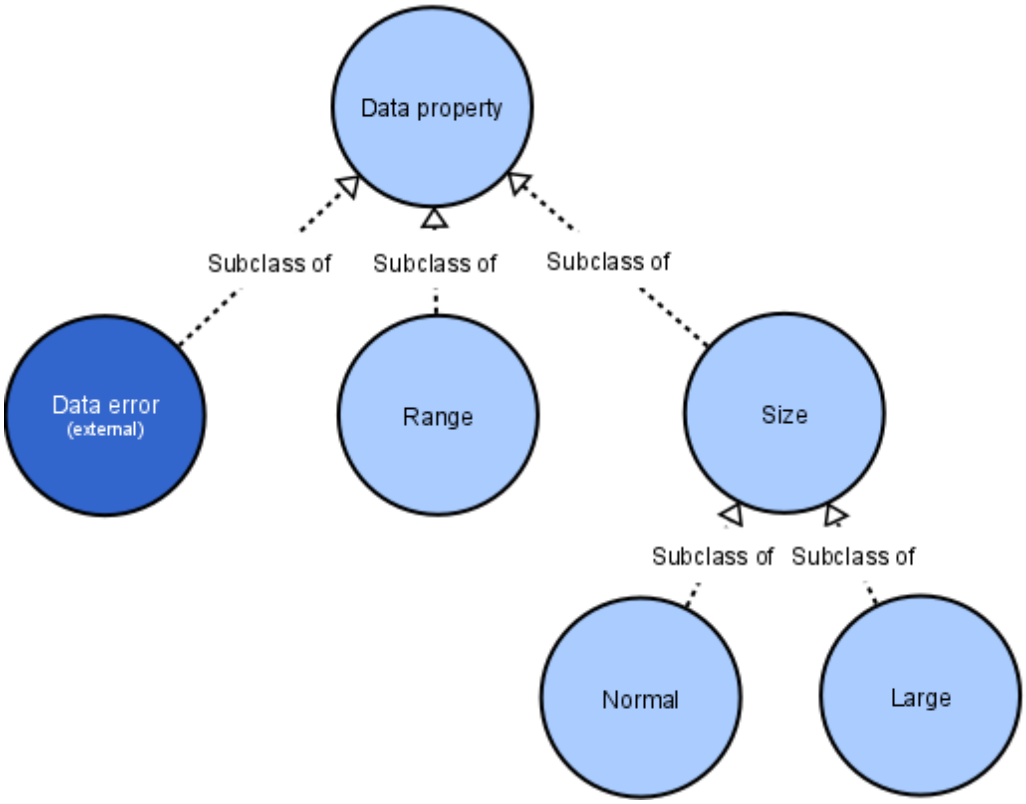


4.2. ábra. Adattípust leíró ontológia

szöbölési lehetőségeinek felírásához volt szükség, valamint a következő, adattulajdonságok ontológia alosztályát képezik. (A *DataError* ontológiáról nem került be ábra, mert teljes egészében a 3.4. ábrán látható taxonómiát írja le.)

Adattulajdonságok

Különböző tulajdonságokkal jellemezhetjük az adattisztító/adatelemző folyamaton végighaladó adatot. Egy adatnak lehetnek adathibái, értékkészlete és mérete. Ezeket a jellemzőket írja le a *DataProperty* ontológia (4.3. ábra). Az adattulajdonságok modellezésére az operátorok adattulajdonság-transzformációs szabályainak megfogalmazásához volt szükség.



4.3. ábra. Adattulajdonságokat leíró ontológia

Adattisztító/adatelemző műveletek

Az *Operation* ontológiában az egyes lépéstípusok/műveletek (operation) kerültek definiálásra. A lépéstípusok meghatározásához a 3.1.1. fejezet kritériumainak megfelelő eszközök operátorait/elemi lépéstípusait tanulmányoztuk, majd működésük alapján kategorizáltuk őket. Néhány RapidMiner operátor besorolása a 4.4. ábrán látható. A kategóriákat és az lépéstípusok kategóriába sorolását is az *Operation* ontológia (4.5. ábra) írja le. A későbbiek során ez a kategorizálás lehetőséget biztosít majd folyamatok felépítésének ellenőrzésére.

4.2.2. Adattisztító/adatelemző folyamat

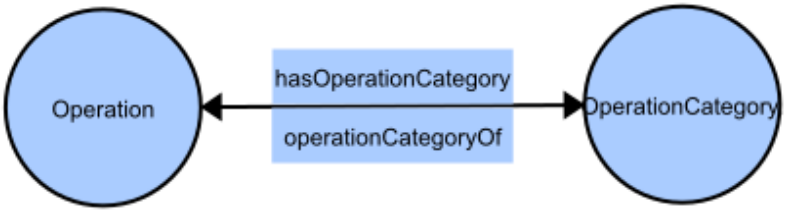
Egy adatelemző folyamat leírásához szükséges osztályok és kapcsolataik a 4.6. ábrán olvashatóak. Egy folyamat (*Workflow*) lépésekből (*Step*) áll. Egy lépés vagy elemi lépés (*Elementary step*), vagy pedig egy folyamat (*Workflow*). Egy lépést(*Step*) folyamatként (*Workflow*) azonosítunk akkor, ha alfolyamatként jelenik meg az adattisztító/adatelemző folyamatban. Minden lépéshez tartozik egy művelet (*Operation*), azaz egy operátor egy lépés (*Step*) és művelet (*Operation*) összessége. Erre azért volt szükség, hogy a későbbiekben egy lépéshez akár több műveletet is hozzá lehessen rendelni vagy egy lépéshez tartozó műveletet más, akár robusztusabb (3.1.3. fejezet) működésű megvalósítással lehessen helyettesíteni. A lépések kaphatnak bemeneti paramétereket (*Parameter*) és paraméter listákat (*ParameterList*), amik paraméterekből állnak. Ezen kívül az egyes lépéseknek vannak bemeneti és kimeneti portjaik (Port). Egy kapcsolat (*Connection*) két portot köt össze. A kapcsolatokon adattáblák (*Data*) haladnak keresztül. Az adattáblák attribútumokból (*Attribute*) állnak. A paramétereknek és az attribútumoknak van típusa (*DataType*).

Nézzük meg a következő egyszerű példán (4.7. ábra) azt, hogy mit jelentenek valójában az egyes osztályok. A *Read Excel* egy elemi lépés, paraméterként megadható neki, hogy honnan olvassa be az állományt. A *Loop Attributes* egy alfolyamat, mert maga is tartalmaz lépéseket, amelyeket a táblázat minden attribútumán végrehajt. A két lépés portjait egy kapcsolat köti össze.

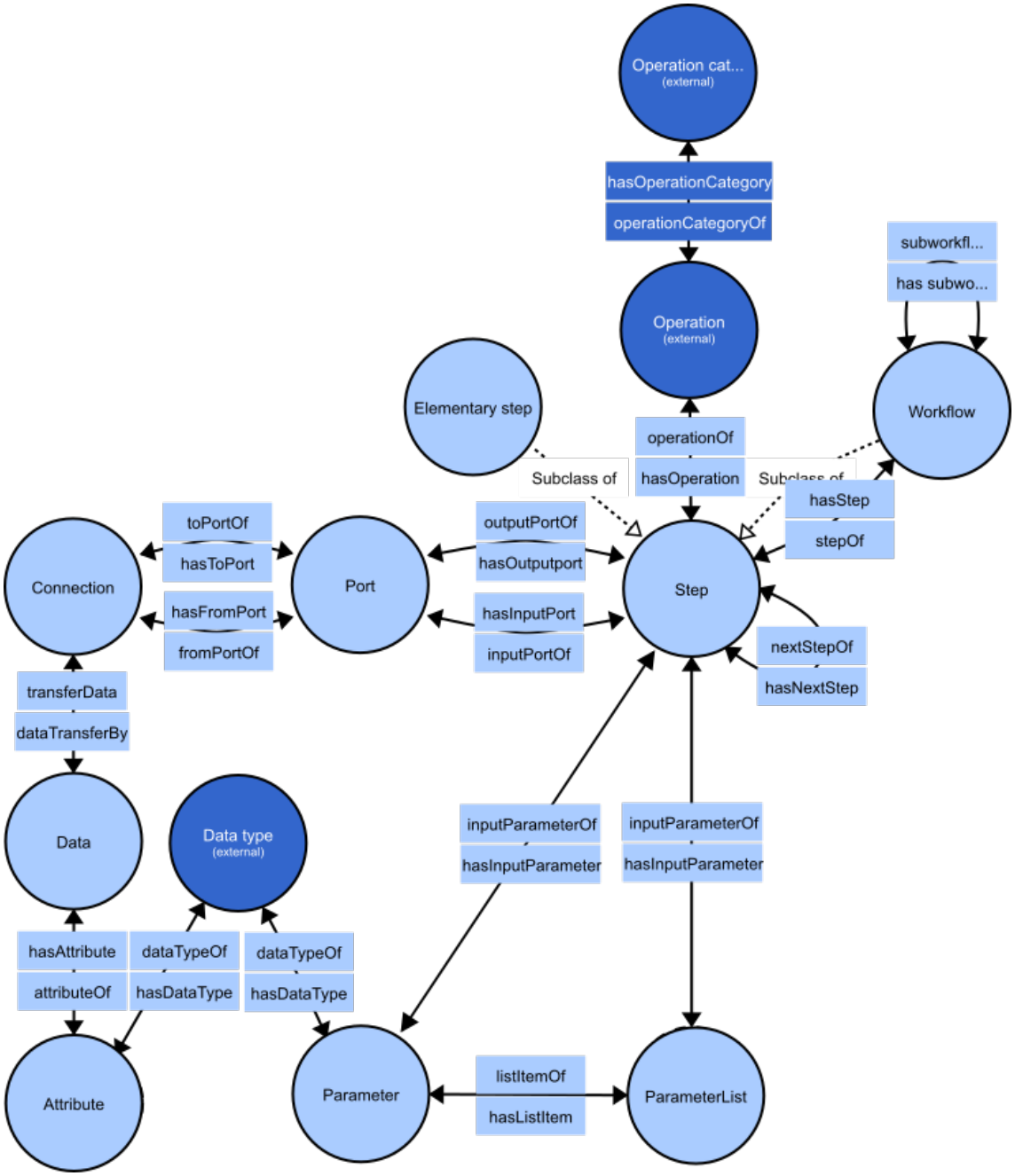
A *Workflow* ontológia definiálásakor fontos szempont volt, hogy a megközelítés ne csak RapidMiner folyamatokra működjön, hanem akár más specializált munkafolyamat alapú adatelemző eszközökkel vagy akár R-scriptekkel megírt adatelemző folyamatra is (6.1. ábra).

	Import	SetOperation	Filter	Visualization	DataCleaning	DataGeneration	ProcessControl	ValueModification	CollectionOperation	Rotation	AttributeOperation	Type Conversion	NameAndRoleModification	Aggregation
Agregate														x
Append		x												
Branch							x							
De-Pivot										x				
Extract Macro						x								
Fill Data Gaps					x									
Filter Example Range			x											
Filter Examples			x											
Generate Attributes						x								
Generate Empty Attribute						x								
Generate Macro						x								
Loop							x							
Loop Attributes							x							
Loop Files							x							
Multiply							x							
Nominal to Date												x		
Nominal to Numerical												x		
Pivot										x				
Process							x							
Read CSV	x													
Read Excel	x													
Recall							x							
Remember							x							
Remove Duplicates			x											
Rename													x	
Replace Missing Values					x									
Select									x					
Select Attributes											x			
Set Minus		x												
Set Role													x	
Set Data								x						
Split								x						
Subprocess							x							
Transpose										x				

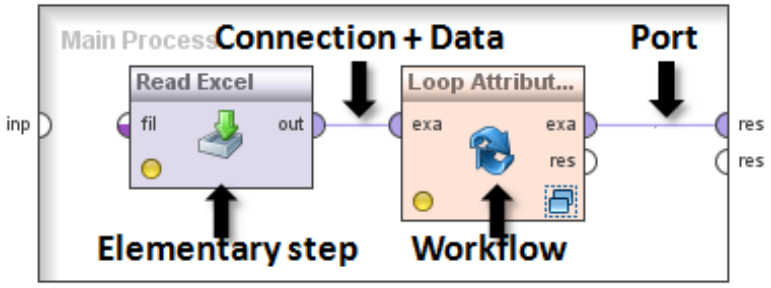
4.4. ábra. RapidMiner operátorok kategóriákba sorolása



4.5. ábra. Műveleteket leíró ontológia



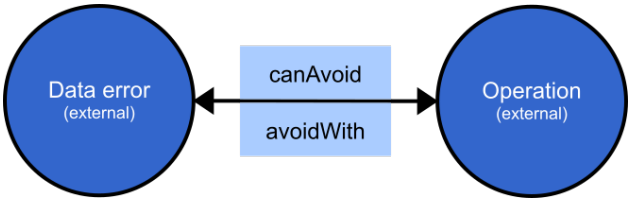
4.6. ábra. Adattisztító/adatelemző folyamatot leíró ontológia



4.7. ábra. Ontológia osztályok azonosítása egy rövid RapidMiner folyamaton

4.2.3. Adathibák kiküszöbölési lehetőségei

Az *ErrorProtection* ontológia (4.8. ábra) meghatározza, hogy milyen adathibát (*DataError*) hogyan lehet kivédeni. Ennek felírására azért van szükség, mert így a hibaterjedés-analízis után javaslatot tehetünk a felhasználónak arra, hogy milyen típusú lépés (*Operation*) beiktatásával küszöbölhető ki az adott adathiba hatása. Ez azt jelenti, ha észleltük, hogy a kimenetünk duplikátumokat (adathiba) tartalmaz, akkor egy duplikátummentesítő lépés beiktatásával ennek kimenetre gyakorolt hatása kiküszöbölhető.

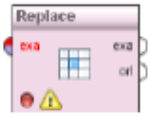
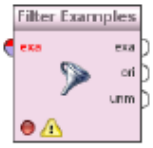
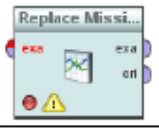


4.8. ábra. Adathibák kiküszöbölési lehetőségét leíró ontológia

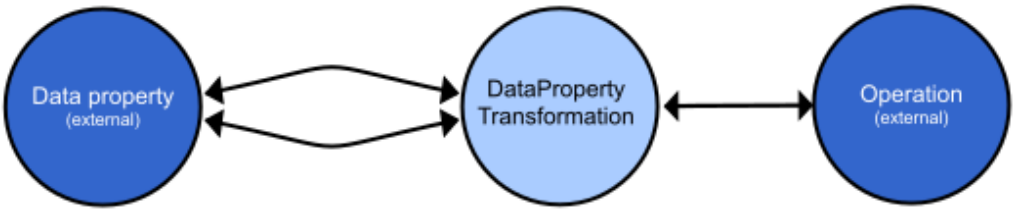
Az esettanulmány során (2.6. fejezet) láthattunk már pár lehetőséget erre. A 4.9. ábrán mutatunk néhány példát arra, hogy az egyes adathibák (3.4. ábra) kimenetre gyakorolt hatása milyen RapidMiner operátorokkal küszöbölhető ki. (A táblázat költség oszlopára az 5.6. fejezetben térünk ki.)

4.2.4. Operátorok adattulajdonság transzformációja

A hibaterjedés-analízis miatt azt is fontos definiálni, hogy az egyes operátorok mely adattulajdonságot mivé transzformálják. Előfordulhat, hogy egy *Operation* egy bizonyos típusú adathibából egy másik típusút csinál. A *DataPropertyTransformation* ontológiával (4.10. ábra) leírható, hogy egy adott *Operation* a bemenetén megjelenő adat tulajdonságát milyen tulajdonsággá alakítja, vagyis az *Operation* elvégzése után milyen tulajdonság jellemzi a kimeneten megjelenő adatot. Például, ha valahol egy cellában hibás érték van (*Cell value error*), akkor egy filter típusú lépés hiányzó sor (*Missing observation*) típusú adathibává alakíthatja, ha éppen az az attribútum szerepel a szűrőfeltételben, ahol a cella értéke hibás.

Adathiba	RapidMiner Operátor	Operátor költsége
• Duplikátumok (Duplicate)	 Remove Duplicates	$O(n^2*m)$
• Hiányzó adatok (Missing Value)	 Replace Missing Values	$O(n*m)$
• Elgépelés következtében keletkezett hibák (Typos) • Helyesírási hibák (Misspellings)	 Replace	$O(n*m)$
• Felesleges, ismétlődő oszlopok (Structural feature/Superfluous)	 Remove Correlated Attributes	$O(n*m^2)$
• Értéktartományra vonatkozó kényszer megsértésére (Min-Max) • Attribútumok közötti függőségi kényszer megsértése (ViolatedAttributeDependency)	 Filter Examples	$O(n*m)$
• Dátum helytelen értelmezéséből származó hibák (Parsing)	 Nominal to Date	$O(n)$
• Egy cellába több érték került bevitelre (Embedded Values)	 Split	$O(n*m)$
• Hiányzó megfigyelések (Observation/Missing)	 Fill Data Gaps	$O(n)$
	 Replace Missing Values (Series)	$O(n)$

4.9. ábra. Adathibák kiküszöbölése RapidMiner operátorokkal



4.10. ábra. Operátorok adattulajdonság-transzformációját leíró ontológia

A 4.11. táblázatban láthatunk példákat a különböző operátorok adattulajdonság-transzformációira. Az első oszlopban a bemenetre jellemző adattulajdonságot látjuk, a másodikban az operátort, az utolsóban pedig azt a tulajdonságot, amivé a bemenetet jellemző tulajdonság alakult az operátor hatására.

Dataproperties in the input	Operation	Dataproperties in the output
Cell value error in filtering cells	Filter	Row error
Cell value error in aggregate or groupby cell	Aggregate	Row error or cell value error in aggregate cell
Cell value error	Pivot	Cell value error
Row error		Structural error
Structural error		Row error
Cell value error	Split	Cell value or Structural error
Row error	Join	Row errors or Cell value error(NAs) based on join type

4.11. ábra. Operátorok adattulajdonság-transzformációja

A megvalósított rendszerben ezeket a szabályokat egy saját leírónyelven kell megadni. Ennek módjáról bővebben az 5.3. fejezetben írunk. Ezek a szabályok azonban előállíthatóak a definiált adattulajdonság transzformációs ontológiából is. Ennek előnye az, hogy így egységes módon kezeljük a szabályokat az adattulajdonságokkal és a folyamattal, valamint így a szabályokat csak egyszer kell leírni, ami után azokat könnyen alkalmazni lehet számos folyamatra.

A 4.11. táblázatban látható pár lépéstípus adathiba transzformációs szabályai amiket a tervezés, és az ismertetett mintafolyamat vizsgálata közben vettünk fel. Ezeken a lépéseken kívül felvettünk számos további lépéstípust is, melyek nem változtatják meg az adatok tulajdonságait, ezekre nem kell szabályokat felvenni, de modellezésükre szükség van a folyamat struktúrájának megtartása érdekében. Ilyen lépések például az *append*, a *multiply*, amelyek nem végeznek semmilyen műveletet az adatokon, csak annak irányításában segítenek.

4.3. Hibaterjedés analízis

Egy adatelemző folyamat futása során számos hiba léphet fel. A bemeneti adathibák okozhatnak hibás kimeneteket, vagy okozhatják a folyamat teljes elakadását is. Ezen hibák többségét kivédhetjük, ha a folyamat megfelelő helyére hibajavító lépéseket iktatunk be. Az optimális hely ezen lépések számára azonban komplex folyamatok esetén kézi módszerekkel nehezen azonosítható.

Erre a problémára adhat megoldást a hibaterjedés analízis módszere. A módszer segítségével megvizsgáljuk, hogy az esetleges bemeneti hibák, hogy terjednek végig lépésről lépésre a folyamaton, észlelve azokat a pontokat, ahol a hibák további problémákat okoznak, illetve azokat ahol a folyamat kijavítja azokat. Ebből

az információból már könnyen meghatározható, hol van az a pont, ahová be kell iktatni valamilyen hibajavító lépést, amellyel megelőzhetőek a kimenet hibái. Ehhez a folyamatot úgy kell modellezni, mint lépések halmaza, amelyek egymással adatátadással kommunikálnak, így a hibák is az adatok átadásával terjedhetnek. Az analízis elvégzéséhez még le kell írni az egyes lépések belső működését is, hogy milyen bemenő adathibára, hogyan reagálnak. Ezt a viselkedést többféleképpen le lehet írni, mi erre egy tranzíció alapú megközelítést választottunk, amelyben egyszerű állapotátmenetekkel írjuk le a viselkedést.

A felhasználónak le kell írni minden lépéstípusra, hogy azok kimenetén milyen típusú hibát okoz egy bemeneti hiba (bemeneti hiba –> kimeneti hiba alakban). Az egyszerűbb felírás érdekében támogatjuk összetett kifejezések leírását is, így felírhatóak olyan szabályok, amelyekben felsoroljuk az összes lehetséges okát egy hibának, vagy akár olyan komplex összefüggések is, hogy egy hiba megjelenéséhez a bemeneten legalább n db hibának jelen kell lennie. Adatelemzési folyamatok esetében elég csak a kimeneti és bemeneti adatok közti összefüggéseket felírni, mivel az adatelemző folyamatok lépései állapotmentesek.

A dolgozat során elkészítettünk egy a [26] disszertációban leírt módszerhez hasonlóan működő eszközt, amellyel elvégezhető ez az analízis. Az analízis végrehajtásakor mindig a legrosszabb esetet vesszük figyelembe, azaz minden adathibáról azt feltételezzük, hogy ha okozhat egy eltérést a kimenetben, akkor azt okozni is fogja. Erre példa lehet egy medián számítás, amely esetében a hibaterjedési szabály azt mondja ki, hogy a medián hibáját okozza, ha létezik adathiba a bemeneti adatokban, pedig valójában ehhez nem mindig elég léteznie egy hibás adatnak, hanem az adatok többségének hibásnak kell lennie.

4.3.1. Hibaterjedés analízis eredményei

A hibaterjedési probléma felírásához szükséges összegyűjteni, milyen módon lehetnek hibásak a bemeneti adatok, és az eredmények. A lehetséges általános hibákat összegyűjtöttük, rendszereztük egy taxonómiába, így ezt felhasználva könnyen felírhatóak a lehetséges hibák bármely speciális esetben. Ezt a taxonómiát a 3.2.2. fejezetben részletezzük.

Ezt a taxonómiát felhasználva már egy kényszerkielégítési problémára segítségével lépésről lépésre lekövethető a hibák terjedése, változása a folyamatban. Ekkor az analízis eredményeként megkapjuk az összes lehetséges kimeneti hibáját a folyamatnak, illetve az ezekhez vezető lefutási utakat. Ennek az információnak a segítségével már meghatározható, hol kell javítani a hibákat, ahol az lehetséges.

5. fejezet

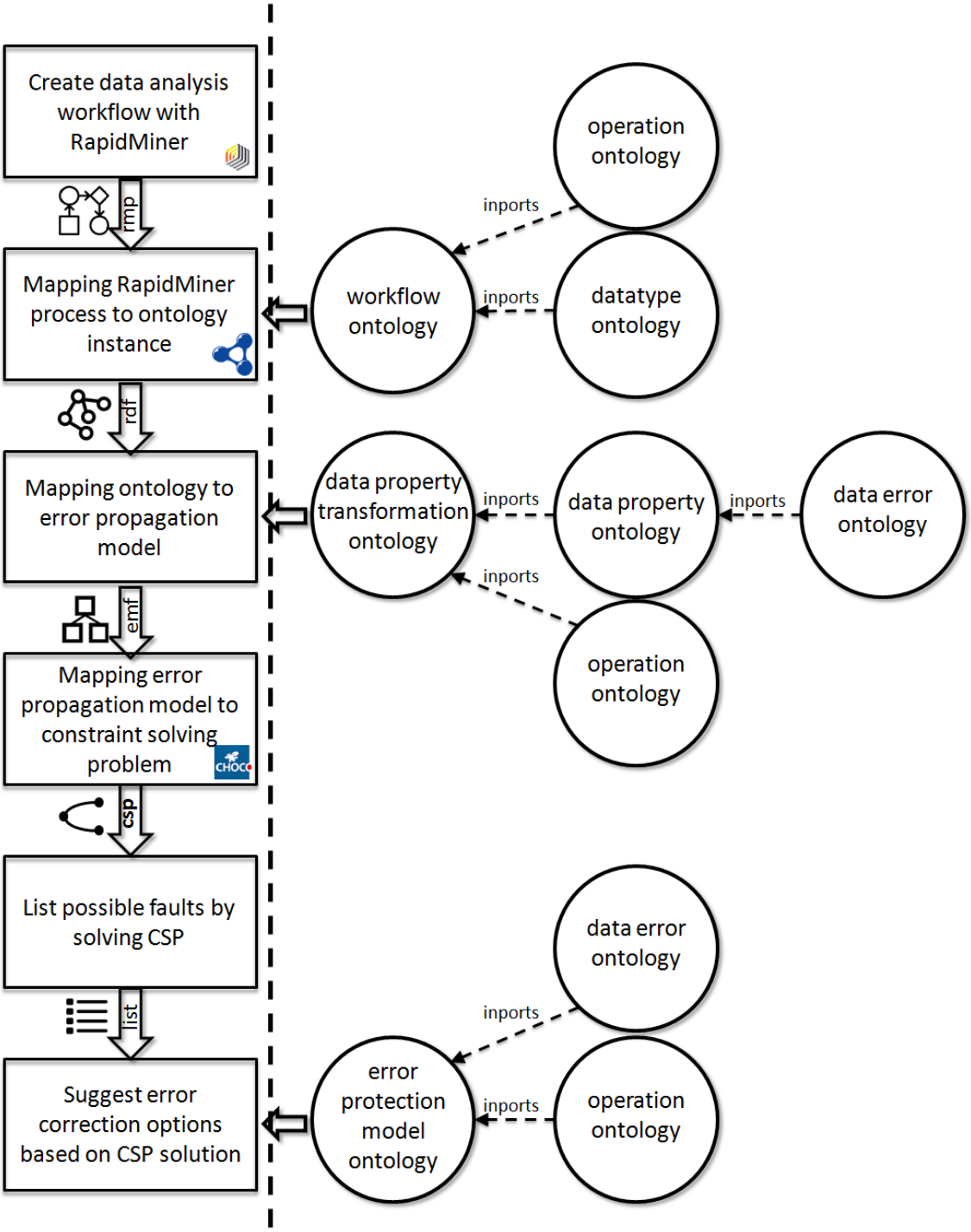
Megvalósítás

A dolgozat elkészítése során elkészítettük a módszer egy implementációját is, ennek a technikai részleteit fejtjük ki ebben a fejezetben. Az implementáció során törekedtünk a minél általánosabb megoldásra, így a módszer és az implementáció többi része könnyen használható más eszközökkel, például bemenetként más eszközből származó modellel.

A megvalósított rendszert, a RapidMiner adatelemző eszköz folyamatainak vizsgálatára képes. Az elkészült rendszer felépítése látható az 5.1. ábrán. A RapidMiner eszköz segítségével állítjuk össze az adatelemzési folyamatokat. A felírt folyamatokat az egységes kezelhetőség érdekében egy ontológiára képezzük. Erről az ontológiáról a 4.2.2. fejezetben, az erre történő automatikus leképezésről pedig az 5.1. fejezetben írunk. Az ontológia modellből az analízis elvégzéséhez elő kell állítani egy hibaterjedési modellt, ami már tartalmazza a hibaterjedési szabályokat. Ezek a szabályok egy ontológiából származnak, amit a 4.2.4. fejezetben mutatunk be. A két ontológia összekapcsolásával készítjük el a hibaterjedési modellt, ezt a leképezést az 5.3. fejezetben mutatjuk be. A modellt ez után egy megoldható matematikai formára hozzuk, egy kényszerkielégítési problémaként reprezentáljuk. A leképezés módjáról az 5.4. fejezetben írunk. A matematikai probléma megoldásával megkapjuk a rendszer lehetséges futásait, amelyet kiértékelve akár javításokat is végezhetünk a folyamaton. Erről az 5.5. fejezetben írunk.

5.1. Ontológia példánymodell generálása az adattisztítási folyamatból

Amennyiben rendelkezésre áll a RapidMiner eszközzel megalkotott folyamat (RapidMiner process - rmp) és az adatelemző folyamatokat leíró (*workflow*) ontológia metamodell (rdf), akkor a tetszőlegesen megválasztott RapidMiner folyamat leképezhető a metamodellnek megfelelő ontológia példánymodellé. Leképzéshez a RapidMiner Java API biztosítja a folyamatelemek, paraméterek, adatszerkezetek elérését. Az ontológia példánymodell generáláshoz az Apache Jena RDF API került felhasználásra. Tehát a leképezés során az *rmp* állományból előáll egy annak megfelelő *rdf* állomány.



5.1. ábra. A megvalósított rendszer áttekintése

5.2. Hibaterjedés analízis modellezési megközelítése

A megvalósítás során egy minél általánosabb hibaterjedés analízis eszközt szerettünk volna elkészíteni.

5.2.1. Hasonló eszközök az irodalomban

Az irodalomban többször foglalkoztak már hasonló hibaterjedés analízis megközelítéssel. A tervezés során ezeket megvizsgáltuk, összehasonlítottuk saját megközelítésünkkel. A létező megközelítések egy részéről született már összehasonlító elemzés is [2] [12]. A lényeges, saját megközelítésünkhöz hasonló megközelítéseket, itt röviden bemutatjuk, de az itt felsorolt megközelítéseken kívül továbbiakat is megvizsgáltunk. Ezekre nem térünk ki bővebben, mert többségük más modellezési megközelítést alkalmaz, kvantitatív analízist mutatnak be ([9], [23], [27]).

Fault Propagation and Transformation Calculus

A [37] cikkben ismertetett Fault Propagation and Transformation Calculus (FPTC) definiál egy leírónyelvet általános hibaterjedési problémákra, valamint egy fixpont algoritmust azok megoldására. A nyelv igen egyszerű, csak szimpla átmeneteket lehet definiálni a hibaállapotok között, így komplexebb modellek leírása igen bonyolulttá válhat benne, mivel nincs támogatás a vizsgálandó rendszer felépítésének leírására. A saját rendszerünkben ezzel szemben megpróbáltuk a lehető legjobban támogatni azt, így a szabályok könnyebb, strukturáltabb leírását is.

A probléma megoldására egy fixpont algoritmust ajánlanak. Az algoritmus segítségével felderíthető egy feltételezett hiba összes lehetséges forrása, azonban azoknak a kialakulási útja nem. Így megkapjuk az összes lehetséges hibaforrást, azonban arról nem kapunk információt, hogy az, milyen úton terjedt végig a folyamaton. Ez sok esetben elég lehet, azonban ahhoz, hogy adatelemző folyamatok esetében, megtaláljuk, hová kell beiktatnunk hibajavító lépéseket nem, hiszen ahhoz tudnunk kell, hol váltja ki a hibahatást a bemeneti hiba. Ezért a munkánkban a fixpont algoritmus helyett, egy korlátkielégítési probléma segítségével oldottuk meg a feladatot, amely képes meghatározni a hibák kialakulásának módját is.

FI4FA (Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures Analysis)

Az FPTC formalizmusára több munka is épül. Az egyik ilyen a FI4FA [11]. A FI4FA egy általános hibamodellt ad a hibaterjedési szabályok mögé, a hibák klasszikus osztályozása alapján, amelyet már bemutattunk a 3.2.2. fejezetben. A FI4FA ezt egy tranzakció alapú komponensekből felépülő rendszeren mutatja be. Ezen jól alkalmazható ez a hibaosztályozás, de adathibák esetében nem, ezért készítettünk egy saját hibataxonómiát, amit már a 3.2.2. fejezetben bemutattunk.

A cikkben tranzakciós rendszerekre definiálnak hibamodellt, ennek minden elemére megpróbálnak adni egy tranzakciós szemantikát, amellyel elnyomható vagy javítható az adott hiba. Megfelelő tranzakciós szabályokkal betarthatóak az ACID (Atomicity, Consistency, Isolation and Durability) tulajdonságok, amelyek javíthatnak az előállított adatok minőségén. Azonban adatelemző folyamatok esetében nem

ez a helyzet, hiszen nem a folyamat készíti el az adatokat, így nem a folyamat okozza az adathibákat. Teljesen más típusú hibákat kell kezelnünk, mert már a folyamat bemenetére hibás adatok érkeznek, a FI4FA pedig a rendszer működése során keletkező hibákat modellezi.

A probléma megoldására az FPTC algoritmusait használják.

CHESS-FLA

A [10] cikkben bemutatnak egy környezetet, amellyel a FI4FA eszközt integrálják a CHESS környezettel. A CHESS egy nyílt forráskódú, Eclipse alapú modellező környezet, amelyben egy saját nyelven írhatóak le komponens alapú rendszerek. A CHESS-FLA kiegészíti ezt a nyelvet hibaterjedési információkkal, valamint nyújt egy leképezést a FI4FA bemenetére, így a CHESS környezetben felépített modellen tudunk futtatni egy FI4FA elemzést.

HIP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies)

A HIP-HOPS [25] környezet segítségével hierarchikusan modellezhetünk komponens alapú rendszereket. A hibaterjedési szabályok leírásához a komponensek közt futó összes kapcsolatra le kell írni minden hibalehetőségre a terjedési szabályokat. Az így elkészült rendszermodellből a rendszer képes hibafákat generálni, ezzel támogatva egy FMEA elemzést.

GFL (Generalized Failure Logic)

A [40] cikkben bemutatják a HIP-HOPS-nak egy kiegészítését, amely a hibaterjedési szabályok leírását próbálja egyszerűsíteni. Segítségével változócsoportokat lehet definiálni, használni lehet kifejezéseget, amellyel a hibamódok számosságát vizsgálhatjuk (az összes egyszerre, a többség, néhány, bármennyi). Ezzel egy jóval erősebb leírónyelvet hoz létre a HIP-HOPS fölé.

AADL (Architecture Analysis and Design Language)

Az AADL alapvetően egy architektúra leírónyelv, de definiálja a hibaterjedési szabályok leírását is a [13]. Az AADL egy hatékonyt modellt ad rendszerek leírására, komponensek definiálhatóak, valamint a közöttük lévő kapcsolatok. Az egyes komponensekre felvehetünk szabályokat, amelyekkel leírhatjuk, hogy a bemenő hibák hatására milyen állapotba kerül egy komponens valamint, hogy ennek hatására, mit ad ki a kimenetein.

A hibamodellt alapvetően Petri hálókra és hibafákra képes lefordítani. Így analízis is elvégezhető segítségével a generált Petri hálókon.

UML MARTE DAM

AZ UML MARTE DAM [1] [4] [6] [5] egy UML profil, amely segítségével UML diagramokkal írhatjuk le komponensekből felépülő rendszereket. A diagramokra megjegyzések formájában írhatóak fel, a hibaterjedési információk, valamint a diagramok közti kapcsolatok. Ez egy meglehetősen nehezen használható megoldás,

hiszen strukturált szöveget kell írni a megjegyzésekbe. A hibaterjedési probléma megoldását nem definiálták a profilban.

AltaRica

Az AltaRica [3] [20] egy az AADL-hez hasonló leírónyelv. Az AADL-el ellentétben azonban ez egy nyílt forráskódú projekt, így több munka is feldolgozza, amelyek különféle analíziseket képesek elvégezni AltaRica modelleken, például a [19] cikkben Petri háló modellekké konvertálják az AltaRica leírásokat, ennek segítségével vizsgálnak repülőipari rendszereket.

5.3. Ontológiák leképezése komponens alapú modellre

Miután leképeztük a adatelemző folyamatot egy általános ontológia modellre, azt tovább kell fordítanunk egy olyan modellre, amely már alkalmas a hibaterjedés analízis elvégzéséhez. Ehhez az szükséges, hogy a modell tartalmazza a hibaterjedési információkat is. Ezek azonban nem nyerhetőek ki egy csak a folyamatot tartalmazó ontológiából, ezért készítettünk egy másik ontológiát, amely tartalmazza minden folyamatlépés-típushoz, a hibaterjedési szabályokat (4.2.4 fejezet). A hibaterjedési szabályok leírják, hogy az egyes lépéstípusok, milyen bemeneti hibákra, hogyan reagálnak, milyen más hibává válhatnak a kimeneten.

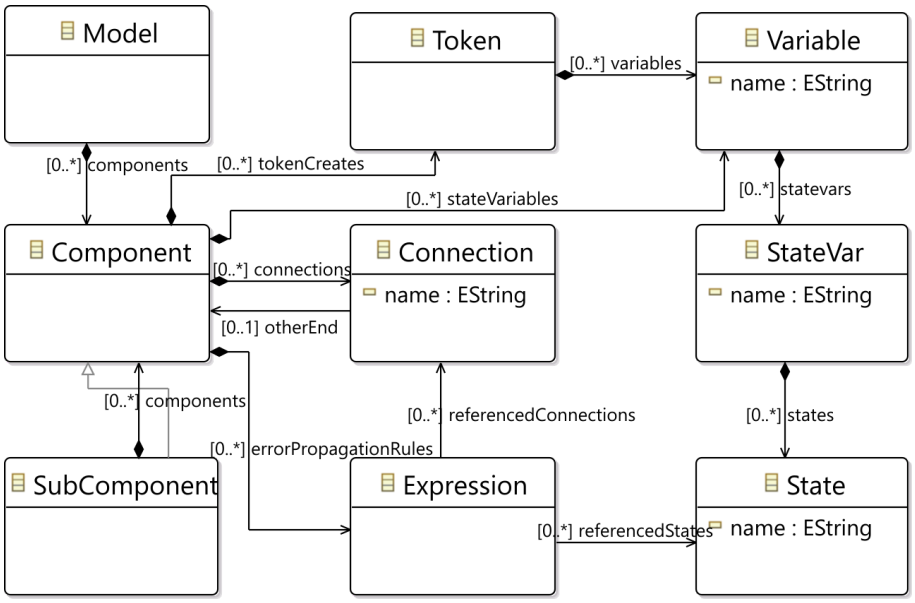
Ebből a két ontológiából már előállítható a hibaterjedés analízis bemeneti modellje.

5.3.1. Komponens alapú hibaterjedési modell

A hibaterjedés analízis megoldásához egy általános eszközt szerettünk volna készíteni, amely segítségével nem csak adatelemzési folyamatokat vizsgálhatunk, mivel hibaterjedés analízis több más területen (folyamat alapú rendszerek, komplex infrastruktúrák) is hasznos lehet [35]. Egy általános eszköz készítéséhez egy általános bemeneti modellt kellett készítenünk. Erre legalkalmasabbnak egy komponens alapú leírást találtunk, az ennek megfelelően megalkotott modell egyszerűsített felépítése látható az 5.2. ábrán.

A modell *komponens*eket tartalmaz, amelyek függhetnek egymástól. Ebben az esetben komponens –> komponens irányú hibaterjedst írhatunk le. Ha az egyik komponensnek aktív valamely hibamódja, akkor a tőle függő komponensek is hibásak lesznek a felírt szabályoknak megfelelően. A modell ezen felül *token*eket tartalmaz. A tokeneket komponensek adhatják át egymás között. Az adatelemző folyamatok esetében az adattáblák például tekinthetőek tokennek. A tokenek rendelkeznek valamilyen kezdőállapottal, amikor bekerülnek a rendszerbe, ezt követően minden komponens amelyen áthalad módosíthatja a token állapotát a hibaterjedési szabályoknak megfelelően.

A hibaterjedési szabályok felírása Boole-algebrai kifejezésekkel történik. A szabályoknak két fajtáját különböztetjük meg. A hibaterjedést leíró szabályok két részből állnak, egy bal oldalból (feltétel) és egy jobb oldalból (következmény). A bal oldalra felírt kifejezés változói a tokenek komponensbe érkezéskori állapotát vizsgálják, a jobb oldal pedig a token kilépésekor aktív hibamódokat írja le. Így felírhatóak az összefüggések a bemenetek és kimenetek között. A másik típusú szabályok csak



5.2. ábra. Egyszerűsített komponensmodell

egy részből állnak, csak a kimenetre írnak fel általános érvényű összefüggéseket. Ilyen szabályokkal lehet megfogalmazni például egy elágazást, ha leírhatjuk, hogy egyszerre mindig csak egy kimenet aktív. Ilyen szabályok alkalmazhatók lekötések felírására is, amelyek leírhatják az adatok ismert paramétereit, például, hogy tudjuk, nem tartalmaz duplikátumokat az adatsor, ezzel szűrhetünk a megoldások között, hiszen így nem jelennek meg olyan eredmények, ahol egy duplikátum okozott hibát.

A szabályok mindkét oldalára komplex Boole-algebrai kifejezéseket felírhatunk. Erre látható pár példa az 5.3. ábrán, ahol egy szűrést végző komponens szabályait írtuk fel. Az első szabály leírja, hogy mikor lesznek sorhibák a kimeneten: akkor ha már eleve voltak a bemeneten, vagy valamilyen adathiba volt abban a változóban, amire a szűrést végeztük. A másik szabály pedig leírja, mikor nem lesz semmilyen sorhiba a lépés kimenetén. Az F.2. függelékben láthatóak további példák hibaterjedési szabályokra, lekötésekre.

A modellhez elkészítettünk egy leírónyelvet, így a hibaterjedési modellek könnyen felvehetőek egy egyszerű szöveges modellben. A nyelv és így a modell tartalmaz lehetőséget a modell elemeinek típusos leírására, így ha több ugyanúgy vagy hasonlóan viselkedő token vagy komponens van a rendszerben, azok egyszerűen leírhatóak egy típus több példányaként. Ennek a megközelítésnek az a legfőbb előnye, hogy így a rendszer általánosan előforduló elemei összegyűjthetőek egy könyvtárba. Ezután az azonos típusú rendszerek könnyen felírhatóak, a könyvtárban szereplő komponens és tokentípusok példányosításával.

A nyelvben felvehetőek alkomponensek, alfolyamatok is. Ez egyrészt szükséges, hiszen az adatelemző folyamatokban is sok helyen szerepelnek alfolyamatok, másrészt lehetőséget nyújt hierarchikus vizsgálatok elvégzésére is. Ha az alfolyamaton futtatunk egy hibaterjedés analízist, megkapjuk minden lehetséges bemeneti hibára, milyen kimenetet ad a teljes alfolyamat. Így későbbi futások során a fő folyamatban elég az ebből a kimenetből előállított szabályokat leírni az alfolyamathoz az abban szereplő lépések helyett, ezzel jelentősen gyorsítva az analízis futását. Erre egyelőre csak korlátozott támogatást nyújtunk, mert kézzel kell lecserélni az alfolyamatot

egy azt reprezentáló lépésre, ezt a példafolyamatban szereplő *Loop Measurement* alfolyamattal teszteltük. Ez könnyen automatizálható, ezért a továbbfejlesztési terveinkben szerepel is ennek automatizálása.

Az 5.3. ábrán látható egy egyszerű példa, amely egy komponenstípus működését írja le. A típus egy adatelemző folyamat filter lépését írja le, amely egy adott oszlop

```
25 ComponentType Filter{
26     TokenInOut row param Row row
27     TokenInOut filter param DataToken filter
28     Rules{
29         filter.State.value.ExistError | row.State.existence.Error
30                 -> row.State.existence.Error
31         !(filter.State.value.ExistError | row.State.existence.Error)
32                 -> row.State.existence.Ok
33     }
34 }
```

5.3. ábra. Egy komponenstípus leírása

változói szerint szűri a táblát.

A nyelvet az Xtext [41] környezettel készítettük el, így az elkészült eszköz jól használható az Eclipse fejlesztőkörnyezettel.

5.3.2. Adatelemző folyamatok leképezése

Az adatelemző folyamatok leképzésekor két ontológiát használhatunk fel. Egy ontológia leírja a folyamat felépítését, egy másik pedig a lépéstípusok viselkedését.

A lépéstípusok viselkedését leíró ontológia leírja minden típusra, hogy hogy viselkednek az előforduló adathibákra, összegyűjti az adathiba-transzformációs szabályokat. Ezeket a leírásokat komponenstípusokra lehet fordítani, hasonlóakra, mint amelyen az 5.3. ábrán bemutattunk. Hibatranszformációs szabályokból a mintapéldához kapcsolódóan néhányat már bemutattunk a 4.2.4. fejezetben. A dolgozat során leírtuk több lépéstípus hibatranszformációs szabályait (Filter Aggregate, Pivot, Split, Join, ...), illetve azonosítottunk sok olyan tipikusan vezérlési jellegű lépést, amelyek nem változtatnak az adatok hibáin, így nem szükséges szabályokat definiálni hozzájuk (Append, Select Attributes, Multiply, ...). További folyamatok vizsgálatához, amelyekben még nem modellezett operátorokat használunk, le kell írni ezeket a lépéstípusokat. Ehhez azonban csak definiálnunk kell az operátor működését hibaterjedési szempontból, hogy milyen bemenetekre, hogyan reagál, majd ezt az információt beírni az operátorok adattulajdonság transzformációját leíró ontológiába. Leírhatjuk az előbb bemutatott szöveges formalizmussal is, közvetlenül a hibaterjedési modellbe, ez esetben azonban ha következőleg egy másik folyamatban is előfordul az operátor, ott is le kell majd ugyanezt írnunk, így ezt a megoldást egyedi scriptet futtató operátorok esetében érdemes alkalmazni.

A létrejött komponenstípusokat példányosítjuk és összekötjük a folyamat felépítését leíró ontológiának megfelelően. Ezen felül létre kell hozni a folyamaton végighaladó tábláknak és macroknak megfelelő tokeneket. Így a modell már tartalmaz minden információt a hibaterjedés analízis elvégzéséhez. (A példában szereplő folyamat teljes modelljét csatoltjuk az F.2. függelékben.) A függelékben csatolt mintapélda a jobb olvashatóság érdekében kézzel egyszerűsítve lett, egyértelmű, rövidebb változónevekkel kerül bemutatásra, mint amilyenek a RapidMinerből kinyerhetőek.

5.4. A hibaterjedési probléma leképezése

Miután felépítettük a hibaterjedési problémát leíró modellt, azt egy kényszerkielégítési problémaként oldjuk meg. Ehhez a modellből elő kell állítani egy kényszer és változóhalmazt, amelyet megoldva a hibaterjedési probléma megoldását kapjuk meg.

A felírt kényszerkielégítési problémát könnyen megoldhatjuk általános megoldó algoritmusokkal. A dolgozat során a Choco [28] Java alapú általános megoldót használtuk. A Choco használatakor a kényszereket Java objektumok példányosításával lehet létrehozni, így el tudtunk készíteni egy leképezést, amely a folyamatmodellnek megfelelően példányosítja a megfelelő kényszereket, változókat.

A kényszerkielégítési probléma felírásához bool típusú változókat használtunk. A probléma többféle változót tartalmaz:

- Az egyes komponensek (folyamatlépések) állapotát leíró változók:
  - Minden állapotváltozó minden hibamódjához egy bool változó, ami jelzi, hogy aktív-e az adott hibamód.
- A tokenek (adattáblák) állapotát leíró változók:
  - Minden állapotváltozó minden hibamódjához egy bool változó, ami jelzi, hogy a folyamatba lépéskor aktív volt-e a hibamód.
  - Az összes komponenshez, amelyet érinthet az adott token felvesszük a token összes hibamódjához tartozó változót, amely jelzi, hogy aktív volt-e az adott hibamód, mikor a token áthaladt a komponensen.
  - Az összes komponenshez minden ott lehetségesen áthaladó tokenhez egy változót, amely jelzi, hogy az adott futás során tényleg átment-e ott a token.

A változókon felül számos kényszert kell felvenni. A probléma felírásának egyszerűsítése érdekében a szabályok többségének van egy alapértelmezett változata, amelyet felülírhatunk a bemeneti modellben konkrét szabályokkal.

- Van néhány szabály, amely felülírhatatlan, mindenképp aktív:
  - Egy hibamódnak egyszerre csak egy állapota lehet aktív. Erre példa lehet az, hogy a sorszintű hibákat leíró változó értéke egyszerre csak egy lehet az alábbiak közül: [*„Ok”, „felesleges sort tartalmaz”, „hiányzó sort tartalmaz”*].
- A szabályok többsége azonban felülírható az aktuális lefutásnak megfelelően:
  - Ha egy komponens átad egy tokent egy másiknak, egyenlő lesz a két lépésnek az a változója, ami jelzi, hogy áthaladt-e ott a token.
  - A komponens a rajta áthaladó token összes állapotváltozóját átmásoljuk ahhoz a komponenshez, amelyhez átadja a tokent.
- Illetve van egy pár szabály, amelyek csak a modellből származhatnak:

– Két egymást használó komponens állapotai közti szabályok.

Ha a bemeneti modellben szerepel egy egyedi kényszer egy változóra, akkor ahhoz semmilyen alapértelmezett a bemeneti modellből automatikusan generált szabály nem kerül példányosításra. Így felülírhatóak az alapértelmezett szabályok többsége.

Az egyedi szabályok igen sokfélék lehetnek. Gyakorlatilag bármilyen Boole-algebrai kifejezést felvehetünk a komponensek, tokenek állapotváltozói, valamint a tokenek áthaladását jelző változók között. Ilyen szabályra láthattunk példát az 5.3. ábrán. Megfelelő kifejezésekkel így felvehetőek lekötések is, azaz például valamely hibamódot ismerten hibásnak jelölhetjük.

Az ezekből a változókból és kényszerekből felírt probléma megoldása megadja a rendszer (adatelemzési folyamat) összes lehetséges lefutását. Ezekből a lefutási lehetőségekből szűrhetőek ki a számunkra érdekes megoldások.

A könnyebb fejlesztés érdekében módosítottuk a Choco bemeneti interfészét, hogy le lehessen kérdezni az összes felvett kényszert, azoknak eredeti formájában.

5.5. Analízis eredményeinek kiértékelése

Vizsgáljuk meg a 2. fejezetben leírt adatelemzési folyamatot. A folyamat rendelkezik egy a folyamaton kívüli lépéssel, a boxplot rajzolással (A boxplot felépítéséről írtunk a 2.5. fejezetben). A RapidMiner Studio a folyamat kimeneteire tud tetszőleges diagramot rajzolni. Ezt a hibaterjedési modellben legegyszerűbb úgy modellezni, hogy a folyamat után kötünk egy új lépést, ami a rajzolást reprezentálja, és definiálja a boxplot a folyamat kimenetétől függő hibáit. Ehhez definiálni kell egy tokent is, ami reprezentálja a diagramot és annak tulajdonságait. A token típusa fogja összegyűjteni a boxplot lehetséges hibáit. Ezeket a komponenseket láthatjuk leírva az 5.4. ábrán.

A hibaterjedés analízis segítségével több dolgot vizsgálhatunk meg ezen a folyamaton, de példaként tekintsük azt az esetet, ahol feltételezzük, hogy valamelyik boxplot minimumértéke irreális, például negatív. Ebben az esetben az analízis bemeneti modelljében le tudjuk kötni, hogy csak ennek a hibamódnak az okait szeretnénk vizsgálni.

Ha kikötjük, hogy egyszerre csak egy hiba lehet a bemeneti adatokban, akkor az analízis 3 különböző megoldást fog adni:

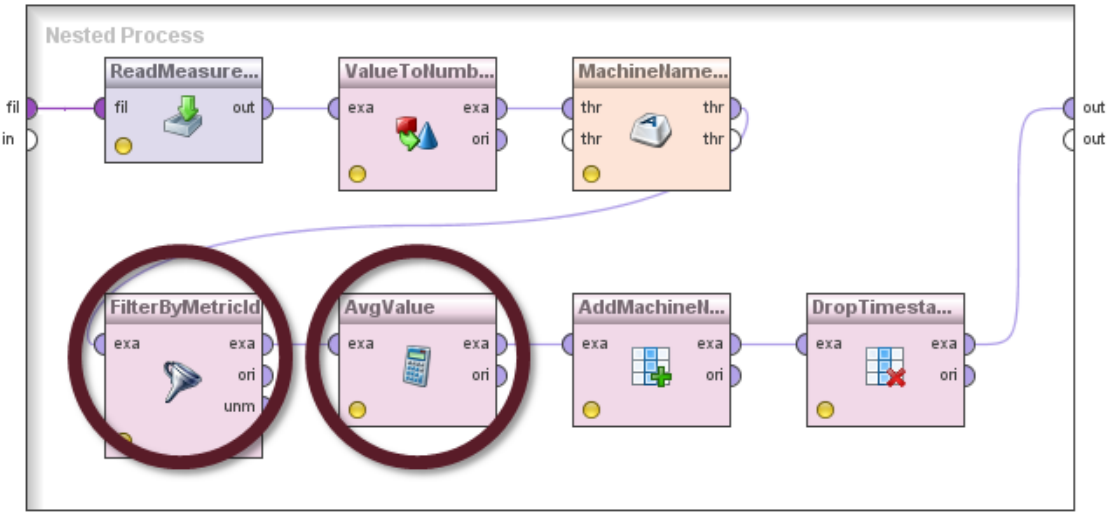
- A bemenetben a metricId oszlopban hiba volt, ami a filter lépésnél sorhibához vezetett, ami a kimenetben okozhatja a minimum változását.
- Eredetileg is volt egy hibás érték a bemenetben.
- Volt plusz sor a bemenetben, ami érvénytelen értéket adhatott.

A hibák hatását az 5.5. ábrán szemléltetjük. Ugyan három hibaok lehetséges, de ezek csak két operátorban okoznak további hibát.

Ha nem kötöttük volna ki, hogy egyszeres hiba lehet csak a bemenetben, akkor is megkapnánk ezeket a megoldásokat, illetve ezen felül ezek összes kombinációját. Mivel ebben az a példában nincs olyan hibamód, amelyhez szükséges lenne a bemenet több hibája, így az egyszeres hibákon felüli megoldások feleslegesek számunkra,

```
VariableType BoxPlot{
    Min(Ok, Error)
    Max(Ok, Error)
    Q1(Ok, Error)
    Q2(Ok, Error)
    Q3(Ok, Error)
    MissingBox(True, False)
    ExtraBox(True, False)
}
ComponentType BoxPlot{
    TokenInOut row param Row row
    TokenInOut value param DataToken value
    TokenInOut category param DataToken category
    TokenOut boxPlot boxPlot
    Create Token boxPlot{
        BoxPlot State
    }
    Rules{
        category.State.value.ExistError
                -> boxPlot.State.ExtraBox.True | boxPlot.State.MissingBox.True
        category.State.value.Ok
                -> boxPlot.State.ExtraBox.False & boxPlot.State.MissingBox.False
        row.State.existence.Error | value.State.value.ExistError
                -> boxPlot.State.Max.Error | boxPlot.State.Min.Error |
                    boxPlot.State.Q1.Error | boxPlot.State.Q2.Error |
                    boxPlot.State.Q3.Error
        row.State.existence.Ok & value.State.value.Ok
                -> !(boxPlot.State.Max.Error | boxPlot.State.Min.Error |
                        boxPlot.State.Q1.Error | boxPlot.State.Q2.Error |
                        boxPlot.State.Q3.Error
                    )
    }
}
```

5.4. ábra. A boxplot működését leíró modellrészlet



5.5. ábra. Adathibák kiküszöbölése RapidMiner operátorokkal

hiszen ha kivédjük az egyszeres hibákat, azzal azoknak összes kombinációját kivédtük.

A felsorolt három hibamódot többféleképpen is kivédhetjük, az általános módszereket lásd a 4.9. ábrán. A hibajavító lépéseket az analízis eredménye alapján

kikereshetjük az adathibák kiküszöbölei lehetőségét leíró ontológiából. hibajavító lépések többnyire igényelnek valamilyen szakterület-specifikus információt, amely segítségével észlelni tudják a hibás adatokat. Ilyen információ lehet például egy értelmezési tartomány. Ezeket az információkat felhasználva javíthatunk az adatminőségen.

- A metricId oszlop hibáját úgy próbálhatjuk meg kivédeni, hogy a bemenetből kiszűrjük az érvénytelen, nem létező azonosítókat, vagy az egyszerű adathibákat észrevesszük és javítjuk például egy replace operátor segítségével. Ez ugyan nem nyújt teljes védettséget, de a hibák többsége kiküszöbölhető vele.

- Ha egy érték eredetileg is hibás volt, azt csak úgy tudjuk kiszűrni, ha elvégzünk egy hihetőségvizsgálatot. Ehhez ebben a példában azt a szakterületi tudást kell alkalmaznunk, hogy a CPU különböző állapotokban eltöltött időhányadainak összege kiadja a 100%-ot így azokat a sorokat kiszűrve, ahol ez nem teljesül, javíthatunk az adatokon. Ez sem nyújt tökéletes védettséget az adathibák ellen, de a durva hibák kiküszöbölhetőek segítségével.

- Ha volt egy felesleges sor a bemeneten, azt csak úgy tudjuk kiszűrni, ha annak valamilyen attribútuma hibás. Ebben az esetben azt az attribútumot vizsgálva kiszűrhetjük ezeket a hibás értékeket. A hiányzó sorok már nehezebben kezelhetőek. Ebben az esetben, mint szakterületi információként tudhatjuk, hogy egy idősor elemzéséről van szó, így az előző és a következő érték átlagát véve jobb eredményt kaphatunk.

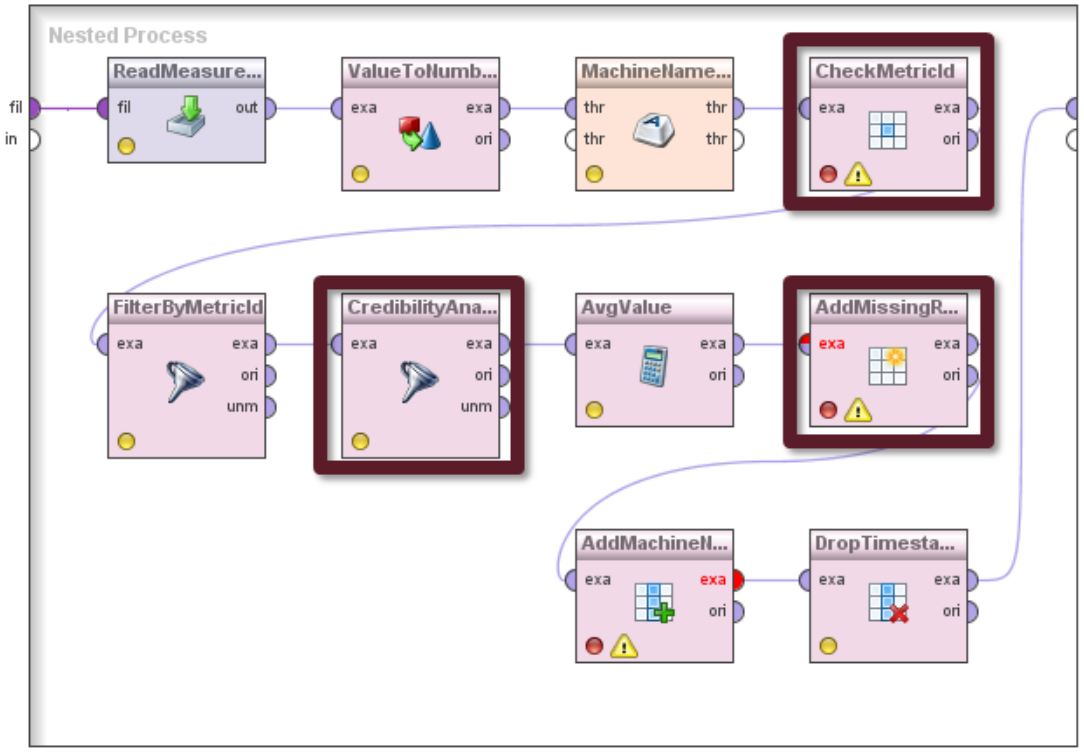
A folyamat egy lehetséges javításokkal kiegészített változata látható az 5.6. ábrán. Az analízis eredménye megmutatja, a folyamat minden pontjára, hogy ott milyen adathibák lépnek fel. Ebből látszik, hogy melyik bemeneti hiba hol okoz további hibát a folyamatban, így hibakezelő lépéseket ezek a pontok elé érdemes tenni. Egyedi esetben előfordulhat ugyan, hogy a bemeneti hiba által okozott későbbi hibát egyszerűbb kezelni, de ez általánosan nem mondható el. Általánosságban érdemes a bemeneti hibát még azelőtt kezelni, mielőtt az egyéb hibát okozna.

Mint látható, nem minden esetben lehet tökéletesen megtisztítani a bemeneti adatokat, hiszen a subtle jellegű adathibák egy része nem észrevehető, mivel értékük nem tér el jelentősen a valós, hihető értéktől, de ezekkel az ellenőrzésekkel jelentősen javítható az adatminőség.

Az analízis eredményeit egy egyszerű eredménymodellben írtuk fel, hogy programozottan könnyen felhasználható legyen. Így később akár automatikusan is beilleszthetőek lesznek a hibajavító lépések, hiszen az aktív hibamódokból és a 4.2.3. fejezetben leírt hibajavító lehetőségekből felírt ontológiából már egyértelműen következik, hogy hová, milyen lépéseket kell beszúrni.

5.6. Hibajavító és adattisztító operátorok beszúrásának költsége

Felmerült a kérdés bennünk, hogy mi a következménye a plusz adathiba-javító operátorok beszúrásának. A 4.9. ábrán pár adathiba-javító operátor költségét olvashatjuk, amennyiben az operátor bemenetén megjelenő adatszerzet n sorból és m oszlopból áll. Kíváncsinak voltunk, hogy egy nagyobb elemi költségű lépés beiktatása, mint



5.6. ábra. Adathibák kiküszöbölése RapidMiner operátorokkal

például a duplikátummentesítő operátor, mennyivel növeli meg a RapidMiner folyamat futásának hosszát. Ezért méréseket végeztünk a korábban bemutatott esettanulmányon (*CpuUsageAverage* - 2. fejezet), valamint a függelékben leírásra kerülő komplex folyamaton (*CompletenessCheck* - F.1. fejezet).

Az 5.7. táblázatban olvashatjuk a mérések eredményeit. Látható, hogy az adatelemző folyamat elvégzésének időtartama elsősorban az egymásba ágyazott ciklusok számától függ, hiszen ezek futtatása nagyon költséges művelet. A mérések alapján megállapítottuk, hogy egy-egy hibajavító és adattisztító operátor beszúrása nem befolyásolja jelentősen a folyamat végrehajtásának idejét.

A folyamataink során felhasznált mérési adatok nem BigData jellegűek. Az elosztottan nagy mennyiségű adatot tároló Hadoop NoSQL adatbázishoz a RapidMiner Radoop [29] kiterjesztése biztosít hozzáférést. BigData környezetben az elemi lépések komplexitásának sokkal nagyobb szerepe van. A költségek felírásával módszerünket részben felkészítettük arra, hogy BigData adatszerkezettel dolgozó folyamatokban költséghatékony megoldásokat tudjunk ajánlani az adathibák kiküszöbölésére.

	CpuUsageAverage	CompletenessCheck
Operátorok száma	9	49
Ágymásba ágyazott ciklusok száma	0	4
Futási idő, duplikátummentesítő lépés nélkül Bemenet: 19232 megfigyelés	<1s	11s
Futási idő, duplikátummentesítő lépéssel Bemenet: 19232 megfigyelés	<1s	12s
Futási idő, duplikátummentesítő lépés nélkül Bemenet: 2x19232 megfigyelés	<1s	25s
Futási idő, duplikátummentesítő lépéssel Bemenet: 2x19232 megfigyelés	<1s	25s
Futási idő, duplikátummentesítő lépés nélkül Bemenet: 100000 megfigyelés	<1s	92s
Futási idő, duplikátummentesítő lépéssel Bemenet: 100000 megfigyelés	<1s	92s

5.7. ábra. Mérési eredmények

6. fejezet

Továbbfejlesztési lehetőségek

Az eszköz elkészítése közben sok továbbfejlesztési lehetőség merült fel. Ezek közül a legfontosabbakat röviden bemutatjuk ebben a fejezetben. A további kutatási és fejlesztési irányokat megkülönbözettük aszerint, hogy az eszköz technológiáját fejlesztenék, a gyakorlati felhasználási lehetőségeit könnyítenék vagy a háttérben használt módszerhez adnának elvi továbbfejlesztést.

6.1. Technológiai továbbfejlesztések

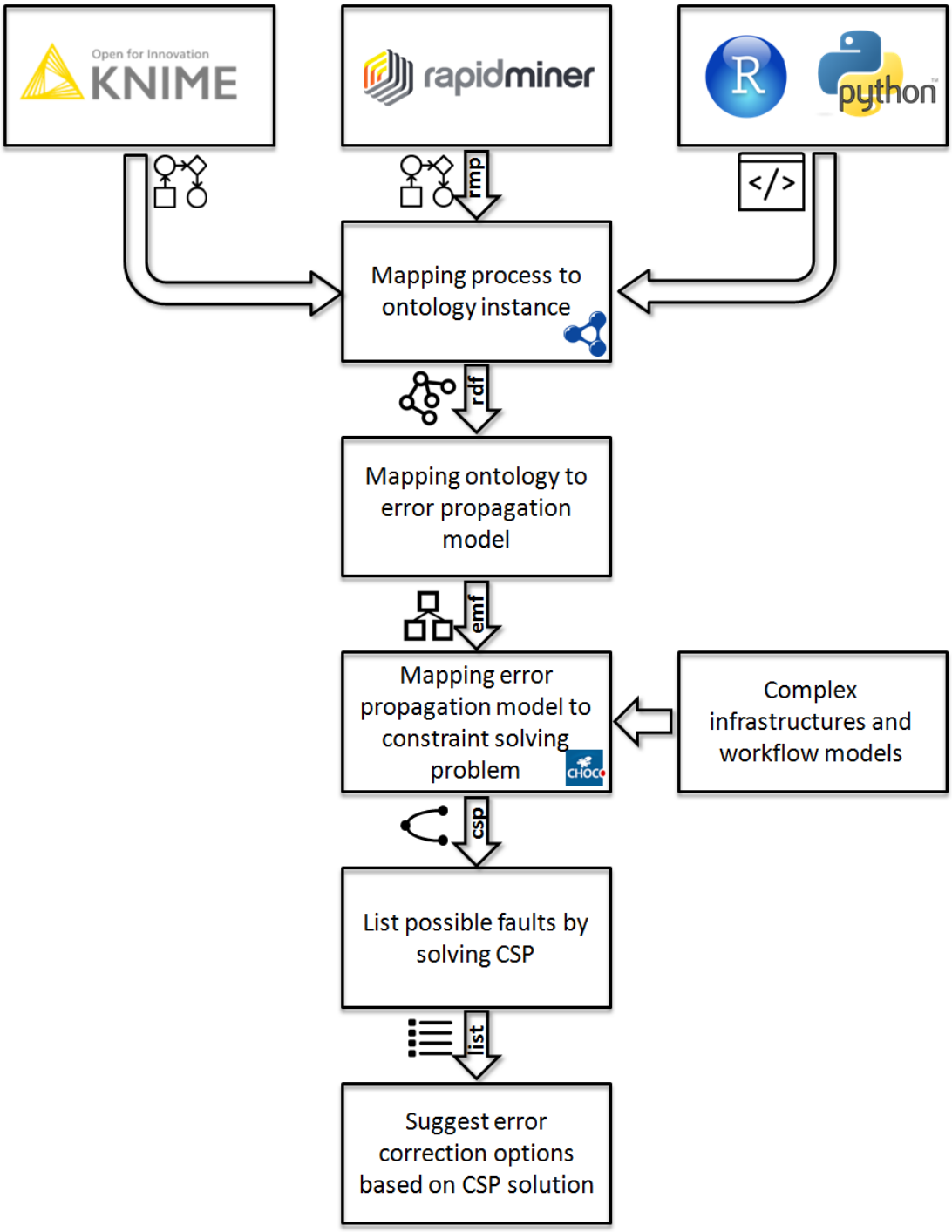
További adatelemző eszköz integrációja Az elkészült eszközt úgy készítettük el, hogy a RapidMiner könnyen lecserélhető legyen más folyamat alapú eszközökre, de akár más nem folyamat alapú eszközre is. Ezt az általános leíró ontológiák és az általános hibaterjedési modell teszi lehetővé. Ezt a kiterjeszthetőséget szemlélteti a 6.1. ábra.

Hibajavítási lehetőségek automatikus beszúrása a folyamatba Az adatelemzés során fellépő hibák kiküszöbölésére javaslatot tudunk adni egy a hibajavítási lehetőségeket összegyűjtő ontológia alapján. Biztosítottuk operátorok a folyamatok tetszőleges pontjára történő beszúrását. A továbbiakban a kettő automatizált együttműködését szeretnénk megvalósítani, azaz a hibajavító lépések automatikus felvételét a folyamatba.

6.2. Használhatósági fejlesztések

RapidMiner integráció Jelenleg a RapidMinerből exportált folyamatokat tudjuk vizsgálni, azonban a RapidMiner Studio biztosít lehetőséget beépülő modulok készítésére, így az általunk készített eszközt integrálni lehetne fejlesztőkörnyezetbe. Így az adatelemző folyamat fejlesztőkörnyezetén belül elvégezhető lenne az analízis és a javítási lehetőségek azonnali beszúrása.

Vizuális visszajelzés a RapidMiner eszközben Érdemes volna egy egyszerű intuitív módon megjeleníteni az analízis eredményeit. Erre legalkalmasabb a RapidMiner saját eszköze volna, ahol megjelölhetnénk a hibát okozó lépéseket. A RapidMiner API nyújt is erre lehetőséget, egy RapidMiner Studio beépülő modullal megoldható a vizuális visszajelzés.



6.1. ábra. Bemutatott módszer további felhasználási lehetőségei

A hibaterjedés analízishez könyvtárak definiálása A hibaterjedés analízis eszköz önmagában is jól használható, azonban érdemes volna a gyakori komponenstípusok viselkedését leírni egy könyvtárban. Így ez után egy új rendszer modellezése a könyvtár használatával sokkal egyszerűbb lehet, hiszen csak példányosítani kell a könyvtárban szereplő komponenseket.

Hibaterjedés analízis hierarchikus futtatásának támogatása Egy adatelemző folyamat alfolyamatait vizsgálva felállíthatunk egy kényszerhalmazt, amely a fő folyamat analízise során helyettesíteni tudja az alfolyamatot. Így a további analízis során már nem kell az alfolyamat lépéseit figyelembe venni, ezzel jelentősen felgyorsítva az analízist. Ennek kézi elvégzésére már most nyújt támogatást az eszköz, azonban ezt érdemes volna automatizálni is.

6.3. A módszer fejlesztési lehetőségei

Mérési környezet szakterületi információinak felhasználása A vizsgált rendszer szakterületi információit felhasználva tovább lehet pontosítani az analízis eredményeit. Kifinomultabb hibaterjedési szabályok, kényszerek vehetőek fel (például meg tudjuk különböztetni az érvényes és érvénytelen adatokat, összefüggéseket írhatunk fel az attribútumok között), illetve akár automatikusan felparaméterezhetőek a hibajavításként beiktatott lépések (Például egy százalékokat tartalmazó attribútum értéke csak 0 és 100 között lehet).

Adattulajdonság transzformációs ontológia vizsgálata Az adattulajdonság transzformációs ontológia írja le a hibaterjedési szabályokat, így az ontológia helyes felírása határozza meg az analízis sikerességét. Ezt az ontológiát ezért érdemes volna ellenőrizni. Vizsgálhatnánk teljesség szempontjából, hogy minden kitüntetett esetre írtunk-e fel szabályt, illetve helyesség szempontjából nincsenek-e ellentmondások a szabályok között (nincs olyan eset, mikor két hibaterjedési szabály alkalmazása közül nem tudunk dönteni).

Adatminőségi jellemzők használata A későbbiekben az általunk kidolgozott módszer kiegészíthető az IBM Watson Anlalytics eszköznél megismert, adatminőséget meghatározó jellemzők figyelembevételével. Amikor felírjuk a hibaterjedési szabályokat, akkor figyelembe vesszük, hogy mik lesznek pl. az influential category típusok, és elvárjuk azt, hogy legyen hozzájuk hibaterjedési szabály.

Adattípustól függő viselkedés felírása Jelenleg az egyes lépéstípusokra csak általánosságban írhatóak fel a hibaterjedési szabályok, azonban egyes operátorok viselkedése függhet az adat típusától. Például egy filter lépés estén, ha kategóriaváltozót vizsgál, akkor már ellenőrizetlen bemeneten egy elgépelés is sorhibához vezethet, azonban ha egy számszerű értékeket tartalmazó oszlopot vizsgál, akkor csak küszöbérték feletti értékhiba esetén keletkezik valamilyen sorhiba. Ezért érdemes lehet adattípus-függő szabályok felírását is lehetővé tenni.

Hibajavító lépések költségének figyelembevétele Méréseink alapján egy-egy hibajavító lépés nem módosít sokat egy bonyolultabb adatelemző folyamat futási idején. Azonban BigData környezetben az extrém mennyiségű adat miatt nagy

különbség lehet két hibajavító módszer futási ideje között. Ezért érdemes lehet az egyes módszerek költségét megbecsülni, és ez alapján javaslatot tenni a hibajavításra.

Folyamat felépítésének ellenőrzése lépéstípus kategóriák alapján Munkánk során definiáltunk lépéstípus kategóriákat. Jó volna ezeket használni az folyamatok felépítésének ellenőrzésére. Például ennek segítségével megállapíthatjuk, hogy egy folyamatban van-e hibakezelésért felelős ág.

7. fejezet

Összefoglalás

A dolgozat során elkészítettünk egy módszert és eszközt, amellyel elvégezhető adatelemző folyamatok hibaterjedés analízise. Ahhoz, hogy általánosan elvégezhető legyen az analízis, a szakirodalom alapján létrehoztunk ontológiákat, amelyekbe összegyűjtöttük az adatelemző folyamatok általános jellemzőit. Ezek az ontológiák önmagukban is használhatóak. Módszerünket tanszéki kutatási projekthez kapcsolódó mintapéldán kipróbáltuk. Az elkészített rendszer közvetlenül integrálódik egy elterjedt adatfeldolgozó/elemző eszköz kimeneti modelljéhez.

Megvizsgáltuk az adatelemző folyamatok általános felépítését Adatelemző eszközök vizsgálata alapján definiáltunk egy általános folyamatleíró ontológiát, amellyel jól reprezentálhatóak az adatelemző folyamatok.

Megalkottunk egy adathibákat leíró taxonómiát Modellünkben ötvöztük a hibaterjedés analízis során megjelenő kvalitatív hibakategóriákat az általános adathibákkal. Az adathibák kategorizálását szakterülettől függetlenül végeztük el. Ennek használatával felírhatóvá váltak az adatelemző folyamatok bemeneteinek és eredményeinek hibái.

Az adatelemző lépések viselkedését felírtuk, összegyűjtöttük Definiáltunk egy ontológiát, amely segítségével felírható, hogy hogyan reagálnak az egyes lépések a hibás bemenetekre, azaz a bemenetükön megjelenő adathibát milyen kimeneti adathibává transzformálják vagy javítják.

Leképezést adtunk RapidMiner folyamatokról ontológiára A RapidMiner adatelemző eszköz automatikus folyamataihoz leképezést nyújtottunk az általunk definiált folyamatleíró ontológiára.

Definiáltunk egy nyelvet hibaterjedési problémák felírására Elkészítettünk egy nyelvet, amellyel felírhatóak hibaterjedési feladatok. A nyelvet, és így az analízis bemeneti modelljét úgy alkottuk meg, hogy minél általánosabban legyen használható, ne csak adatelemző folyamatokra.

Elkészítettük a hibaterjedés analízist megoldó eszközt Készítettünk egy eszközt, amellyel a felírt hibaterjedési problémák leképezhetőek egy kényszerkielégítési problémára, amelyet a Choco általános megoldó segítségével megoldunk, ezzel azonosítottuk a lehetséges hibás pontokat.

Összegyűjtöttük a lehetséges javítási megoldásokat Az analízis során megtalált lehetséges hibák egy része javítható a folyamatba illesztett javító lépésekkel. Egy ontológiában összegyűjtöttük az egyes adathibák lehetséges javításait.

A bemutatott módszer általánosan alkalmazható adatelemző folyamatok analízisére. Az elkészült eszközöket, modelleket úgy alkottuk meg, hogy a rendszer könnyen alkalmazható legyen más eszközökön is.

Irodalomjegyzék

[1] Nathanael L Ackerman – Cameron E Freer – Daniel M Roy: On the computability of conditional probability. *arXiv preprint arXiv:1005.3014*, 2010.

[2] Jose Ignacio Aizpurua – Enaut Muxika: Model-based design of dependable systems: Limitations and evolution of analysis and verification approaches. *International Journal on Advances in Security Volume 6, Number 1 & 2, 2013*, 2013.

[3] André Arnold – Gérald Point – Alain Griffault – Antoine Rauzy: The altarica formalism for describing concurrent systems. *Fundam. Inform.*, 40. évf. (1999) 2-3. sz., 109–124. p.

[4] Simona Bernardi – Francesco Flammini – Stefano Marrone – Nicola Mazzocca – José Merseguer – Roberto Nardone – Valeria Vittorini: Enabling the usage of uml in the verification of railway systems: the dam-rail approach. *Reliability Engineering & System Safety*, 120. évf. (2013), 112–126. p.

[5] Simona Bernardi – José Merseguer – Dorina C Petriu: Adding dependability analysis capabilities to the marte profile. In *Model Driven Engineering Languages and Systems*. 2008, Springer, 736–750. p.

[6] Simona Bernardi – José Merseguer – Dorina C Petriu: A dependability profile within marte. *Software & Systems Modeling*, 10. évf. (2011) 3. sz., 313–336. p.

[7] Andera Bondavalli – Luca Simoncini: Failure classification with respect to detection. In *Distributed Computing Systems, 1990. Proceedings., Second IEEE Workshop on Future Trends of* (konferenciaanyag). 1990, IEEE, 47–53. p.

[8] György Csertán – András Pataricza – Péter Harang – Orsolya Dobán – Gabor Biros – András Dancsecz – Ferenc Friedler: Bpm based robust e-business application development. In *Dependable Computing EDCC-4*. 2002, Springer, 32–43. p.

[9] Nan Feng – Harry Jiannan Wang – Minqiang Li: A security risk analysis model for information systems: causal relationships of risk factors and vulnerability propagation analysis. *Information sciences*, 256. évf. (2014), 57–73. p.

[10] Barbara Gallina – Muhammad Atif Javed – Faiz UL Muram – Sasikumar Punnekkat: A model-driven dependability analysis method for component-based architectures. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on* (konferenciaanyag). 2012, IEEE, 233–240. p.

[11] Barbara Gallina – Sasikumar Punnekkat: Fi4fa: A formalism for incompletion, inconsistency, interference and impermanence failures' analysis. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on* (konferenciaanyag). 2011, IEEE, 493–500. p.

[12] Lars Grunske – Jun Han: A comparative study into architecture-based safety evaluation methodologies using aadl's error annex and failure propagation models. In *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE* (konferenciaanyag). 2008, IEEE, 283–292. p.

[13] Bin Gu – Yunwei Dong – Xiaomin Wei: A qualitative safety analysis method for aadl model. In *Software Security and Reliability-Companion (SERE-C), 2014 IEEE Eighth International Conference on* (konferenciaanyag). 2014, IEEE, 213–217. p.

[14] PeterJ. Huber: *Robust Statistics*. 2011, Springer Berlin Heidelberg, 1248–1251. p. ISBN 978-3-642-04897-5.

[15] IBMWatsonAnalytics. `http://www.ibm.com/analytics/watson-analytics/`, 2015. Oct.

[16] ISO/IEC 25012 x ISO/IEC 25012 Software-Engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Data quality model.

[17] AJ Izenman: *Modern multivariate statistical techniques*. 1. köt. ISBN 978-0-387-78188-4.

[18] Knime Konstanz information miner. `https://www.knime.org/`, 2015. Oct.

[19] Fabien Kuntz – Stéphanie Gaudan – Christian Sannino – Éric Laurent – Alain Griffault – Gérald Point: Model-based diagnosis for avionics systems using minimal cuts. In *DX 2011* (konferenciaanyag). 2011, 138–145. p.

[20] Xiaoxun Li – Shaojun Li: Graphical modeling of system failure behavior and its translating into altarica. *Procedia Engineering*, 80. évf. (2014), 581–591. p.

[21] Carmen Moraga – Mª Moraga – Coral Calero – Angélica Caro: Square-aligned data quality model for web portals. In *Quality Software, 2009. QSIC'09. 9th International Conference on* (konferenciaanyag). 2009, IEEE, 117–122. p.

[22] Carmen Moraga – Mª Ángeles Moraga – Angélica Caro – Coral Calero: Spdqm: Square-aligned portal data quality model. In *Ninth International Conference on Quality Software, QSIC* (konferenciaanyag). 2009, 24–25. p.

[23] Andrey Morozov – Klaus Janschek: Dual graph error propagation model for mechatronic system analysis. In *18th IFAC World Congress, Milano, Italy* (konferenciaanyag). 2011.

[24] OpenRefine. `http://openrefine.org/`, 2015. Oct.

[25] Yiannis Papadopoulos – John A McDermid: Hierarchically performed hazard origin and propagation studies. In *Computer Safety, Reliability and Security*. 1999, Springer, 139–152. p.

[26] András Pataricza: Model-based dependability analysis. *DSc Thesis. Hungarian Academy of Sciences*, 2006.

[27] Thanh-Trung Pham – Xavier Défago: Reliability prediction for component-based systems: Incorporating error propagation analysis and different execution models. In *Quality Software (QSIC), 2012 12th International Conference on* (konferenciaanyag). 2012, IEEE, 106–115. p.

[28] Charles Prud'homme – Jean-Guillaume Fages – Xavier Lorca: *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014. URL `http://www.choco-solver.org`.

[29] Radoop. `https://rapidminer.com/products/radoop/`, 2015. Oct.

[30] Irfan Rafique – Philip Lew – Maissom Qanber Abbasi – Zhang Li: Information quality evaluation framework: Extending iso 25012 data quality model. *World Academy of Science, Engineering and Technology*, 65. évf. (2012), 523–528. p.

[31] Erhard Rahm – Hong Hai Do: Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23. évf. (2000) 4. sz., 3–13. p.

[32] RapidMiner. `https://rapidminer.com/`, 2015. Oct.

[33] OASIS Standard: Web services business process execution language version 2.0. 2007. URL `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[34] OMG Standard: Business process model and notation (bpmn) version 2.0. 2011. URL `http://www.omg.org/spec/BPMN/2.0/`.

[35] Gábor Urbanics – László Gönczy – Balázs Urbán – János Hartwig – Imre Kocsis: Combined error propagation analysis and runtime event detection in process-driven systems. In *Software Engineering for Resilient Systems*. 2014, Springer, 169–183. p.

[36] Jan Van den Broeck – Solveig Argeseanu Cunningham – Roger Eeckels – Kobus Herbst: Data cleaning: detecting, diagnosing, and editing data abnormalities. *PLoS medicine*, 2. évf. (2005) 10. sz., 966. p.

[37] Malcolm Wallace: Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science*, 141. évf. (2005) 3. sz., 53–71. p.

[38] Hadley Wickham: Tidy data. *The Journal of Statistical Software*, 59. évf. (2014). URL `http://www.jstatsoft.org/v59/i10/`.

[39] Wings. `http://www.wings-workflows.org/`, 2015. Oct.

[40] Ian Wolforth – Martin Walker – Yiannis Papadopoulos – Lars Grunske: Capture and reuse of composable failure patterns. *International Journal of Critical Computer-Based Systems*, 1. évf. (2010) 1-3. sz., 128–147. p.

[41] Xtext 2.8.4. `http://www.eclipse.org/Xtext`, 2015. Oct.

Függelék

F.1. Mérés teljességellenőrző folyamat bemutatása és hibaterjedési lehetőségeinek leírása

Ez a RapidMiner folyamat azt ellenőrzi, hogy minden szükséges metrikát mintavételeztünk-e a mérés során. Azaz minden példányon, minden mintavételezési időpillanatban, az összes szükséges metrikát mértük-e. A vizsgálat kimenetén azok az időbélyeg-erőforrás-metrikanév hármasok jelennek meg, amelyeket a mérési eredményeket leíró adatszerkezet nem tartalmaz, pedig rögzíteni kellett volna.

F.1.1. Mérési környezet

Megegyezik a 2.1. fejezetben bemutatottal.

F.1.2. A teljességellenőrző folyamat bemenete

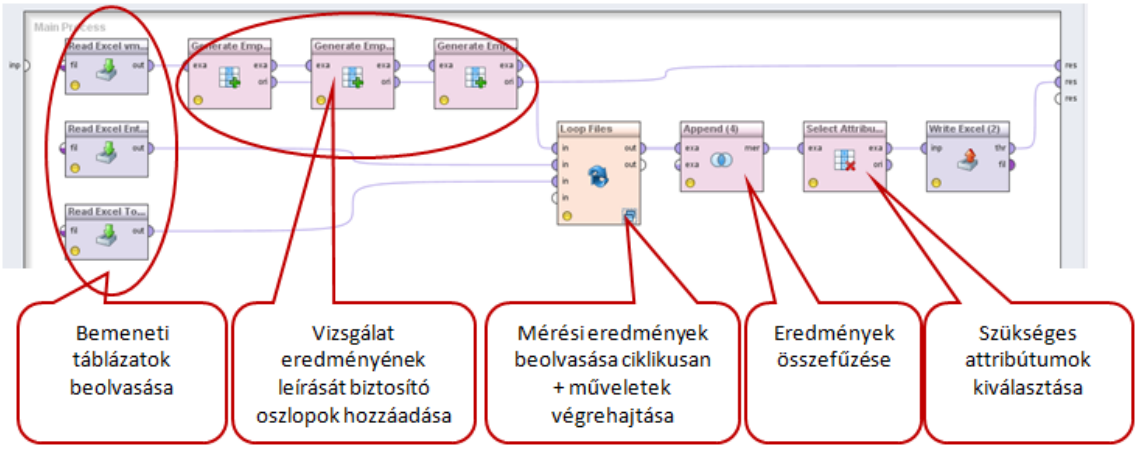
- *vmware_metrics.xlsx*: Leírja, hogy egy metrikát milyen típusú erőforrásnál kell mintavételezni. Ezt a táblázat Name (metrika neve), Guest/Host (guest vagy host esetében szükséges mintavételezni) és FeatureOfIntrest (milyen típusú erőforráspéldány esetén kell mintavételezni) attribútumai határozzák meg.

- *EntityMapping_0150.xlsx*: Leírja, hogy az egyes gépnevek fizikai, vagy virtuális gépeket takarnak.

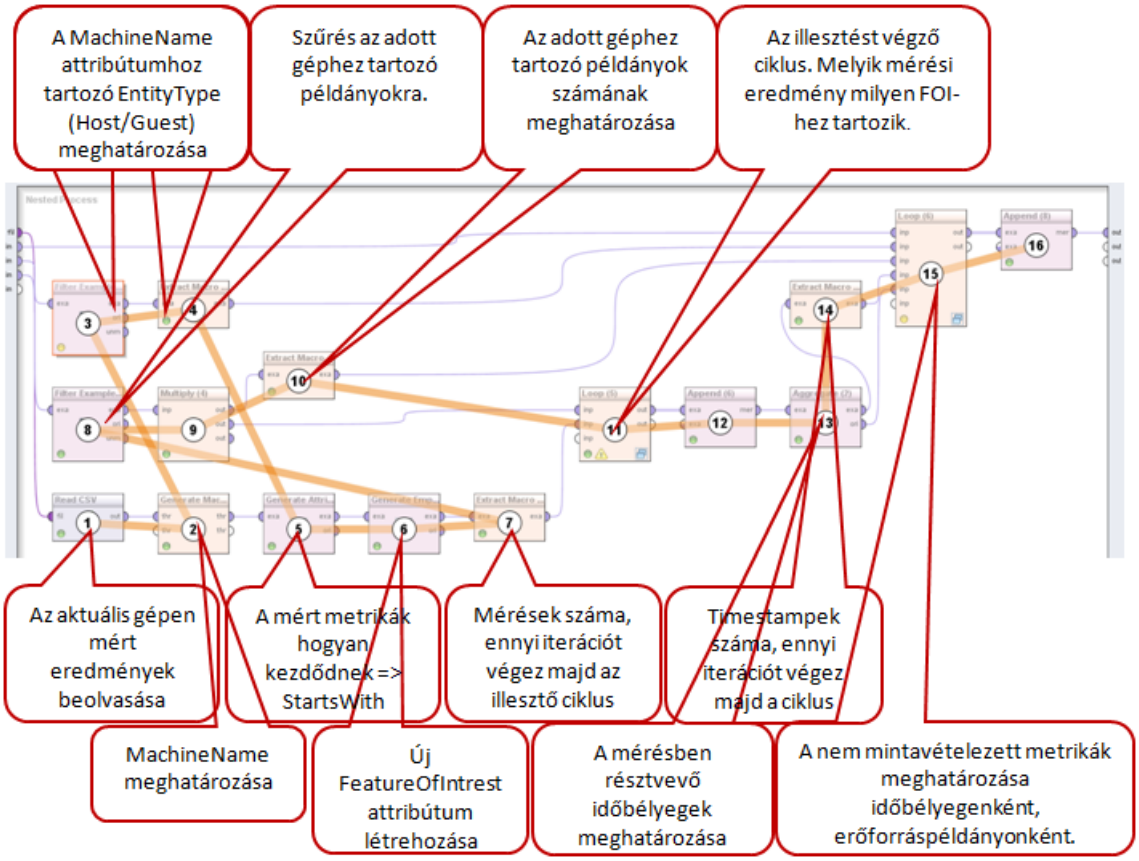
- *TopologyMapping_0150.xlsx*: Leírja, hogy az egyes gépeken milyen erőforráspéldányok találhatóak, és ezeknek mik a FOI attribútumai. Ezen kívül megtalálható, hogy az adott FOI-hoz tartozó metrikák hogyan kezdődhetnek. (Erre azért van szükség, mert a mérések eredményeit leíró állományokból (Timestamp,Value,MetricId,Instance) nem derül ki, hogy az egyes példányokhoz milyen FOI tartozik, azaz, hogy valójában mit mértünk ott. Ezért valahonnan ki kell derüljön, hogy az adott metrika milyen FOI-hoz tartozik.)

- *ESXi_iteration*: Ebben a mappában találhatók az egyes gépekről a mérési eredmények.

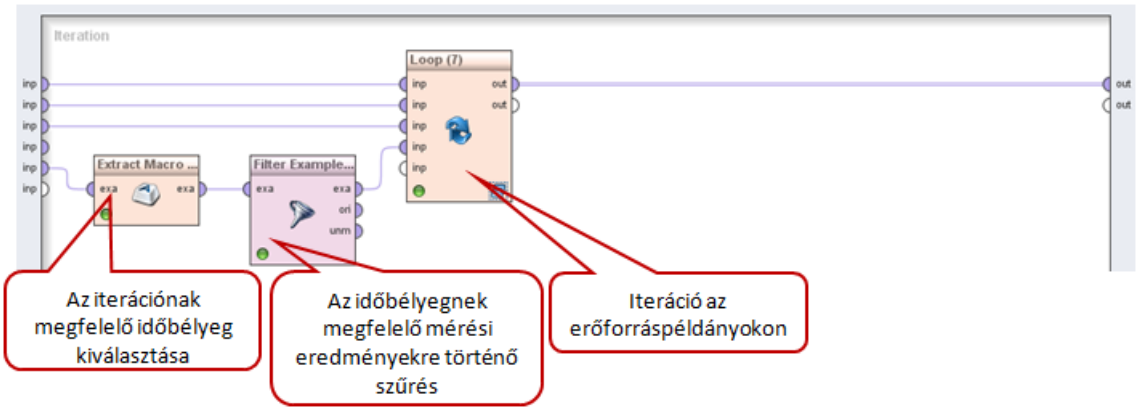
F.1.3. A teljességellenőrző folyamat felépítése



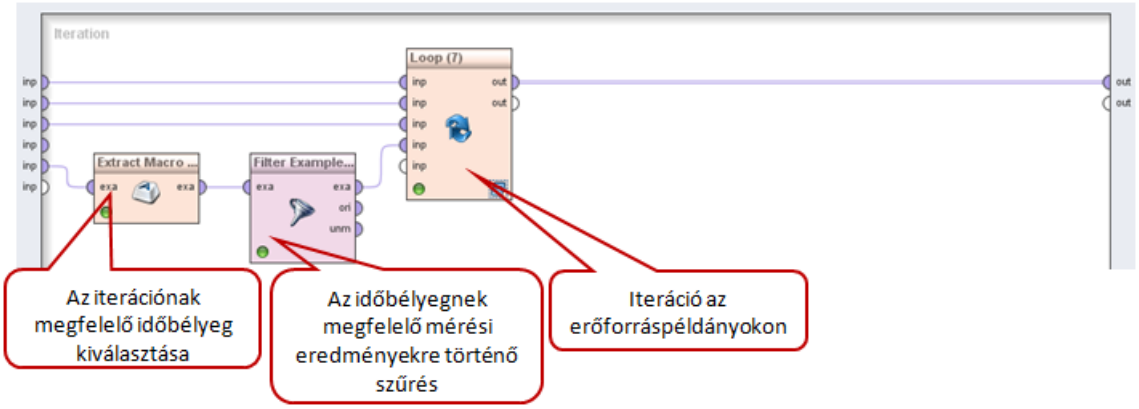
F.1.1. ábra. Fő folyamat felépítése



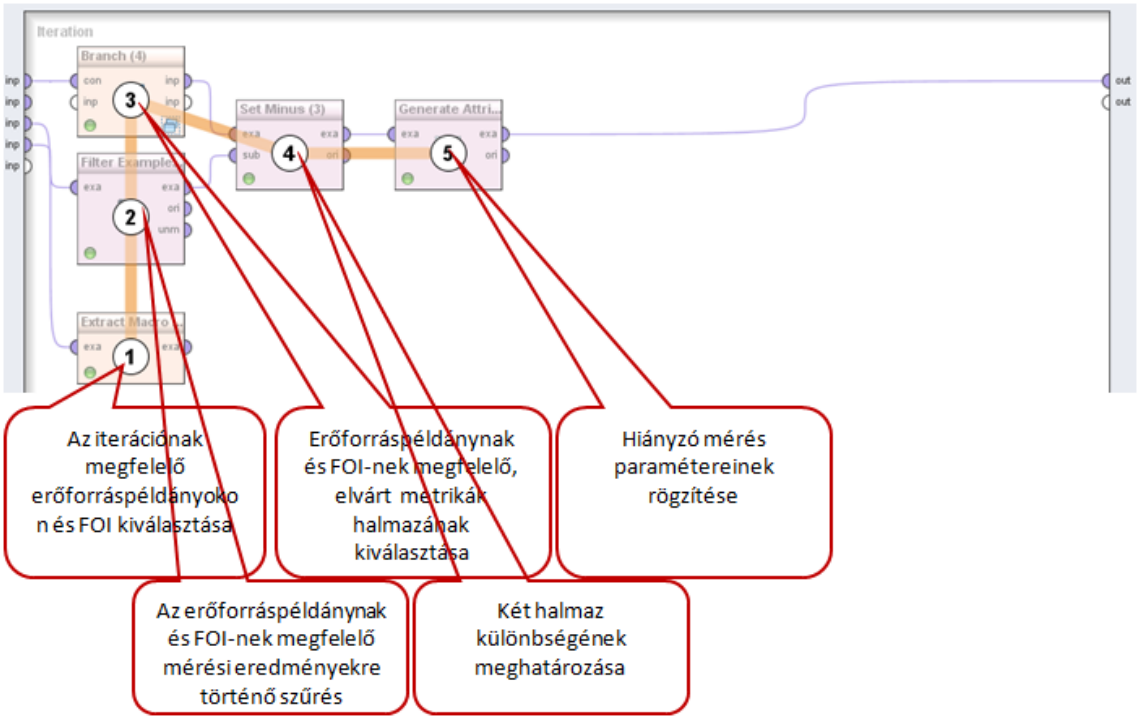
F.1.2. ábra. Mérési eredményeken iteráló ciklus



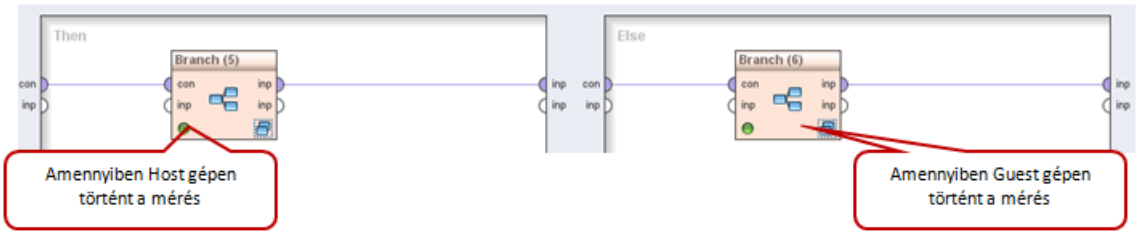
F.1.3. ábra. Illesztést elvégző ciklus



F.1.4. ábra. Időbélyegeken iteráló ciklus



F.1.5. ábra. Időbélyegeken iteráló ciklus



F.1.6. ábra. Branch a erőforrástípusoknak megfelelően

F.1.7. ábra. Albranch a erőforrástípusoknak megfelelően

F.1.4. Hibaterjedés a teljesség-ellenőrző folyamatban

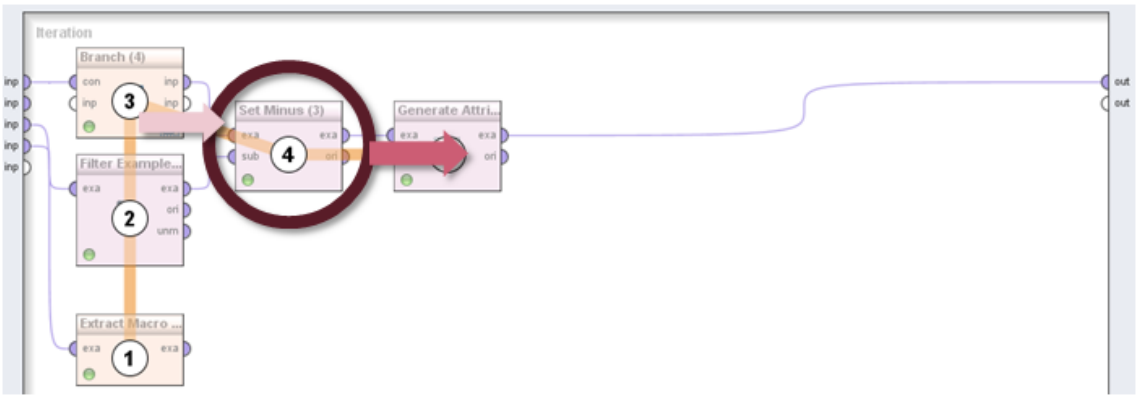
A következő fejezetekben az egyes adathibatípusok terjedését figyelhetjük meg.

Duplikátumot tartalmaz

Jelen esetben elég, ha két sor a következő attribútumokban megegyezik: Name, Host, Guest, Feature of interest (FOI) (F.1.8. ábra), mivel a vizsgálat során ezeket az attribútumokat használja a folyamat a szűrőfeltételekben. Nyilván így a táblázat végig hibás, de a hiba ott jelentkezik először, amikor a táblázattal kezd dolgozni a folyamat. Ez akkor történik meg, amikor az adott példánynak megfelelő FOI-ra szűrjük le a táblázatot (F.1.9. ábra), azaz megkeressük, hogy mit kellett volna mintavételezni. Hiszen itt a duplikált metrika kétszer fog szerepelni, így amikor kivonjuk ebből a halmazból a valójában mintavételezett metrikákat, akkor a különbségben szerepelni fog a duplikált metrika kétszer, de csak akkor, ha tényleg nem volt benne. Ha mértük, akkor nem fog egyszer sem, mert ID attribútum (metrika neve) alapján megy a kivonás, és az is duplikálva van. Szóval valójában a kivonás eredménye gyakorol hibát a kimenetre(F.1.10. ábra).

Name	Property	Description	Unit	StatsType	RollupType	Entity	Host	Guest	ResourcePool	Storage	Range	Feature of
cpu.idle.summation	Utilization	Total time that the CPU	millisecond	delta	summation	Guest	None	Both	None	None	0 – 20000	VirtualCPU
cpu.ready.summation	Saturation	Percentage of time that	millisecond	delta	summation	Guest	None	Both	None	None	See Vmware	VirtualCPU
cpu.run.summation	Saturation	Time the virtual machine	millisecond	delta	summation	Guest	None	Both	None	None	???	VirtualCPU
cpu.swapwait.summation	Saturation	CPU time spent waiting f	millisecond	delta	summation	Host,	None	Both	None	None	See Vmware	VirtualCPU
cpu.system.summation	Utilization	Amount of time spent	millisecond	delta	summation	Guest	None	Both	None	None	See Vmware	VirtualCPU
cpu.usage.average	Utilization	CPU usage as a	percent	rate	average (minimum)	Host,	Both	Aggregated	None	None	See Vmware	PhysicalCPU,
cpu.usagemhz.average	Utilization	CPU usage, as	megaHertz	rate	average (minimum)	Host,	Aggregated	Both	None	None	See Vmware	PhysicalCPU,
cpu.used.summation	Utilization	Total CPU usage.	millisecond	delta	summation	Host,	Both	Both	None	None	See Vmware	PhysicalCPU,
cpu.wait.summation	Saturation	Total CPU time spent in	percent	delta	summation	Host,	None	Both	None	None	See Vmware	VirtualCPU
disk.commands.summation	Transactio	Number of SCSI	number	delta	summation	Host,	Instance	Instance	None	None	See Vmware	PhysicalDisk,
disk.commandsaborted.summ	Error	Number of SCSI	number	delta	summation	Host,	Instance	Instance	None	None	See Vmware	PhysicalDisk,
disk.devicelatency.average	Latency	Average amount of time,	millisecond	absolute	average	Host	Instance	None	None	None	See Vmware	PhysicalDisk
disk.kernellatency.average	Latency	Average amount of time,	millisecond	absolute	average	Host	Instance	None	None	None	See Vmware	PhysicalDisk
disk.numberread.summation	Transactio	Number of disk reads	number	delta	summation	Host,	Instance	Instance	None	None	See Vmware	PhysicalDisk,
disk.numberwrite.summation	Transactio	Number of disk writes	number	delta	summation	Host,	Instance	Instance	None	None	See Vmware	PhysicalDisk,
disk.queuelatency.average	Latency	Average amount of time	millisecond	absolute	average	Host	Instance	None	None	None	See Vmware	PhysicalDisk
disk.read.average	Data	Average number of	kiloBytesPerSe	rate	average	Host,	Both	Both	None	None	See Vmware	PhysicalDisk,
disk.totallatency.average	Latency	Average amount of time	millisecond	absolute	average	Host	Instance	None	None	None	See Vmware	PhysicalDisk
disk.totalreadlatency.average	Latency	Average amount of time	millisecond	absolute	average	Host	Instance	None	None	None	See Vmware	PhysicalDisk
disk.totalwritelatency.average	Latency	Average amount of time	millisecond	absolute	average	Host	Instance	None	None	None	See Vmware	PhysicalDisk
disk.totalwritelatency.average	Data	Average number of	kiloBytesPerSe	rate	summation	Guest	Instance	None	None	None	See Vmware	PhysicalDisk
disk.write.average	Data	Average number of	kiloBytesPerSe	rate	average	Host,	Both	Both	None	None	See Vmware	PhysicalDisk,

F.1.8. ábra. Duplikátumokat tartalmazó input



F.1.9. ábra. Duplikátum hibatranszformáció helye

ProblematicMachineName	EntityType	ProblematicInstanceName	ProblematicTimestamp	Name
beren.ftslab.local	Host	naa.5000cca00a2a07b8	1430418780	storageadapter.totalreadlatency.average
beren.ftslab.local	Host	naa.5000cca00a2a07b8	1430418780	storageadapter.totalwritelatency.average
beren.ftslab.local	Host	naa.5000cca00a2a07b8	1430418780	storagepath.totalreadlatency.average
beren.ftslab.local	Host	naa.5000cca00a2a07b8	1430418780	storagepath.totalwritelatency.average
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	disk.commands.summation
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	disk.commandsaborted.summation
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	disk.devicelatency.average
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	disk.kernellatency.average
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	disk.numberread.summation
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	disk.numberwrite.summation
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	disk.queuelatency.average
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	disk.read.average
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	disk.totallatency.average
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	disk.totalreadlatency.average
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	disk.totalwritelatency.average
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	disk.totalwritelatency.average
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	disk.write.average
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	storageadapter.totalreadlatency.average
beren.ftslab.local	Host	sas.5005076b08802b82-sas.5000cca0	1430418780	storageadapter.totalwritelatency.average

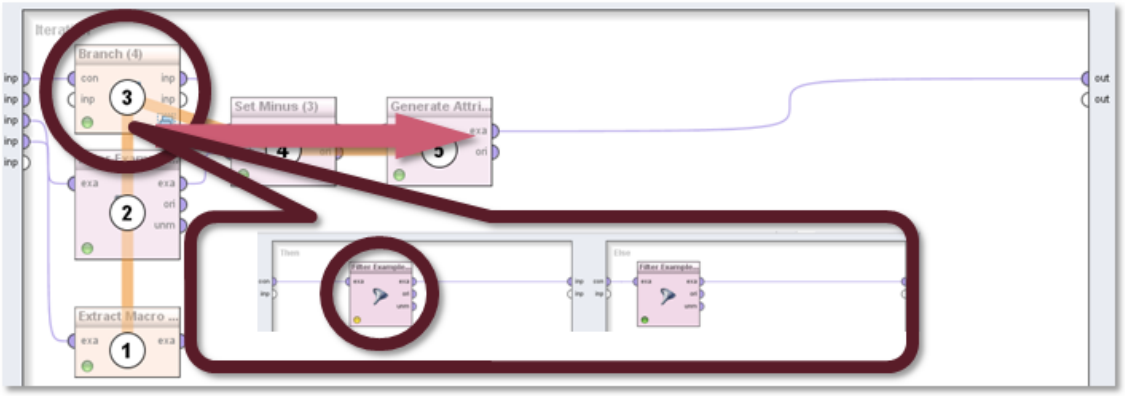
F.1.10. ábra. Duplikátum kimenetre gyakorolt hatása

Nem NA mentes adat

Hiányzó értékek szerepelnek a táblázat Guest/Host oszlopában(F.1.11. ábra), akkor az erre az attribútumra vonatkozó szűrő nem fogja megtalálni ezeket a sorokat, aminek az lesz a következménye, hogy kevesebb metrikáról várjuk el a mintavételezést (*Set Minus*). Így előfordulhat, hogy nem mintavételeztünk olyan metrikát, amit kellett volna, és ez nem is jelenik meg a folyamat kimenetén hibaként (F.1.13. ábra).

Name	Property	Description	Unit	StatsType	RollupType	Entity	Host	Guest	ResourcePool	Storage	Range	Feature of
cpu.idle.summation	Utilization	Total time that the CPU	millisecond	delta	summation	Guest	None	Both	None	None	0 – 20000	VirtualCPU
cpu.ready.summation	Saturation	Percentage of time that	millisecond	delta	summation	Guest	None	Both	None	None	See Vmware	VirtualCPU
cpu.run.summation	Saturation	Time the virtual machine	millisecond	delta	summation	Guest	None	Both	None	None	???	VirtualCPU
cpu.swapwait.summation	Saturation	CPU time spent waiting f	millisecond	delta	summation	Host,	None	Both	None	None	See Vmware	VirtualCPU
cpu.system.summation	Utilization	Amount of time spent	millisecond	delta	summation	Guest	None	Both	None	None	See Vmware	VirtualCPU
cpu.usage.average	Utilization	CPU usage as a	percent	rate	average (minimum)	Host,	Both	Aggregated	None	None	See Vmware	PhysicalCPU,
cpu.usagemhz.average	Utilization	CPU usage, as	megaHertz	rate	average (minimum)	Host,	Aggregated	Both	None	None	See Vmware	PhysicalCPU,
cpu.used.summation	Utilization	Total CPU usage.	millisecond	delta	summation	Host,	Both	Both	None	None	See Vmware	PhysicalCPU,
cpu.wait.summation	Saturation	Total CPU time spent in	percent	delta	summation	Host,	None	Both	None	None	See Vmware	VirtualCPU
disk.commands.summation	Transactio	Number of SCSI	number	delta	summation	Host,	Instance	Instance	None	None	See Vmware	PhysicalDisk,
disk.commandsaborted.summ	Error	Number of SCSI	number	delta	summation	Host,	Instance	Instance	None	None	See Vmware	PhysicalDisk,
disk.devicelatency.average	Latency	Average amount of time,	millisecond	absolute	average	Host	Instance	None	None	None	See Vmware	PhysicalDisk
disk.kernellatency.average	Latency	Average amount of time,	millisecond	absolute	average	Host	Instance	None	None	None	See Vmware	PhysicalDisk
disk.numberread.summation	Transactio	Number of disk reads	number	delta	summation	Host,	Instance	Instance	None	None	See Vmware	PhysicalDisk,
disk.numberwrite.summation	Transactio	Number of disk writes	number	delta	summation	Host,	Instance	Instance	None	None	See Vmware	PhysicalDisk,
disk.queuelatency.average	Latency	Average amount of time	millisecond	absolute	average	Host	Instance	None	None	None	See Vmware	PhysicalDisk
disk.read.average	Data	Average number of	kiloBytesPerSe	rate	average	Host,	Both	Both	None	None	See Vmware	PhysicalDisk,
disk.totallatency.average	Latency	Average amount of time	millisecond	absolute	average	Host	Instance	None	None	None	See Vmware	PhysicalDisk
disk.totalreadlatency.average	Latency	Average amount of time	millisecond	absolute	average	Host	Instance	None	None	None	See Vmware	PhysicalDisk
disk.totalwritelatency.average	Latency	Average amount of time	millisecond	absolute	average	Host		None	None	None	See Vmware	PhysicalDisk
disk.write.average	Data	Average number of	kiloBytesPerSe	rate	average	Host,	Both	Both	None	None	See Vmware	PhysicalDisk,

F.1.11. ábra. NA adathibát tartalmazó input



F.1.12. ábra. NA hibatranszformáció helye

EntityType	ProblematicInstanceName	ProblematicTimestamp	Name
Host	naa.5000cca00a2a07b8	1430418780	storageadapter.totalreadlatency.average
Host	naa.5000cca00a2a07b8	1430418780	storageadapter.totalwritelatency.average
Host	naa.5000cca00a2a07b8	1430418780	storagepath.totalreadlatency.average
Host	naa.5000cca00a2a07b8	1430418780	storagepath.totalwritelatency.average
Host	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.commands.summation
Host	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.commandsaborted.summation
Host	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.devicelatency.average
Host	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.kernellatency.average
Host	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.numberread.summation
Host	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.numberwrite.summation
Host	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.queuelatency.average
Host	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.read.average
Host	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.totallatency.average
Host	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.totalreadlatency.average
Host	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.write.average
Host	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	storageadapter.totalreadlatency.average
Host	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	storageadapter.totalwritelatency.average

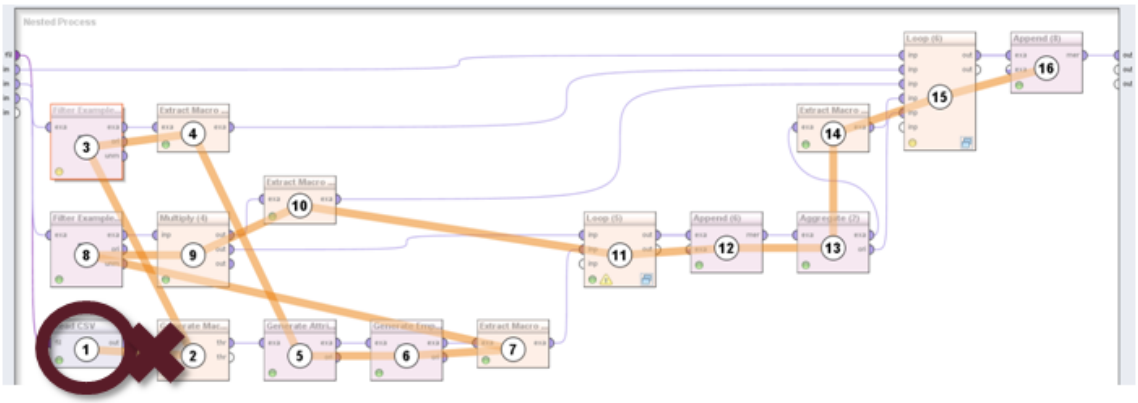
F.1.13. ábra. NA adathiba kimenetre gyakorolt hatása

Karakterkódolás hibás

Amennyiben a CSV állományok nem ugyanúgy vannak tagolva (az egyik „ , ”-vel, a másik„ ; ”-vel), akkor már a beolvasás sem fog sikerülni (F.1.14. ábra), mivel nem fog illeszkedni az operátor paramétereként megadott mintára. Ha a beolvasás közben nem jelentkezne hibaüzenet, akkor ezek az Example Setek nem tartalmaznának semmit, így a műveletek sem lennének elvégezhetőek rajtuk. Így végeredményben azt kapnánk, hogy mindent mintavételezünk, nem volt hiány.

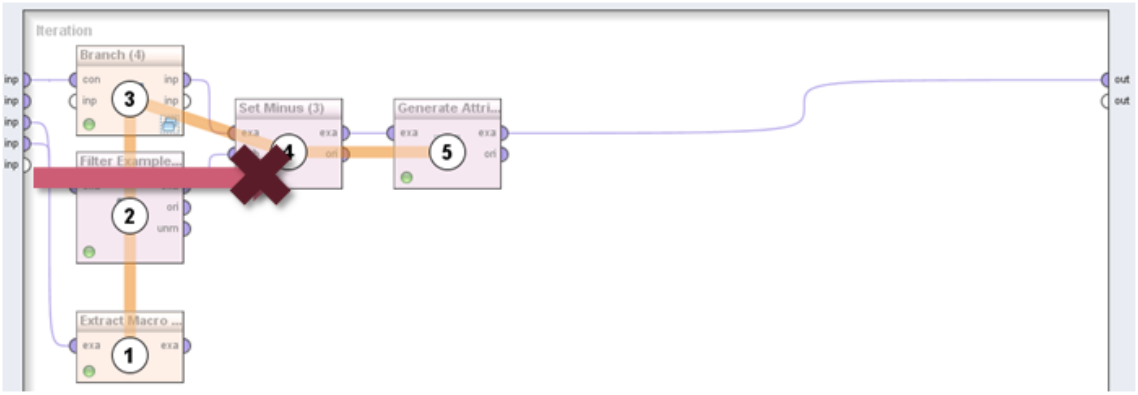
Egyediségre vonatkozó kényszer megsértése

Az egyes mintavételezések esetében az egyediséget a Timestamp, a MetricId és az Instance biztosítja. Hiszen egy időpillanatban, egy példányon, egy adott metrikát egyszer mintavételezhetünk. Amennyiben két sor ezekben az attribútumokban megegyezik, de a Value értékében különbözik, akkor az egyediség megsértéséről beszélhetünk. (Ha mind a négy attribútum megegyezne, akkor duplikátumról beszélnénk.) Amennyiben ezzel a problémával állunk szemben, az semmilyen hatással nem lesz a kimenetre, hiszen oda azokat az eseteket keressük, amikor egyáltalán nem mintavételeztünk. A rendszerben sincsen jelen végig a probléma, hiszen van egy egymásba ágyazott ciklus, ami az időbélyegeken, azon belül az példányokon megy végig, majd



F.1.14. ábra. Karakterkódolási hiba hatása

elvégez egy halmazműveletet. Ez a kivonás, amit mintavételezni kellett volna, abból vonja ki, amit mintavételeztünk, ha nagyobb a ténylegesen mért halmaz az elvártnál, azzal nem foglalkozik, tehát ebben a folyamatban itt nem terjed tovább a hiba.



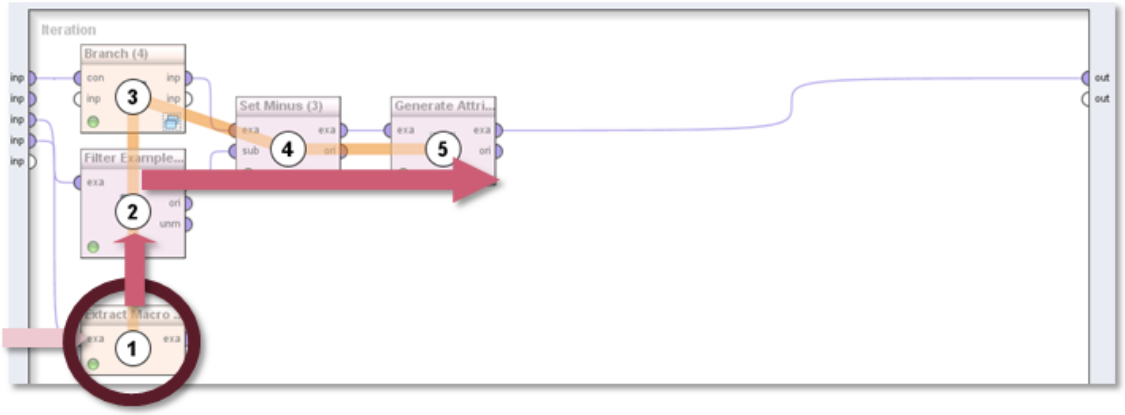
F.1.15. ábra. Egyediségre vonatkozó kényszer megsértése nem gyakorol hibát a kimenetre

Nem létező objektumra történő hivatkozás

A TopologyMapping táblázat tartalmaz olyan példányt az egyik gépen, ami valójában nem is létezik (F.1.16. ábra). Tehát ez egy nem létező objektumra történő referenciaként fog megjelenni ott, ahol az adott gépen lévő példányokon iterálunk végig, azaz amikor bekerül az értéke az InstanceName makróba (F.1.17. ábra). Ugyanis a filter nem talál egyetlen ilyen példány nevű mérést sem a mérési eredmények között, tehát az összes olyan metrika meg fog jelenni hiányzóként, amit egy olyan FOI esetén mérnünk kellene, amivel a nem létező példány rendelkezik. Így rengeteg hiányzó mérés kerül a kimenetre, holott az adott példány nem is létezik (F.1.18. ábra).

Machine	ResourceName	ResourceType_MetricStartsWith	FeatureOfIntrest
beren.ftslab.local		cpu	PhysicalCPU
beren.ftslab.local	0	cpu	PhysicalCPU
beren.ftslab.local	1	cpu	PhysicalCPU
beren.ftslab.local	2	cpu	PhysicalCPU
beren.ftslab.local	3	cpu	PhysicalCPU
beren.ftslab.local	NOTEXISTINGCPU	disk;storagepath;storageadapter	PhysicalDisk
beren.ftslab.local		disk;storagepath;storageadapter	PhysicalDisk
beren.ftslab.local	naa.5000cca00a2a07b8	disk;storagepath;storageadapter	PhysicalDisk
beren.ftslab.local	sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	disk;storagepath;storageadapter	PhysicalDisk
beren.ftslab.local	vmhba0	disk;storagepath;storageadapter	PhysicalDisk
beren.ftslab.local		net	PhysicalInterface
beren.ftslab.local	vmnic0	net	PhysicalInterface
beren.ftslab.local	vmnic1	net	PhysicalInterface
beren.ftslab.local	vusb0	net	PhysicalInterface
beren.ftslab.local		mem	PhysicalMemory

F.1.16. ábra. Nem létező objektumra történő hivatkozást tartalmazó input



F.1.17. ábra. Nem létező objektumra történő hivatkozás hibatranszformáció helye

ProblematicInstanceName	ProblematicTimestamp	Name
NOTEXISTINGCPU	1430418780	disk.commands.summation
NOTEXISTINGCPU	1430418780	disk.commandsaborted.summation
NOTEXISTINGCPU	1430418780	disk.devicelatency.average
NOTEXISTINGCPU	1430418780	disk.kernellatency.average
NOTEXISTINGCPU	1430418780	disk.numberread.summation
NOTEXISTINGCPU	1430418780	disk.numberwrite.summation
NOTEXISTINGCPU	1430418780	disk.queuelatency.average
NOTEXISTINGCPU	1430418780	disk.read.average
NOTEXISTINGCPU	1430418780	disk.totallatency.average
NOTEXISTINGCPU	1430418780	disk.totalreadlatency.average
NOTEXISTINGCPU	1430418780	disk.totalwritelatency.average
NOTEXISTINGCPU	1430418780	disk.write.average
NOTEXISTINGCPU	1430418780	storageadapter.totalreadlatency.average
NOTEXISTINGCPU	1430418780	storageadapter.totalwritelatency.average
NOTEXISTINGCPU	1430418780	storagepath.totalreadlatency.average
NOTEXISTINGCPU	1430418780	storagepath.totalwritelatency.average
naa.5000cca00a2a07b8	1430418780	storageadapter.totalreadlatency.average
naa.5000cca00a2a07b8	1430418780	storageadapter.totalwritelatency.average
naa.5000cca00a2a07b8	1430418780	storagepath.totalreadlatency.average
naa.5000cca00a2a07b8	1430418780	storagepath.totalwritelatency.average
sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.commands.summation
sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.commandsaborted.summation
sas.5005076b08802b82-sas.5000cca00a2a07b9-naa.5000cca00a2a07b8	1430418780	disk.devicelatency.average

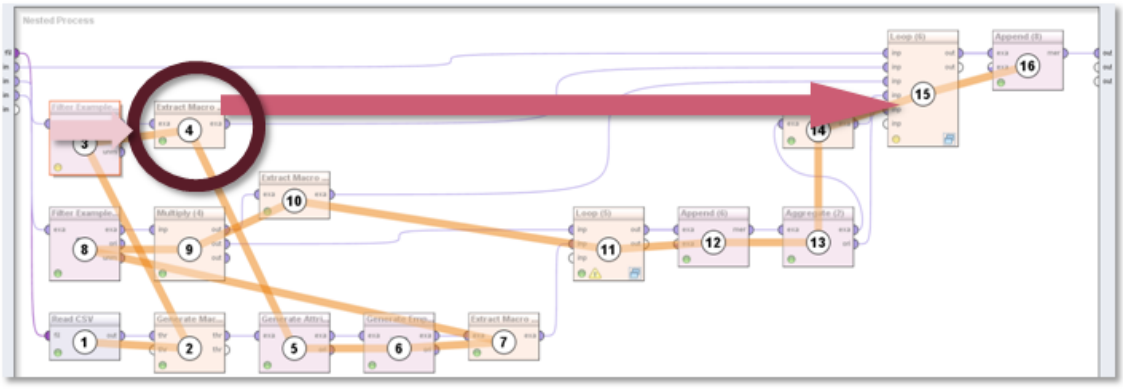
F.1.18. ábra. Nem létező objektumra történő hivatkozás kimenetre gyakorolt hatása

Hibás objektumra történő hivatkozás

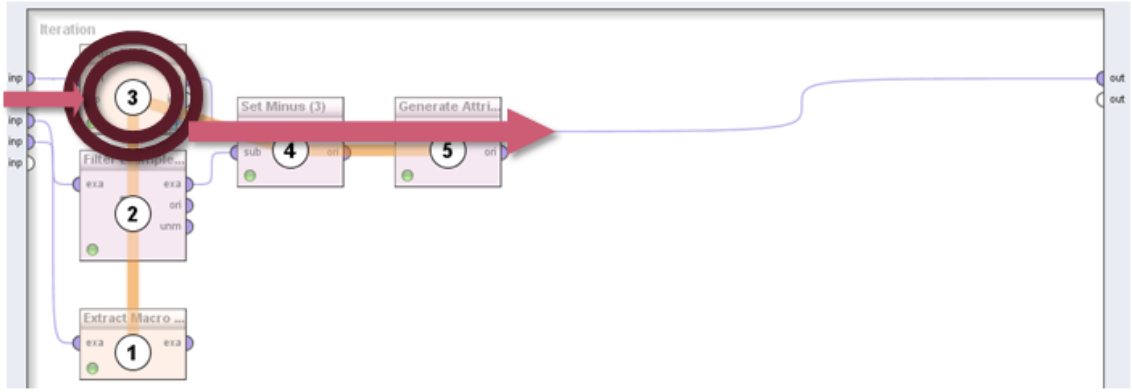
Az EntityMapping táblában a gépnevek egyikéhez hibás érték van megadva, pl. Guest helyett Host (F.1.19. ábra). Így a gépnévre történő szűrés során, a géphez tartozó hibás typot olvassa be a folyamat az EntityType makróba (F.1.20. ábra). Ebből pedig az következik, hogy hibás döntési ágra fut a *Branch* (F.1.21. ábra), így más metrikák mintavételezését várjuk el, mint az adott példány esetében kellene, tehát a kimeneten megjelenő nem mintavételezett metrikák nem fedik a valóságot.

MachineName	Entity
beren.ftslab.local	Host
elrond.ftslab.local	Host
luthien.ftslab.local	Host
Sipp	Guest
Clearwater	Host

F.1.19. ábra. Hibás objektumra történő hivatkozást tartalmazó input



F.1.20. ábra. Hibás objektumra történő hivatkozás hibatranszformáció helye (*ExtractMacro*)



F.1.21. ábra. Hibás objektumra történő hivatkozás hibatranszformáció helye (*Branch*)

F.2. Az esettanulmány hibaterjedés analízise

Ebben a függelékben bemutatjuk a teljes mintapélda hibaterjedési modelljét, az általunk készített leíró nyelvben.

A hibaterjedés bemeneti modelljében szerepeltetünk minden információt a rendszerből. Először fel kell írnunk a a lehetséges változótípusokat, ez látható az F.2.1. ábrán. Itt egy egyszerűsített modell szerepel, amelyben csak a legegyszerűbb típusokat szerepeltetjük a könnyű érthetőség kedvéért.

```
VariableType Basic {
    value(Ok, ExistError)
}
VariableType Category extends Basic
VariableType Numerical extends Basic
VariableType BasicRow{
    existence(Ok, Error)
}
VariableType Row extends BasicRow/*{
    existence(refine Error(ExistMissing, ExistSuperfluous))
}*/
TokenType Row{
    Row State
}
TokenType DataToken{
    Basic State
}
TokenType Category extends DataToken{
    Category State
}
TokenType Numerical extends DataToken{
    Numerical State
}
```

F.2.1. ábra. Változótípusok felírása

Ez után felírtuk a modellben szereplő általános komponenstípusokat és azok viselkedését (F.2.2). Ez és az előző része a modellnek kiemelhető egy könyvtárként külön állományokba, így ennek az információnak nem kell szerepelnie minden egyes modellben.

```
25 ComponentType Filter{
26     TokenInOut row param Row row
27     TokenInOut filter param DataToken filter
28     Rules{
29         filter.State.value.ExistError | row.State.existence.Error
30                 -> row.State.existence.Error
31         !(filter.State.value.ExistError | row.State.existence.Error)
32                 -> row.State.existence.Ok
33     }
34 }
35 ComponentType Aggregate{
36     TokenInOut groupBy param DataToken groupBy
37     TokenInOut aggregate param DataToken aggregate
38     TokenInOut row param Row row
39     Rules{
40         aggregate.State.value.ExistError | row.State.existence.Error
41                 -> aggregate.State.value.ExistError
42         !(aggregate.State.value.ExistError | row.State.existence.Error)
43                 -> aggregate.State.value.Ok
44         groupBy.State.value.ExistError | row.State.existence.Error
45                 -> row.State.existence.Error
46         !(groupBy.State.value.ExistError | row.State.existence.Error)
47                 -> row.State.existence.Ok
48     }
49 }
50 ComponentType GenerateMacro{
51     TokenInOut macro param DataToken source
52 }
53 ComponentType GenerateAttribute{
54     TokenInOut generatedAttribute param DataToken generatedAttribute
55 }
```

F.2.2. ábra. Komponenstípusok

A következő szakaszban felírtunk pár változó és komponenstípust, ami segíti az analízis elvégzését. Létrehoztunk egy változót és komponenst a *boxplot* rajzolásnak, annak ellenére, hogy ez nem szerepel a folyamatmodellben. Ez látható az F.2.3. ábrán. Ennek a komponensnek a segítségével kaphatunk olyan kimeneteket, amelyek már *boxplot* rajzolás specifikusak.

```
VariableType BoxPlot{
    Min(Ok, Error)
    Max(Ok, Error)
    Q1(Ok, Error)
    Q2(Ok, Error)
    Q3(Ok, Error)
    MissingBox(True, False)
    ExtraBox(True, False)
}
ComponentType BoxPlot{
    TokenInOut row param Row row
    TokenInOut value param DataToken value
    TokenInOut category param DataToken category
    TokenOut boxPlot boxPlot
    Create Token boxPlot{
        BoxPlot State
    }
    Rules{
        category.State.value.ExistError
                -> boxPlot.State.ExtraBox.True | boxPlot.State.MissingBox.True
        category.State.value.Ok
                -> boxPlot.State.ExtraBox.False & boxPlot.State.MissingBox.False
        row.State.existence.Error | value.State.value.ExistError
                -> boxPlot.State.Max.Error | boxPlot.State.Min.Error |
                    boxPlot.State.Q1.Error | boxPlot.State.Q2.Error |
                    boxPlot.State.Q3.Error
        row.State.existence.Ok & value.State.value.Ok
                -> !(boxPlot.State.Max.Error | boxPlot.State.Min.Error |
                        boxPlot.State.Q1.Error | boxPlot.State.Q2.Error |
                        boxPlot.State.Q3.Error
                    )
    }
}

ComponentType boxplotComponent{
    TokenInOut row param boxplotComponent prev.row
    TokenInOut metricId param boxplotComponent prev.metricId
    TokenInOut value param boxplotComponent prev.value
    TokenInOut timeStamp param boxplotComponent prev.timeStamp
}
```

F.2.3. ábra. A boxplot specifikus komponensek

Majd példányosítjuk a folyamatba belépő tokeneket (F.2.4. ábra).

```
Create Token row extends Row
Create Token metricId extends Category
Create Token value extends Numerical
Create Token timeStamp extends Numerical
Create Token CSVName extends DataToken
```

F.2.4. ábra. A folyamatba belépő tokenek példányosítása

Ez után létrehozzuk a folyamatnak megfelelő komponenst, hozzáadjuk a megfelelő lekötéseket. Lekötöttük, hogy egyszerre csak egy hiba jöhet be a rendszerbe, valamint pár változót létrehoztunk lekötöttünk, jelezve, hogy azokban nem fordul elő hiba. Ez látható az F.2.5. ábrán.

```
104 SubComponent boxplotProcess{
105     TokenIn row row
106     TokenIn metricId metricId
107     TokenIn value value
108     TokenIn timeStamp timeStamp
109     TokenIn CSVNames CSVName
110     Rules{
111         //Csak egyszeres hiba
112         1 of(CSVNames.State.value.ExistError, metricId.State.value.ExistError,
113             timeStamp.State.value.ExistError, value.State.value.ExistError,
114             row.State.existence.Error
115         )
116         CSVNames.State.value.Ok
117         //metricId.State.value.Ok
118         timeStamp.State.value.Ok
119         //row.State.existence.Ok
120     }
```

F.2.5. ábra. A folyamat példányosítása a lekötésekkel

Végül példányosítjuk az összes folyamatlépést (F.2.6. ábra).

```
121     Components{
122         SubComponent LoopMeasurements extends boxplotComponent(prev = boxplotProcess){
123             TokenInOut row boxplotProcess.row
124             TokenInOut metricId boxplotProcess.metricId
125             TokenInOut value boxplotProcess.value
126             TokenInOut timeStamp boxplotProcess.timeStamp
127             TokenOut machineName DropTimestamp.machineName
128             Components{
129                 Component ReadMeasurements extends
130                         boxplotComponent(prev = LoopMeasurements){}
131                 Component ValueToNumber extends
132                         boxplotComponent(prev = ReadMeasurements){}
133                 Component MachineNameMacro extends
134                         boxplotComponent(prev = ValueToNumber),
135                         GenerateMacro(source = boxplotProcess.CSVNames){}
136                 Component FilterByMetricId extends
137                         boxplotComponent(prev = MachineNameMacro),
138                         Filter(row = MachineNameMacro.row,
139                             filter = MachineNameMacro.metricId
140                         ){}
141                 Component AvgValue extends
142                         boxplotComponent(prev = FilterByMetricId),
143                         Aggregate(row = FilterByMetricId.row,
144                             aggregate = FilterByMetricId.value,
145                             groupBy = FilterByMetricId.timeStamp
146                         ){}
147                 Component AddMachineName extends
148                         boxplotComponent(prev = AvgValue),
149                         GenerateAttribute(generatedAttribute = MachineNameMacro.macro){}
150                 Component DropTimestamp extends
151                         boxplotComponent(prev = AddMachineName){
152                     TokenInOut machineName AddMachineName.generatedAttribute
153                 }
154             }
155         }
157         Component AppendResults extends
158                 boxplotComponent(prev = DropTimestamp){
159             TokenInOut machineName DropTimestamp.machineName
160         }
161         Component PlotBoxPlot extends
162                 boxplotComponent(prev = AppendResults),
163                 BoxPlot(row = AppendResults.row,
164                     value = AppendResults.value,
165                     category = AppendResults.machineName
166                 ){
167             Rules{
168                 boxPlot.State.ExtraBox.False
169                 boxPlot.State.MissingBox.False
170                 boxPlot.State.Max.Ok
171                 boxPlot.State.Min.Error
172                 boxPlot.State.Q1.Ok
173                 boxPlot.State.Q2.Ok
174                 boxPlot.State.Q3.Ok
175             }
176         }
177     }
178 }
```

F.2.6. ábra. A komponensek példányosítása