SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM

TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

RELNORM: RELÁCIÓS ADATBÁZISOK NORMALIZÁLÁSÁHOZ HASZNÁLT ESZKÖZ AZ OKTATÁSBAN

Szerző: **Kiss Gergely**

Számítástechnika MSc. szak, I. évf.

Konzulens: **Milan Čeliković**

egyetemi docens

Temesvár, 2022

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM

RELNORM: RELÁCIÓS ADATBÁZISOK NORMALIZÁLÁSÁHOZ HASZNÁLT ESZKÖZ AZ OKTATÁSBAN

RELNORM: RELATIONAL DATABASE NORMALIZATION TOOL IN EDUCATION

Kiss Gergely

Konzulens:
Milan Čeliković

Kézirat lezárva: 2022. április 11.

Abstract

The use of relational databases and database normalization problems have been a field of interest to many computer scientists over the decades. There have been many solutions to these problems over the years, but there are still gaps in some aspects. As a teaching assistant at the Faculty of Technical Sciences in Novi Sad, we were thinking about developing a program that would be a proper tool for performing teaching assistant tasks. Our main requirement is that the software need to support the normalization algorithms studied at the university: the synthesis and decomposition algorithms. A requirement for the decomposition algorithm is to implement an interactive flow of the algorithm. An additional detail of the requirements was that task series need to be easily specified and replaced.

As a result of these requirements, we developed an application called *RelNorm* to solve the problem of normalizing relational databases as a console application. It is an application written in the *Java* programming language that breaks down a relational schema into multiple relational schemas, depending on the normalization algorithm. Unit tests were used to verify the correctness of the implementation. Code base analysis was done with *SonarCloud* measuring code coverage, reliability and maintainability. The results of the tests and the analysis are included in this paper.

RelNorm has already been proven in practice, as it has already been used by teaching assistants in the 2021/2022 academic year to assemble database normalization task series and review solved tasks.

Keywords: relational database, normalization, relational schemas, software testing, code coverage.

Kivonat

A relációs adatbázisok használata és a velük járó normalizációs problémák már több évtizede foglalkoztatják a számítástechnikában tevékenykedő egyéneket. Az évek során megannyi megoldás született ezekre a problémákra, azok bizonyos aspektusainál mégis hiányosságokat fedezhetünk fel. Az újvidéki Műszaki Tudományok Karán dolgozó tanársegédként egy testreszabott eszköz kifejlesztésében gondolkodtunk, amely kézenfekvő lenne tanársegédi feladatok elvégzéséhez. Fő követelménynek tűztük ki azt, hogy a szoftver támogassa az egyetemen feldolgozott normalizálási algoritmusokat: a szintézis és a dekompozíció algoritmusát. A dekompozíció algoritmusánál további követelmény az algoritmus interaktív lefolyásának a megvalósítása. További részletkövetelmény volt, hogy a feladatsorok könnyen megadhatóak és cserélhetőek legyenek.

Ezen követelmények hatására fejlesztettük ki a *RelNorm* szoftvert, ami a relációs adatbázisok normalizálási problémáját hivatott megoldani konzol applikációként. *Java* programnyelvben íródott applikációról van szó, amely egy relációs sémát bont fel több relációs sémára, a normalizálási algoritmustól függően. Unit-tesztekkel ellenőriztük az algoritmusok helyes megvalósítását, valamint teljes kódbázis elemzést is végrehajtottunk a *SonarCloud* eszközzel. A tesztek eredményét és az elemzést is a dolgozat tartalmazza.

A *RelNorm* a gyakorlatban már bizonyított, ugyanis a 2021–2022-es tanévben már használták a tanársegédek feladatsorok összeállításához és a megoldott feladatlapok átnézésénél.

Kulcsszavak: relációs adatbázis, normalizálás, relációs sémák, szoftvertesztelés, kód lefedettség

Tartalomjegyzék

Ábrák jegyzéke

Táblázatok jegyzéke

1. fejezet

Bevezető

A relációs adatbázisok alapjául szolgáló relációs adatmodell már az 1970-es években sok tudományos munka tárgyát képezte, és az 1990-es évektől kezdve kezdték alkalmazni kommerciális környezetben is (Mogin és Luković 1996). Napjainkban a legkülönfélébb vállalkozások is nagy arányban használnak relációs adatbázisokat valamilyen formában, ezt több felmérés [(*2019 Database Trends* 2019), (Ramel 2015)] és jelentés (Loukides 2022) is bizonyítja. Egy nagy előnye a relációs adatbázisoknak, hogy a relációs adatmodell szilárd matematikai alapokra épül. Ezek az alapok többek között a halmazelméletet és a matematikai relációkat foglalják magukban, ami engedélyezi a relációs algebra és kalkulus műveleteinek a használatát (Mogin és Luković 1996).

Annak érdekében, hogy egy relációs adatbázison el tudjunk végezni bizonyos relációs műveleteket, és pontos eredményekhez jussunk, elengedhetetlen az a feltétel, hogy az adatainkat veszteségmentesen egyesíthető relációkba szervezzük. Veszteséges egyesítésnél adatok tűnhetnek el, vagy megjelenhetnek további téves adatok. Ezen probléma kiküszöböléséhez adatbázis normalizálást kell végrehajtanunk, amivel a relációs sémákat átszervezzük olyan formába, amely veszteségmentes egyesítést eredményez. Ezen normalizálási műveletek sikeres elvégzéséhez a dolgozat két algoritmust is bemutat.

Jelenleg az Újvidéki Egyetem munkatársa vagyok, a Műszaki Tudományok Karán végzek tanársegédi feladatokat. Ezen a karon több szakirányon is folyik relációs adatbázistervezéssel foglalkozó tárgy. A tárgy neve Adatbázisok 2 (*Adatbázisok 2* 2021), melynek keretén belül előadjuk az említett normalizálási algoritmusokat is. Az algoritmusokat feladatok kíséretében dolgozzuk fel kézileg és ugyanígy papíron történik a hallgatók vizsgáztatása is. Egy feladatlap elkészítése, megoldáskulcs ellenőrzése, majd a későbbi hallgatók által kitöltött feladatlapok átnézése potenciálisan sok időt felemészt, valamint nagy felelőséggel is jár.

A dekompozíció normalizálási algoritmusa választási lehetőség elé állítja az alanyt, vagyis különböző útvonalakon juthat el a hallgató a helyes eredményig. Ez az interaktív mozzanat tovább bonyolítja a megoldott feladatok kiértékelését – adott esetben a részeredmények helyességének a megállapítását.

Annak érdekében, hogy enyhítsünk a tanársegédekre helyezedő nehézségeken, kifejlesztettük a *RelNorm* nevezetű szoftvert, mely képes pontosan elvégezni a relációs adatbázis normalizálási feladatait, lehetővé téve az interaktív lépések végrehajtását. A szoftver emellett még megkönnyíti a feladatsorok megadási módját is, ezzel is felgyorsítja a feladatok kidolgozásának a folyamatát.

A bevezető mondatok után először egy irodalmi áttekintés következik, majd a dolgozat célkitűzéseit definiáljuk. Az ezt követő fejezeteket a szoftver kifejlesztésének elméleti háttere, relációs alapfogalmak és algoritmusok bemutatása követi. A bemutatott algoritmusok megvalósításáról lesz szó az ezt követő fejezetben, majd későbbi fejezetekben a szoftvertesztelés és annak eredményeinek az elemzése kap helyet. Végül záró fejezetként összefoglalásra kerülnek a dolgozatban leírtak.

2. fejezet

Irodalom áttekintés

A bevezető részben már szó volt a relációs adatmodell történetéről, és arról a tényről is, hogy már a múlt század második felében sok kutatói munka tárgyát képezte. Átfogó és bizonyított elméleti alapokkal vághatunk tehát neki a téma kidolgozásának. Ebben a dolgozatban az újvidéki Műszaki Tudományok Karán is tekintélyes irodalomra támaszkodunk (Mogin és Luković 1996), (Mogin, Luković és Govedarica 2004), (Čeliković és tsai. 2021) és (Kordić és tsai. 2018). A magyar szakterületen a Budapesti Műszaki Egyetem profeszorának a könyvét (Gajdos 2019) idézem a dolgozatom több pontján. A magyar és szerb irodalom nagy hányadában átfedik egymást, ez is a többévtizedes kiforrt és megszilárdult alapoknak köszönhető.

Több kutatás is foglalkozott az évek során az adatbázisok normalizációjának a problémájával. A kutatásaink során kifejezetten ügyeltünk arra, hogy olyan irodalmat szemlézzünk, amelyek gyújtópontjában az oktatás áll.

Antonia Mitrovic munkájában (Mitrovic 2002) egy hallgató-központú relációs adatmodellel foglalkozó weboldallal foglalkozik, amely – többek közt relációs adatbázisok normalizációjáról is szóló – kérdések-válaszok formájában próbálja meg átadni a tudást a hallgatóknak.

Hongbo Du és Laurent Wery 1999-es dolgozatában (Hongbo és Wery 1999) foglalkoznak konkrét adatbázis normalizációs problémamegoldó eszközzel. Ez a szoftver grafikus felhasználói felülettel is rendelkezik, mely az 1990-es évek esztétikai világát nyújtja. Feladatok beolvasása nehézkesnek tűnik, mivel több dialóguson keresztül lehet csak megadni függőségeket, ami időigényes feladat. Nincs lehetőség algoritmus kiválasztására a normalizációs probléma megoldására. Kimenetnek a *Microsoft Office* szoftvercsomag *Access* nevezetű adatbáziskezelő programát használja, melyben létrehozza a normalizáció eredményeként létrejött relációs sémákat. Ez bizonyos felhasználási esetekben nagyon is kívánatos eljárás, bár

a mi esetünkben egyszerűbb kimeneti eredményt is elfogadhatónak tartanánk.

Amir Bahmani és munkatársai egy automatikus normalizációs eszközről írnak dolgozatukban (A. Bahmani, Naghibzadeh és B. Bahmani 2008), mely táblázatok kész adatai alapján határoz meg függőségeket és azok alapján generál normalizált relációs sémákat. A mi elvárásaink alapján a bemeneti adatok helyett elegendő volna csak egy attribútumhalmaz és egy függőséghalmaz betáplálása a programba.

Egy kidolgozott normalizációs eszközről ír Nikolay Georgiev a dolgozatában (Georgiev 2008). Az eszköz rendelkezik grafikus felhasználói felülettel, többdialógusos beviteli lehetőségekkel (melyek némileg lassíthatják a bevitel sebességét) valamint csak egy algoritmus áll rendelkezésre. A szoftver leírásának alapján a szerző inkább a hallgató szemszögéből vezette le a funkcionális követelményeket, így inkább a normalizáció elsajátításán van a szoftverben nagyobb hangsúly, mintsem a feladatsorok gyors átnézésében és megoldásában.

3. fejezet

Célkitűzések

A dolgozat céljai között szerepel:

- a relációs adatbázisok normalizálási algoritmusainak megismertetése az olvasóval,
- ezen algoritmusok kivitelezését szolgáló szoftver fejlesztésének a dokumentálása.

A kifejlesztendő szoftver céljai között szerepel:

- relációs adatbázis normalizálási algoritmusoknak a kivitelezése,
- könnyen megadható és cserélhető bemeneti feladatsorok,
- interaktív üzemmód engedélyezése az algoritmus bizonyos lépéseinél,
- forráskód karbantartása nagy lefedettségű tesztekkel.

4. fejezet

Elméleti megalapozás

Ez a fejezet alapvető fogalmakat, folyamatokat és algoritmusokat dolgoz fel, amelyek elengedhetetlenek voltak a végső szoftver kifejlesztése közben.

4.1. A relációs adatmodell alapvető fogalmai

Relációs séma egy rendezett pár (R, C), ahol R attribútumhalmazt, C pedig kényszerhalmazt jelöl (Mogin és Luković 1996). Relációs séma megjelenési formája a *reláció* , amely korlátolt számú *sort* tartalmaz (Mogin és Luković 1996).

Relációs adatbázis séma egy rendezett pár (S, I), ahol S relációs séma halmazt, I pedig relációközi kényszerhalmazt jelöl (Mogin és Luković 1996). Relációs adatbázis séma megjelenési formája a *relációs adatbázis* (Mogin és Luković 1996).

Funkcionális függőségek (röviden függőségek) a relációs sémák integritását őrzik, más szóval a relációkban tárolt adatok közt vezetnek be összefüggéseket. Ha egy adott relációban egy funkcionális függőséget $f : X \to Y$ szemlélünk, akkor X jelöli a baloldali-, míg Y a jobboldali attribútumhalmazt, melyek között funkcionális függőség van. Ebből kifolyólag a szóban forgó reláció bármely két sorára (u és v) érvényes a 4.1 képlet (Mogin és Luković 1996). A függőségek megjelölésénél általában elhanyagoljuk a függőség megnevezését, így $f : X \to Y$ helyett csak $X \to Y$ írunk.

$$(\forall u, v \in r)(u[X] = v[X] \implies u[Y] = v[Y]) \tag{4.1}$$

Egy funkcionális függőség $X \to Y$ akkor *triviális*, ha érvényes $Y \subseteq X$.

Amennyiben egy relációban érvényes egy bizonyos F függőséghalmaz, és egy szemlélt f függőség az F *igaz funkcionális függősége* , akkor az adott reláción érvényes az f függőség

is (Gajdos 2019). Ezt következőképp jelöljük: $F \models f$. Másik meghatározás szerint ez azt jelenti, hogy *f logikai következménye* az F függőséghalmaznak (Mogin és Luković 1996).

Bármelyik függőséghalmazon elvégezhetjük a relációs algebra *projekció* műveletét. A 4.2 képlet mutatja be az F függőséghalmaz projekcióját az X attribútumhalmazra.

$$F|_X = \{V \to W \mid F \models V \to W \wedge VW \subseteq X\} \tag{4.2}$$

Megjegyzés: az attribútumhalmazok uniójának (pl. $V \cup W$) helyett a rövidített megjelölést (VW) használjuk a továbbiakban.

Azt a halmazt 4.3, amely az F függőséghalmaz összes igaz függőségét (logikai következményét) tartalmazza, az *F fuggőséghalmaz lezártjának* hívják (Mogin és Luković 1996).

$$F^+ = \{f \mid F \models f\} \tag{4.3}$$

Két függéshalmaz ekvivalens, amennyiben érvényes a 4.4 képlet.

$$F_1 \equiv F_2 \iff F_1^+ = F_2^+ \tag{4.4}$$

Egy tetszőleges X *attribútumhalmaz lezártja* az F függéshalmazra való tekintettel a 4.5 képlettel van definiálva.

$$X_F^+ = \{A \in \mathbf{U} \mid F \models X \to A\} \tag{4.5}$$

Az attribútumhalmaz lezártjának a kiszámolásához két lépést használunk:

1. $X_0 \leftarrow X$
2. $X_{i+1} \leftarrow X_i \cup \{A \in \mathbf{U} \mid (\exists V \to W \in F)(V \subseteq X_i \wedge A \in W)\}$

Az X_n megjelölést a fenti lépéseknél az n-edik ciklust jelöli. A 2. lépést mindaddig kell ismételni, amíg az X_{i+1} és X_i halmazok különböznek.

Az $X \to Y$ függőség *részleges függőség*, ha érvényes $(\exists X' \subset X)(X' \to Y \in F)$.

Az $X \rightarrow Z$ függőség *tranzitív függőség*, ha érvényes ($F \models X \rightarrow Y \wedge F \models Y \rightarrow Z \wedge F \not\models Y \rightarrow X \wedge Z \notin XY$).

Az X attribútumhalmaz a (R, F) relációs séma *kulcsa*, amennyiben érvényes:

1. $F \models X \rightarrow R$
2. $(\forall X' \subset X)(F \not\models X' \rightarrow R)$

Relációs séma kulcsainak a kiszámolásához definiálnunk kell egy redukció műveletet 4.6. Ez a művelet egy attribútumhalmaz (X) minimális attribútumhalmazát határozza meg, amely nem tartalmaz felesleges attribútumokat (egy előre megadott F függéshalmazra tekintettel).

$$\text{Red(X)} : (\forall A \in X)(A \in (X \setminus \{A\})_F^+ \implies X \leftarrow X \setminus \{A\}) \tag{4.6}$$

A relációs séma kulcsainak a kiszámolásához használt algoritmus:

1. $X \leftarrow R$
2. $(\forall X \in K)(\forall V \rightarrow W \in F)(X \cap W \neq \emptyset \implies X_{newk} \leftarrow (X \setminus W)V)$
3. $K \leftarrow K \cup \{Red(X_{newk})\}$

Elsődleges attribútumnak nevezünk minden olyan attribútumot, amely a relációs séma kulcsát alkotja (4.7 képlet). *Másodlagos attribútumnak* nevezünk minden attribútumot, amely nem alkotja a relációs séma egyik kulcsát sem.

$$K_{pr} = \bigcup_{K \in \mathcal{K}} (K) \tag{4.7}$$

4.2. Normálformák

A *normálformák* megszorítások a relációs séma tulajdonságaira vonatkozóan annak érdekében, hogy a sémákra illeszkedő relációkkal végzett műveletek során egyes nemkívánatos jelenségeket elkerülhessünk (Gajdos 2019). Ezeket a nemkívánatos jelenségeket anomáliáknak hívják és beszúrási, módosítási vagy törlési műveletek során bukkanhatnak fel. Káros hatásuk akár az adott műveletek ellehetetlenítését is jelentheti. Ebben a dolgozatban a következő normálformákat mutatjuk be: *1NF*, *2NF*, *3NF* és *BCNF*.

4.2.1. 1NF

Az $N(R, F)$ relációs séma kielégíti az *1NF* (első normálforma) feltételét, amennyiben az R halmazban kizárólag atomi értékeket hordozó attribútumok szerepelnek (Mogin, Luković és Govedarica 2004). Ez azt jelenti, hogy egy attribútum értéke sem lehet tömb vagy halmaz alakú. Az *1NF* normálforma előfeltétele az összes többi normálformának, és ezt a tényt nem fogjuk külön kiemelni minden egyes normálformánál.

4.2.2. 2NF

Az $N(R, F)$ relációs séma kielégíti a *2NF* (második normálforma) feltételét, amennyiben minden másodlagos attribútum teljesen függ a relációs séma összes kulcsától (4.8 képlet) (Mogin, Luković és Govedarica 2004).

$$(\forall A \in R \setminus K_{pr})(\forall X \in K)(\forall Y \subset X)(F \not\models Y \rightarrow A) \tag{4.8}$$

4.2.3. 3NF

Az $N(R, F)$ relációs séma kielégíti a *3NF* (harmadik normálforma) feltételét, amennyiben minden másodlagos attribútum nem tranzitív függőségben van a relációs séma összes kulcsával (4.9 képlet) (Mogin, Luković és Govedarica 2004).

$$(\forall A \in R \setminus K_{pr})(\forall X \in K)(\forall Y \subseteq R \setminus \{A\})(F \models Y \rightarrow A \implies F \models Y \rightarrow X) \tag{4.9}$$

Létezik egy alternatív definíciója is a *3NF* normálformának. E definíció szerint minden nem triviális függőség bal oldalának tartalmaznia kell a relációs séma egy kulcsát, amennyiben a jobb oldali attribútumhalmaz tartalmaz másodlagos attribútumot (4.10 képlet) (Mogin, Luković és Govedarica 2004).

$$(\forall A \in R \setminus K_{pr})(\forall Y \subseteq R \setminus \{A\})(F \models Y \rightarrow A \implies (\exists X \in K)(X \subseteq Y)) \tag{4.10}$$

4.2.4. BCNF

Az $N(R, F)$ relációs séma kielégíti a *BCNF* – Boyce[1]-Codd[2] normálformát, amennyiben minden nem triviális függőség bal oldala tartalmazza a relációs séma egy kulcsát (4.11 képlet) (Mogin, Luković és Govedarica 2004).

$$(\forall A \in R)(\forall Y \subseteq R \setminus \{A\})(F \models Y \rightarrow A \implies (\exists X \in K)(X \subseteq Y)) \tag{4.11}$$

4.2.5. Normálformák összefüggései

Az irodalomban levezetett bizonyítások alapján elmondható, hogy minden alacsonyabb szintű normálforma elengedhetetlen egy magasabb szintű normálforma teljesítéséhez (4.12 képlet) (Mogin, Luković és Govedarica 2004).

$$\begin{aligned} &\text{BCNF} \implies \text{3NF} \implies \text{2NF} \implies \text{1NF} \\ &\neg\text{1NF} \implies \neg\text{2NF} \implies \neg\text{3NF} \implies \neg\text{BCNF} \end{aligned} \tag{4.12}$$

4.3. Normalizációs algoritmusok

Az adatbázisok normalizálásának célja, hogy egy (vagy több) relációs sémát egy bizonyos normálformára vezessen, a nemkívánatos anomáliák elkerülése érdekében. A következőkben bemutatott normalizációs algoritmusok leírásai az *Adatbázis tervezés alapjai* (Mogin, Luković és Govedarica 2004) valamint *Adatbázisok 2 laborgyakorlati munkafüzetből* (Čeliković és tsai. 2021) származnak.

4.3.1. Szintézis

A szintézis algoritmus kiindulópontja az ún. univerzális relációs séma $(\mathbf{U}, \mathbf{F})$, ahol az $\mathbf{U}$ az univerzális attribútumhalmazt, az $\mathbf{F}$ pedig az univerzális függéshalmazt jelöli. A szintézis elvégeztével n darab relációs sémát kapunk, melyek kielégítik a *3NF* normálformát. A relációközi megszorításokkal kiegészülve megkapjuk a szintézis kimenetét, vagyis az adatbázis sémát (4.13 képlet).

[1] Raymond F. Boyce (1946—1974) amerikai informatikus
[2] Edgar F. Codd (1923—2003) angol informatikus

$$S = \{(R_i, K_i) \mid i \in \{1, 2, ..., n\}\} \tag{4.13}$$

A szintézis algoritmusa néhány lépésből áll:

1. *Minimális függéshalmaz* meghatározása

   Fontos megjegyezni, hogy a minimális függéshalmazban (F_{min}) ekvivalens a kiinduló függéshalmazzal (4.14 képlet). A minimális függéshalmazban a függőségek jobboldali attribútumhalmazában csak egyetlen attribútum található (4.15 képlet), a függőségek teljes függőségek (4.16 képlet), valamint nincs olyan függőség, amelyik elhagyható (4.17 képlet) (Gajdos 2019).

$$F \equiv F_{min} \tag{4.14}$$

$$(\forall X \rightarrow A \in F_{min})(A \in \mathbf{U}) \tag{4.15}$$

$$(\forall X \rightarrow A \in F_{min})(\forall X' \subset X)(F \not\vdash X' \rightarrow A) \tag{4.16}$$

$$(\nexists X \rightarrow A \in F_{min})(F_{min} \setminus \{X \rightarrow A\} \equiv F_{min}) \tag{4.17}$$

   A minimális függéshalmaz fent említett tulajdonságait a következő algoritmussal érhetjük el:

```
for X->Y in F:
  Fmin <- Fmin + {X->A | A in Y}

for X->A in Fmin:
  for B in A:
    if Fmin |= X\{B}->A:
      Fmin <- Fmin\{X->A} + {X\{B}->A}

for X->A in Fmin:
  if X->A in (Fmin\{X->A})+:
    Fmin <- F\{X->A}
```

2. Minimális függéshalmaz átalakítása

A minimális függéshalmazt fel kell osztani partícióhalmazra (4.18 képlet), ahol minden egyes partícióba olyan függőségeket csoportosítunk, melyeknek a baloldali halmazai megegyeznek (4.19, 4.20 és 4.21 képletek).

$$\mathbf{G} = \{G(X_i) \mid i \in \{1, ..., n\}\} \tag{4.18}$$

$$G(X_i) = \{Y \rightarrow A \in F_{min} \mid Y = X_i\} \tag{4.19}$$

$$(\forall i, j \in \{1, ..., n\})(i \neq j) \iff (X_i \neq X_j) \tag{4.20}$$

$$(\forall Y \rightarrow A \in F_{min})(\exists G(X_i) \in \mathbf{G})(Y = X_i) \tag{4.21}$$

A megformált partíciókat egyesíteni kell az ekvivalens baloldali halmazok szerint, vagyis minden pár partíció $G(X_i)$ és $G(X_j)$, amelyre érvényes $(X_i)_F^+ = (X_j)_F^+$, azokat egyesítjük $G(X_i, X_j)$ partícióvá, majd a partícióhalmazt a 4.22 képlet szerint módosítjuk.

$$\mathbf{G} \leftarrow \mathbf{G} \setminus \{G(X_i), G(X_j)\} \cup \{G(X_i, X_j)\} \tag{4.22}$$

Miután ezeket a bizonyos partíciókat egyesítettük, felmerülhet az a veszély, hogy tranzitív függőségeket hozunk létre ezekben a partíciókban, ezért létre kell hozni egy J halmazt (4.23 képlet), mely tartalmazza a tranzitivitás megelőzésére alkalmas függőségeket. Annak érdekében törekedünk a tranzitív függőségek felszámolására, hogy a relációs sémák teljesíteni tudják majd a *3NF* normálformát. Ezeknek az újabb függőségeknek a hozzáadásával lehet, hogy sikerült kiküszöbölni a tranzitív függőségeket, de potenciálisan elhagyható függőségek jöttek létre. Az érintett partíciókból átmenetileg ki kell vonni ezeket a függőségeket, majd törölni kell a feleslegessé váltakat (4.24 képlet).

$$J = \{X_i \rightarrow X_j, X_j \rightarrow X_i\} \tag{4.23}$$

$$G(X_i, X_j) \leftarrow G(X_i, X_j) \setminus \big(\{X_i \rightarrow A \mid A \in X_j\} \cup \{X_j \rightarrow A \mid A \in X_i\}\big) \tag{4.24}$$

Az elhagyható függőségek törlése végett létrehozunk egy M halmazt (4.25 képlet), majd ennek a halmaznak a tekintetében végezzük a függőségek egyszerűsítését – a(z) 4.17 képlethez hasonlóan.

$$M = \bigcup_{G_X \in \mathbf{G}} (G_X) \cup J \tag{4.25}$$

Miután a megfelelő partíciókból törlésre kerültek a felesleges függőségek, visszaállítjuk a J halmazbeli függőségeket a megfelelő partíciókba.

3. Relációs adatbázis séma létrehozása

   Minden $G_X \in \mathbf{G}$ partíció egy relációs sémát alkot, ahol az X halmaz jelenti a séma kulcsát. A partíció függőségeiben előforduló attribútumok pedig a séma attribútumhalmazát képezik. A relációközi megszorításokat az idegen kulcsok megszorításai alkotja.

4. Veszteségmentes sémafelbontás megőrzése

   Annak érdekében, hogy meggyőződjünk a veszteségmentes sémafelbontásról, le kell ellenőrizni, hogy bár egy relációs séma kulcsa megegyezik az univerzális relációs séma kulcsával. Ha igen, akkor veszteségmentesen bontottuk fel a sémát. Amennyiben a válasz nem, további relációs sémára lesz szükségünk, melynek kulcsa megegyezik az univerzális relációs séma egyik szabadon választott kulcsával, az attribútumhalmaz pedig a kiválasztott kulcsot képező attribútumokkal.

4.3.2. Dekompozíció

A dekompozíció algoritmus kiindulópontja az ún. univerzális relációs séma ($\mathbf{U}, \mathbf{C}$), ahol az $\mathbf{U}$ az univerzális attribútumhalmazt, az $\mathbf{C}$ pedig az univerzális megkötéshalmazt jelöli. A $\mathbf{C}$

halmaz magában foglalja az univerzális relációs séma függőségeit valamint a többértékű függőségeket is. Mivel a dolgozat csak olyan algoritmusokat taglal, amelyek legfeljebb a *BCNF* normálformát elégítik ki, ezért nem fogjuk figyelembe venni a többértékű függőségeket. A dekompozíció elvégeztével n darab relációs sémát kapunk, melyek kielégítik a *BCNF* normálformát. A relációközi megszorításokkal kiegészülve megkapjuk a dekompozíció kimenetét, vagyis az adatbázis sémát.

A dekompozíció algoritmusa:

1. Megfelelő függőség kiválasztása

   A dekompozíció lépéseinek az első eleme a megfelelő függőség kiválasztása, ami alapján felosszuk az adott relációs sémát. A kívánt $X \rightarrow Y$ függőséget három kritérium alapján tudjuk kiválasztani:

   i) P1 kritérium szerint egy nemtriviális függőséget kell választanunk, ahol az Y halmaz nem szuperkulcs, valamint a függőséghalmaz meghatározott szétválasztása nem jár függőségvesztéssel (4.26 képlet).

   $$(Y \not\subseteq X) \wedge (R \not\subseteq X^+) \wedge \left(F^+ = (F|_{X(R\setminus Y)} \cup F|_{XY})^+\right) \tag{4.26}$$

   ii) P2 kritérium szintén nemtriviális függőség kiválasztását terjeszti elő, melynek jobb- és baloldali attribútumhalmazainak az uniója különböznek az adott sémareláció attribútumhalmazától (R), valamint a függőséghalmaz meghatározott szétválasztása nem jár függőségvesztéssel (4.27 képlet).

   $$(Y \not\subseteq X) \wedge (XY \subset R) \wedge \left(F^+ = (F|_{X(R\setminus Y)} \cup F|_{XY})^+\right) \tag{4.27}$$

   iii) P3 kritérium szintén nemtriviális függőség kiválasztását terjeszti elő, ahol az Y halmaz nem szuperkulcs (4.28 képlet). Az előző két kritériummal ellentétben a P3 kritérium nem szabja feltételként a függőségvesztés kitételt.

   $$(Y \not\subseteq X) \wedge (R \not\subseteq X^+) \tag{4.28}$$

A megfelelő függőség kiválasztásánál ügyelni kell arra, hogy minél magasabb kritérium teljesüljön.

2. Relációs séma szétválasztása a kiválasztott függőség alapján

   Amennyiben sikerült kiválasztani a megfelelő $X \rightarrow Y$ függőséget, akkor az adott relációs sémát (R, F) a r_1 és r_2 relációs sémákra tudjuk felbontani (4.29 képlet).

$$\begin{aligned} r_1 &: (R_1, F_1) = (XY, F|_{XY}) \\ r_2 &: (R_2, F_2) = ((R \setminus Y)X, F|_{(R \setminus Y)X}) \end{aligned} \tag{4.29}$$

   Ilyen felbontás mellett teljesülnek a veszteségmentes összevonás feltételei (4.30 képlet), ahol a K_1 és K_2 halmazok a megfelelő relációs sémák kulcshalmazait jelölik.

$$(R_1 \cup R_2 = R) \wedge (K_1 \subseteq R_1 \cap R_2 \vee K_2 \subseteq R_1 \cap R_2) \tag{4.30}$$

3. Normálforma vizsgálat

   A szétbontott relációs sémákat normálforma vizsgálat alá helyezzük, és amennyiben nem teljesítik a *BCNF* normálformát, további dekompozíciónak vetjük alá a sémákat, kezdve az algoritmusban szereplő 1. ponttal. Ezt a folyamatot rekurzív módon hajtjuk végre, vagyis az első pontban szereplő séma helyét a második pontban kapott sémák veszik át (amennyiben azok nem teljesítik a *BCNF* normálformát). Azokat a relációs sémákat, amelyek teljesítik a *BCNF* normálformát, elvégzetteknek (dekomponáltaknak) tekintünk.

4. A dekompozíciós fa kiértékelése

   A relációs sémák szétválasztása során bináris fa jön létre, melynek levelei alkotják a dekomponált relációs sémahalmazt. Ezt a sémahalmazt további kiértékelésnek vetjük alá, mégpedig az ekvivalens kulccsal rendelkező sémákat összevonjuk. Ezzel a lépéssel visszanyerhetünk időközben elvesztett függőségeket – amennyiben a P3 kritérium alapján tudtunk csak függőséget választani a dekompozíció során. Ezekkel a visszanyert függőségekkel viszont kockáztatjuk az elért *BCNF* normálformát, de a függőségek megőrzése érdekében beáldozhatjuk ezeket a sémákat.

5. Veszteségmentes sémafelbontás megőrzése

   A szintézis algoritmusához hasonlóan meg kell győződjünk a veszteségmentes sémafelbontásról, amit úgy érünk el, hogy leellenőrizzük, hogy bár egy relációs séma kulcsa megegyezik az univerzális relációs séma kulcsával.

4.4. Szoftvermodellezési szempontok

Szoftvermodellezéshez a *UML* modellnyelvet használtam, hogy egy konkrét programnyelvtől független leírást mutathassak be a fejlesztett szoftverről. Mivel a szoftver tervezésétől kezdve objektum-orientált programnyelvi paradigmában gondolkodtam, ezért adatmodellezéshez osztálydiagramot használtam. Az adatbázis normalizálási algoritmusokat szekvenciadiagramokkal terveztem ki.

A következő fejezet részletesen bemutatja a szoftverfejlesztéshez használt diagramokat, valamint konkrét kódrészleteket is magában foglal.

4.5. Szoftvertesztelés

A *szoftvertesztéléssel* meg tudjuk előzni azokat a hibákat, amelyek még a szoftver üzembe helyezése előtt jelentkeznek. Többféle tesztelési módszer létezik, de ebben a dolgozatban a unit-tesztekre fókuszálunk. A *unit-tesztek* olyan automatizált tesztek, melyek kisebb egységnyi kódrészletet gyorsan és elszigetelt módon ellenőriznek (Khorikov 2020).

A unit-tesztek három fázisból állnak:

1. előkészítés (ang. *arrange*) — a tesztben szereplő objektumok létrehozása, változók értékeinek a megadása;

2. végrehajtás (ang. *act*) – a tesztelni kívánt algoritmus metódusának az előhívása;

3. megerősítés/megkötés (ang. *assert*) – a kapott és a kívánt eredmények összehasonlítása.

Egy unit-teszt akkor lesz sikeres, ha a kapott és kívánt eredmények összehasonlítása helytálló.

A dolgozat célkitűzései között szerepel a nagy lefedettségű tesztek alkalmazása. A *kód lefedettség* egy másik szempont, ami alapján vizsgáljuk egy szoftver minőségét. Ennek

érdekében bevezetünk egy kód lefedettségi mérőszámot, amely megmutatja a tesztek lefedettségének a mértékét. Ez a mérték egy arány a tesztek által lefuttatott és a teljes kódbázis sorainak száma között (4.31 képlet) (Khorikov 2020).

$$\text{kód lefedettség} = \frac{\text{lefuttatott kódsorok száma}}{\text{kódbázis sorainak a száma}} \tag{4.31}$$

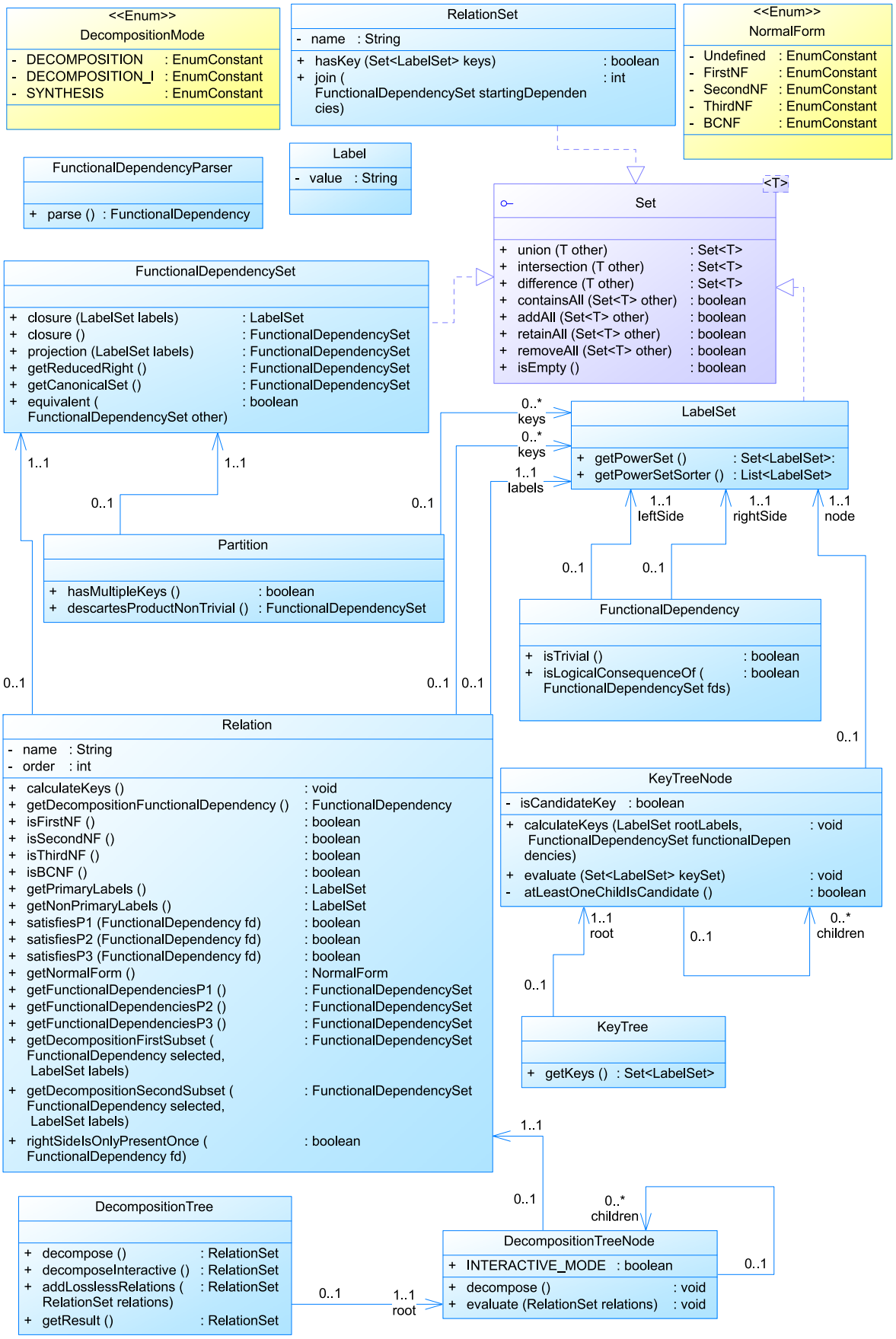
5. fejezet

Gyakorlati megvalósítás

A *RelNorm* fejlesztéséhez *Java* programnyelvet használtunk. Ezt a döntést kisebb elemzés előzte meg, amikor arra kerestük a választ, hogy milyen előnyökkel jár a *Java* programnyelv használata ilyen probléma megoldásához. Kétségkívül objektum-orientált programnyelvet helyeztünk előtérbe, hiszen a funkcionális függőségek és attribútumok léte igazolja a különböző objektumok létét a rendszerünkben. Mindez mellett ezeket az objektumokat kollekciókba (halmazokba) kell rendeznünk, ezért egy objektum-orientált nyelv tűnt ígéretesnek a megvalósításhoz. A *Java* programnyelv `equals`, `hashCode` és `toString` beépített metódusainak használata is pozitívumot hozott a fejlesztéshez. A *Java* programnyelv 8-as verziójától *adatfolyamot* is alkalmazni lehet, amely különféle szűréseket, átképzéseket, kalkulációkat és egyéb kollekciókon való műveleteket is engedélyez. További szempont volt egy automatizált kódépítő eszköz használata, amellyel könnyedén felépíthetjük, tesztelhetjük és lefuttathatjuk a programot. A *Maven* eszközre esett a választás, mivel széleskörben elterjedt, és a számunkra megfelelő funkciókkal van ellátva.

5.1. Osztálydiagram

A *RelNorm* fejlesztése az osztálydiagrammal (5-1 ábra) kezdődött. Az osztálydiagramon – az átláthatóság érdekében – mellőztük a konstruktőrok és egyéb `get`/`set` metódusok leírását, valamint a már említett *Java* beépített metódusokét is. A `Label` és a `LabelSet` osztályok elnevezése megegyezik az irodalomban használt elnevezésekkel: attribute és attribute set. A `Set` interface a `java.util` csomagból került felhasználásra.



5-1. ábra. *RelNorm* osztálydiagram

5.2. Alapvető algoritmusok/kódrészletek

Ebben a részben az előző fejezetben leírt alapvető relációs fogalmak és műveletek algoritmusainak a megvalósítása kerül bemutatásra.

Az attribútumhalmaz lezártjának a definícióját (4.5 képlet) a függőséghalmaz `closure` metódusával tudjuk kiszámolni:

```
public LabelSet closure(LabelSet labels) {
  LabelSet result = new LabelSet(labels);
  LabelSet lastResult;
  do {
    lastResult = new LabelSet(result);

    this.items.forEach(fd -> {
      if(result.containsAll(fd.getLeftSide())) {
        result.addAll(fd.getRightSide());
      }
    });
  } while(!lastResult.equals(result));
  return result;
}
```

A függőséghalmaz projekcióját (4.2 képlet) a függőséghalmaz `projection` metódusával tudjuk kiszámolni:

```
public FunctionalDependencySet projection(LabelSet labels) {
  FunctionalDependencySet projection = new FunctionalDependencySet();
  for(LabelSet subset: labels.getPowerSet()) {
    LabelSet closure = this.closure(subset);
    closure.retainAll(labels);
    closure.removeAll(subset);
    if(!closure.isEmpty()) {
      FunctionalDependency fd =
          new FunctionalDependency(subset, closure);
      projection.add(fd);
    }
  }
  return projection;
}
```

A fenti kódrészletben szerepel egy `getPowerSet` nevezetű metódus, amely egy adott attribútumhalmaz összes részhalmazának a halmazát számolja ki (*Obtaining a Power Set of a Set in Java* 2020).

Az igaz következmény, azaz egy függőséghalmazhoz viszonyított függőség logikai következményének a definícóját (4.3 képlet) az attribútumhalmaz `isLogicalConsequenceOf` metódusával lehet kiszámolni.

Két függőséghalmaz egybevágóságát (4.4 képlet) a függőséghalmaz equivalent metódusa számolja ki.

```
public boolean isLogicalConsequenceOf(FunctionalDependencySet
    dependencies) {
  LabelSet closure = dependencies.closure(leftSide);
  return closure.containsAll(rightSide);
}
```

```
public boolean equivalent(FunctionalDependencySet other) {
  return
    this.stream().allMatch(fd -> fd.isLogicalConsequenceOf(other)) &&
    other.stream().allMatch(fd -> fd.isLogicalConsequenceOf(this));
}
```

A relációs sémák kulcshalmazához az előző fejezetben leírt algoritmust (4.6 képlet) használjuk. A megvalósításhoz elengedhetetlen egy ún. *kulcsfa* használata. Ez a kulcsfa olyan csomópontokból áll össze, melyek tartalmaznak egy atribútumhalmazt, információt a kulcsjelölti státuszukról valamint egy kollekciót a gyermekeiről. A gyermek csomópont pontosan egy attribútummal kevesebbet tartalmaz a szülőtől. Ha egy megadott attribútumhalmaz $U = \{A, B, C, D\}$ és függőséghalmaz $F = \{AC \rightarrow B, BC \rightarrow D, A \rightarrow B, B \rightarrow A\}$ kulcsait szeretnénk megkapni ($K = \{AC, BC\}$), akkor a(z) 5-2 ábra mutatja be ennek a példának a kulcsfáját.

A zölddel jelölt csomópontokat további rekurzív számításnak vetjük alá mindaddig, amíg egy csomópont kulcsjelölti státusza meg nem szűnik (piros) vagy már az adott csomópont attribútumhalmazát bejártuk (sárga). A kulcsfa kiértékelésénél azokat a zöld csomópontokat tekintjük kulcsnak, amelyek kulcsjelöltek, de egyik gyermekük sem kulcsjelölt.

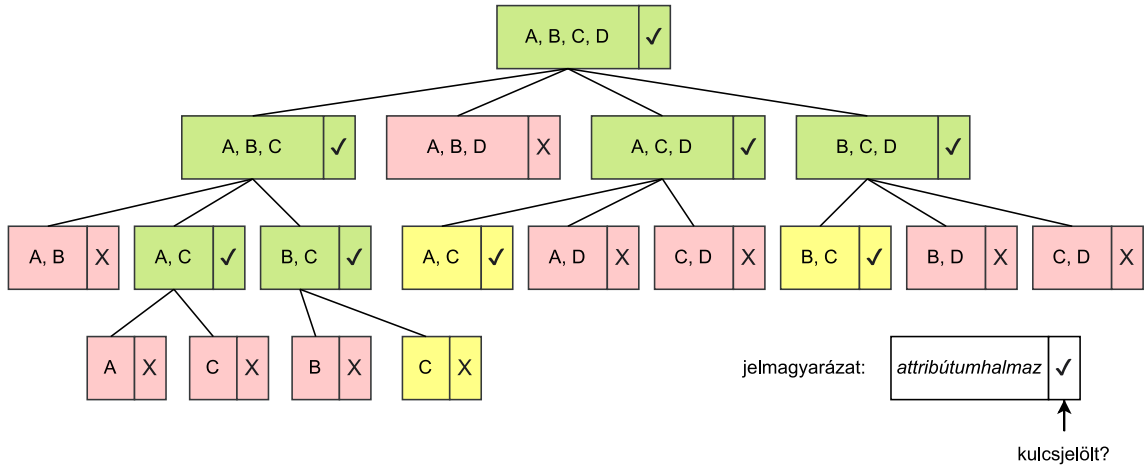
5.3. Normálformák megvalósítása

Az első normálforma, azaz az *1NF* előzetes megegyezés szerint mindig teljesítve lesz, mivel a normalizációs algoritmusok nem tudják megállapítani egyes attribútumok struktúráját.

```
public boolean isFirstNF() {
  return true;
}
```

Az első normálformán túl szükségünk van további kisegítő metódusokra. Ilyen metódusok a primáris és szekundáris attribútumhalmazt kiszámoló metódusok. A primáris attribútumok a `getPrimaryLabels` metódussal kaphatóak meg.

```
private LabelSet getPrimaryLabels() {
  LabelSet primaryLabels = new LabelSet();
  keys.forEach(primaryLabels::addAll);
  return primaryLabels;
}
```



5-2. ábra. Kulcsfa

A szekundáris attribútumok a `getNonPrimaryLabels` metódussal kaphatóak meg.

```
private LabelSet getNonPrimaryLabels() {
  LabelSet allLabels = new LabelSet(labels);
  allLabels.removeAll(getPrimaryLabels());
  return allLabels;
}
```

A reláció `isSecondNF` metódusával kapjuk meg azt az információt, hogy egy adott reláció teljesíti-e a *2NF* normálformát annak definíciója szerint (4.8 képlet).

```
public boolean isSecondNF() {
    LabelSet nonPrimaryLabels = getNonPrimaryLabels();
    if(nonPrimaryLabels.isEmpty()) {
        return true;
    }

    return getKeys().stream().allMatch(key -> {
        Set<LabelSet> subsets = key.getSubsets();
        return subsets.stream().allMatch(subset ->
            nonPrimaryLabels.stream().noneMatch(nonPrimaryLabel ->
                functionalDependencies.closure(subset)
                    .contains(nonPrimaryLabel)));
    });
}
```

A *3NF* normálformának az alternatív definíciója alapján (4.10 képlet) készült el a relációs séma `isThirdNF` metódusa.

```
public boolean isThirdNF() {
  LabelSet nonPrimaryLabels = getNonPrimaryLabels();
  if(nonPrimaryLabels.isEmpty()) {
    return true;
  }

  return functionalDependencies.stream().allMatch(fd -> {
    if(nonPrimaryLabels.containsAll(fd.getRightSide())) {
      return getKeys().stream().anyMatch(key ->
```

```
        fd.getLeftSide().containsAll(key));
    } else {
      return true;
    }
  });
}
```

A *BCNF* normálforma definíciója (4.11 képlet) alapján létrejött `isBCNF` metódussal számítható ki, hogy egy relációs séma teljesíti-e a *BCNF* normálformát.

```
public boolean isBCNF() {
  return functionalDependencies.stream()
    .filter(fd ->!fd.isTrivial())
    .allMatch(fd -> getKeys().stream().
      anyMatch(key -> fd.getLeftSide().containsAll(key)));
}
```

Tekintettel a 4.12 képletekre, egy bizonyos relációs séma normálformájának a megállapítását a `getNormalForm` metódussal valósítottuk meg.

```
public NormalForm getNormalForm() {
  return isBCNF() ? NormalForm.BCNF :
          isThirdNF() ? NormalForm.ThirdNF :
        isSecondNF() ? NormalForm.SecondNF :
          isFirstNF() ? NormalForm.FirstNF :
            NormalForm.Undefined;
}
```

5.4. Szintézis algoritmusának a megvalósítása

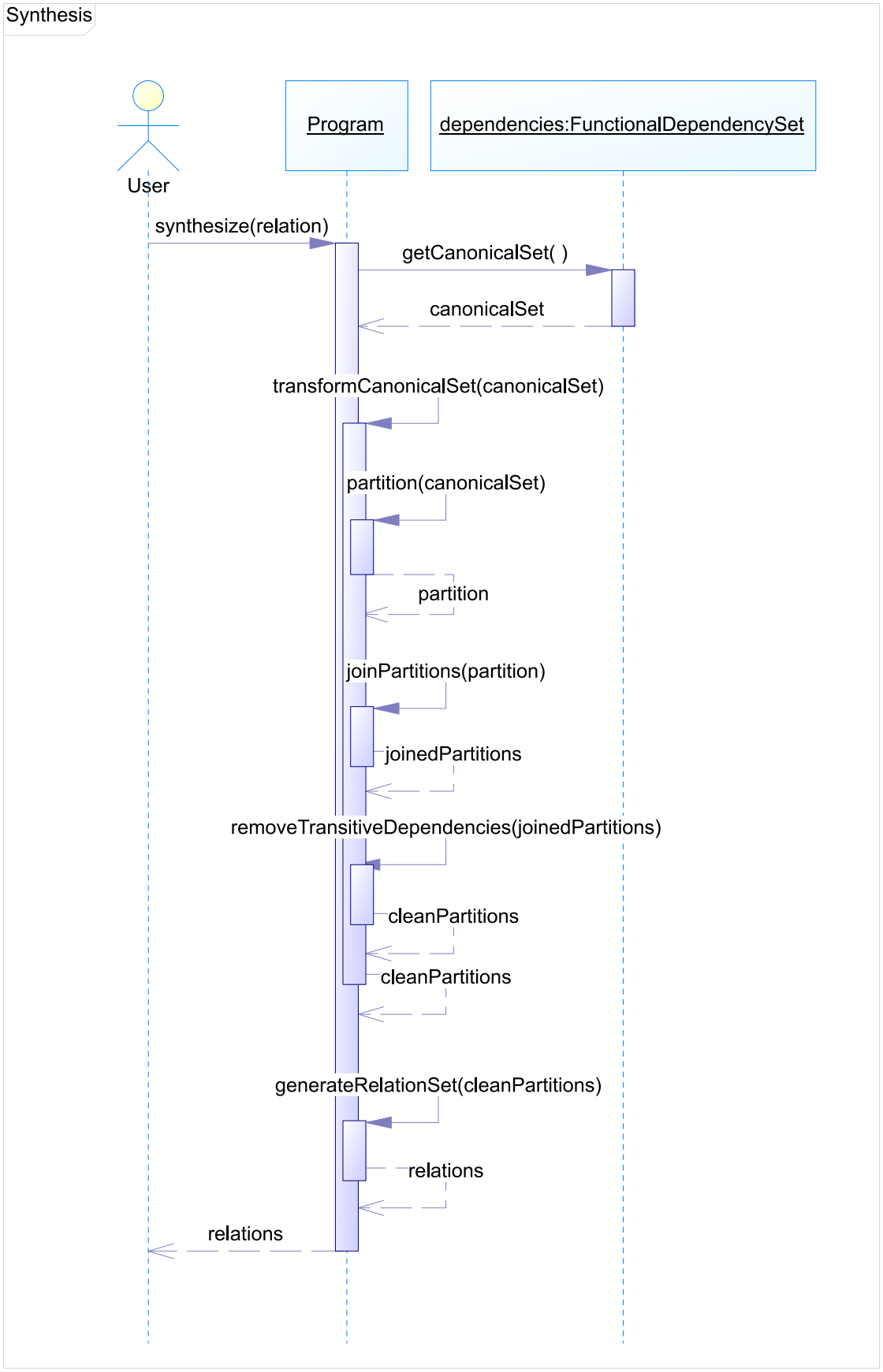
A szintézis algoritmusának számítógépes megvalósítását a(z) 5-3 ábra tükrözi. Felhasználói beavatkozás, ami befolyásolná az algoritmus kimenetelét, nem történik a lefuttatás alatt.

A szintézis algoritmusa alapján a legfontosabb lépés a minimális függéshalmaz kiszámolása. Ezt három lépésből tudjuk megtenni: A minimális függéshalmazban a függőségek jobboldali attribútumhalmazában csak egyetlen attribútum található (4.15 képlet), a függőségek teljes függőségek (4.16 képlet), valamint nincs olyan függőség, amelyik elhagyható (4.17 képlet).

1. A jobboldali függőségek egy atribútummá való redukálása:

```
FunctionalDependencySet reduced = new FunctionalDependencySet();
items.forEach(
  fd -> fd.getRightSide().forEach(
    label -> reduced.add(
      new FunctionalDependency(fd.getLeftSide(), label))
  )
);
```

2. A részleges függőségek teljes függésé való átalakítása



5-3. ábra. A szintézis szekvenciadiagramja

```
FunctionalDependencySet reducedS = new FunctionalDependencySet(
    canonicalSet);
reduced.forEach(
  fd -> fd.getLeftSide().getPowerSetSorted().stream()
  .filter(subset -> !subset.isEmpty() && !subset.equals(fd.
   getLeftSide()))
  .map(subset -> new FunctionalDependency(subset, fd.getRightSide())
   )
  .filter(partial -> partial.isLogicalConsequenceOf(reducedS))
  .limit(1)
  .forEach(partial -> {
    reducedS.add(partial);
    reducedS.remove(fd);
  })
);
```

3. Elhagyható függőségek törlése

```
FunctionalDependencySet setToRemove = new FunctionalDependencySet();
reducedS.stream()
  .filter(fd ->
    fd.isLogicalConsequenceOf(
      reducedS.difference(fd).difference(setToRemove)))
  .forEach(setToRemove::add);
reducedCanonicalSet.removeAll(setToRemove);
```

A partíciók kialakításához először egy `HashMap` kollekcióba rendeztük a függőségeket, a baloldali attribútumhalmazaitól függően. Ezután ezeknek a baloldali attribútumhalmazoknak kiszámoltuk a zártját, és amennyiben azok megegyeztek, összevontuk ezeket a partíciókat. Ezek az algoritmusok viszonylag egyszerűek, ezért nem kerülnek bemutatásra ebben a dolgozatban.

Ezután következik az összevonás által keletkezett potenciálisan tranzitív függőségek eltávolítása a partíciókból. Ennek kiszámolása érdekében a következőképpen létrehozzuk a J függőséghalmazt (4.23 képlet).

```
FunctionalDependencySet jay = new FunctionalDependencySet();
for(Partition partition: partitions) {
  FunctionalDependencySet desc = partition.descartesProductNonTrivial();
  FunctionalDependencySet descReduced = descartes.getReducedRight();
  jay.addAll(descReduced);
  partition.removeAll(descReduced);
}
```

A `descartesProductNonTrivial` metódus kiszámolja az összevont partícióknak az ún. permutációs (nem triviális) függőséghalmazát (4.24 képlet). Azaz, ha A, B és C kulccsal rendelkező partíciókat vontunk össze, akkor ez a metódus eredményként $A \rightarrow B$, $B \rightarrow A$, $A \rightarrow C$, $C \rightarrow A$, $B \rightarrow C$ és $C \rightarrow B$ függőségeket eredményezi. Majd ezeken a függőségeken

jobb oldali egyszerűsítést végzünk annak érdekében, hogy a jobboldali attribútumhalmaz csak egy attribútumot tartalmazzon.

A tranzitív függőségek eltávolításához befejező lépésként megalkotjuk az M függőséghalmazt (4.25 képlet).

```
FunctionalDependencySet em = new FunctionalDependencySet();
em.addAll(jay);
partitions.forEach(p -> em.addAll(p.getValues()));

partitions.forEach(partition -> {
  FunctionalDependencySet functionalDependenciesCopy =
    new FunctionalDependencySet(partition.getValues());
    functionalDependenciesCopy.forEach(fd -> {
    public ChannelService(ChannelRepository repository,
   OrganizationService organizationService, FilesystemUtil fsUtil) {
        this.repository = repository;
        this.organizationService = organizationService;
        this.fsUtil = fsUtil;
    }
      if(fd.isLogicalConsequenceOf(em.difference(fd))) {
        partition.remove(fd);  // fd is transitive dependency
      }
    });
});

partitions.forEach(partition -> partition.addAll(jay));
```

A relációs séma halmaza a szintézis algoritmusa alapján megfelel a létrejött partícióknak, amit így alkotunk meg:

```
RelationSet relations = new RelationSet("S");
int n = 1;
for(Partition partition : partitions) {
  LabelSet labels = partition.getLabels();
  Relation relation =
    new Relation("S", n++, labels, partition.getValues());
  relations.add(relation);
}
```

A veszteségmentes sémafelbontást érdekében a kezdetleges relációs séma egyik kulcsát tartalmazó relációs sémát hozunk létre, amennyiben olyan relációs séma nem képezi az eddig létrejötteket:

```
if(!relations.hasKey(initialRelation.getKeys())) {
  Relation lossless =
    new Relation("S", n,
      initialRelation.getKeys().stream().findAny().get(),
      new FunctionalDependencySet());
  relations.add(lossless);
}
```

5.5. Dekompozíció algoritmusának a megvalósítása

A dekompozíció első lépése a megfelelő függőség kiválasztása. Ezt a függőséget háromféle kritérium alapján tudjuk kiválasztani. A következő metódusok azt vizsgálják, hogy egy adott függőség megfelel-e a P1 (4.26), P2 (4.27) illetve a P3 (4.28) kritériumnak.

```
public boolean satisfiesP1(FunctionalDependency fd) {
  return !fd.isTrivial() &&
    keys.stream().noneMatch(k -> fd.getLeftSide().containsAll(k)) &&
    functionalDependencies.equivalent(
      functionalDependencies.projection(fd.getLeftSide()
        .union(labels.difference(fd.getRightSide())))
    .union(functionalDependencies.projection(fd.getLeftSide()
        .union(fd.getRightSide()))));
}
```

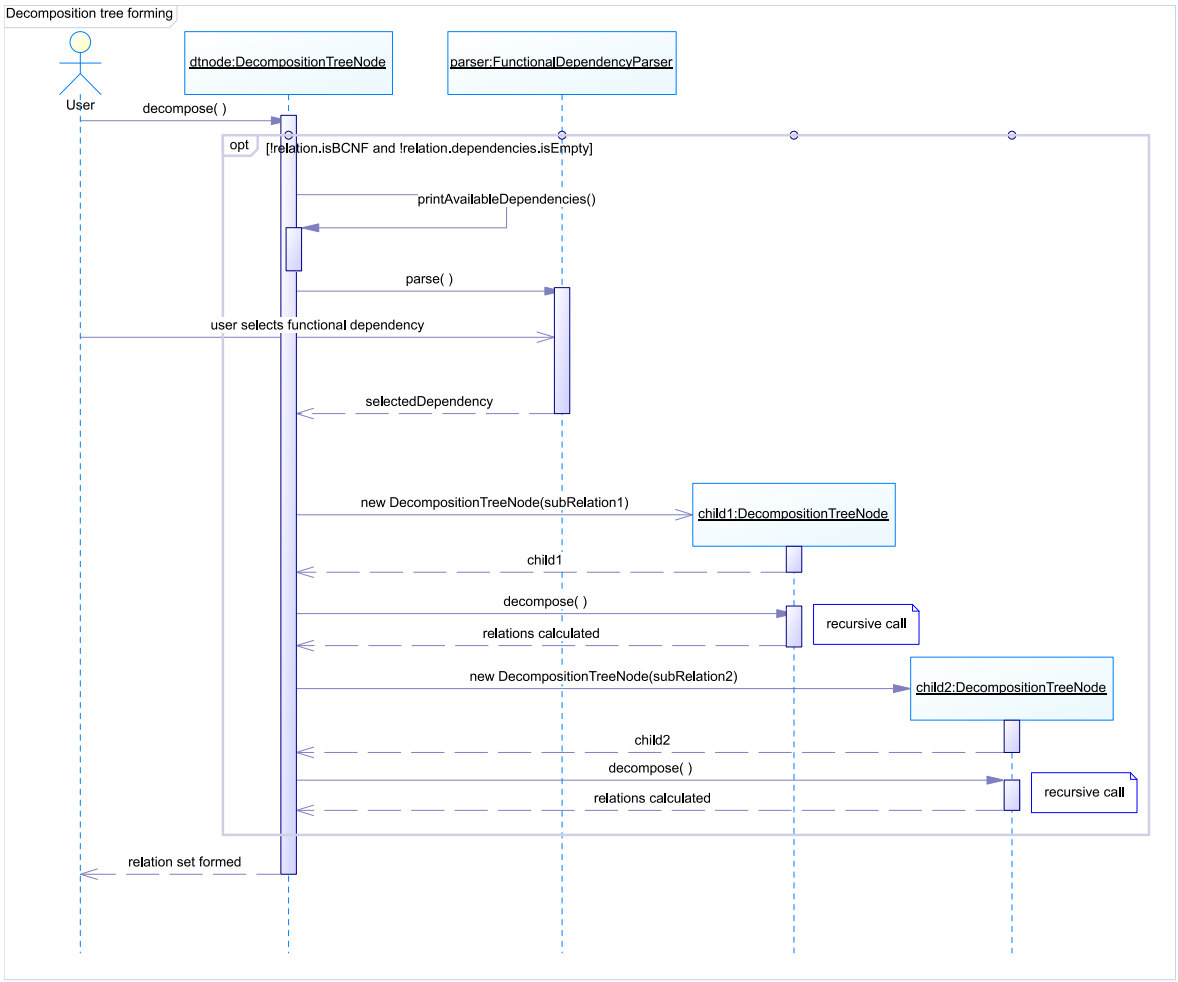
```
public boolean satisfiesP2(FunctionalDependency fd) {
  return !fd.isTrivial() &&
    !fd.getLeftSide().union(fd.getRightSide()).containsAll(labels) &&
    functionalDependencies.equivalent(
      functionalDependencies.projection(fd.getLeftSide()
        .union(labels.difference(fd.getRightSide())))
      .union(functionalDependencies.projection(fd.getLeftSide()
        .union(fd.getRightSide()))));
}
```

```
public boolean satisfiesP3(FunctionalDependency fd) {
  return !fd.isTrivial() && keys.stream().noneMatch(k ->
    fd.getLeftSide().containsAll(k));
}
```

A szintézis algoritmussal ellentétben, a dekompozíció algoritmusánál a felhasználó is beleszólhat az algoritmus lefolyásába. Ezt oly módon teszi, hogy a felkínált függőségekből (melyek lehetőleg minél nagyobb kritériumi szintet teljesítenek) választ egyet. A(z) 5-4 ábra bemutatja, hogy minderre a `decompose` metódus elején kerül sor – a `user selects functional dependency` üzenettel. Az algoritmus interaktív mivolta úgy valósul meg, hogy a program a standard kimenetre nyomtatja a választható függőségeket, a felhasználó pedig a billentyűzet segítségével kiválasztja a neki megfelelő függőséget. Az algoritmus automatikus változata ezt a választás lehetőségét átugorja, és találomra választ egy függőséget a felkínáltak közül.

Mivel a dekompozíció lépéseiben az algoritmus az aktuális relációs sémát két relációs sémára válassza szét (4.29 képlet), innen ered az ötlet, hogy a dekompozíció megvalósításához elég létrehozni egy ún. dekompozíciós fát. Ennek a fának a csomópontjai relációs sémák lesznek, a levelei pedig a dekompozíció eredményeként létrejött relációs sémahalmazt alkotják.

A dekompozíciós fa létrejöttéhez a `decompose` metódus rekurzív előhívása szükséges a



5-4. ábra. A dekompozíciós fa formálásának a szekvenciadiagramja

relációs séma dekomponált gyermekcsomópontjain. Ez a folyamat addig folytatódik, amíg nem kapunk olyan relációs sémákat, amelyek teljesítik a *BCNF* normálformát.

Miután létrehoztuk az ún. dekompozíciós fát, ki kell értékelni azt. Ezt szintén rekurzív módszerekkel oldottuk meg (5-5 ábra). A fa leveleinek a begyűjtése után ki kell vizsgálni, hogy az azonos kulcsokkal rendelkező relációs sémákat összevonjuk, majd leellenőrizzük a veszteségmentes sémafelbontás feltételeit. Ezt hasonlóképp végezzük el, mint a szintézis algoritmusa során.

5.6. Egyéb magvalósítási részletek

A dolgozatban felvázolt célkitűzések között szerepel az a követelmény, hogy a feladatsorok könnyen megadhatóak és cserélhetőek legyenek. Ennek a követelménynek tettünk eleget, hogy szöveges úton engedélyeztük a program bemeneti feladatsorát. A felhasználó egy *JSON* típusú fájlként tudja megadni a feladatsorát, majd lefuttatáskor a megfelelő fájlt tudja bemenetként megadni.

Példaként szolgálva, ha egy N nevezetű relációnak az attribútumhalmaza $\{A, B, C, D, E\}$, függőséghalmaza pedig $\{AB \rightarrow CE, C \rightarrow D\}$, akkor a bemeneti feladatsor fájlja a következőképp néz ki:

```
{
  "name": "N",
  "labels": ["A", "B", "C", "D", "E"],
  "functionalDependencies": [
    {"left": ["A", "B"], "right": ["C", "E"]},
    {"left": ["C"], "right": ["D"]}
  ]
}
```

A forráskód felépítéséhez és becsomagolásához a *Maven* eszköz `package` parancsa szükséges:

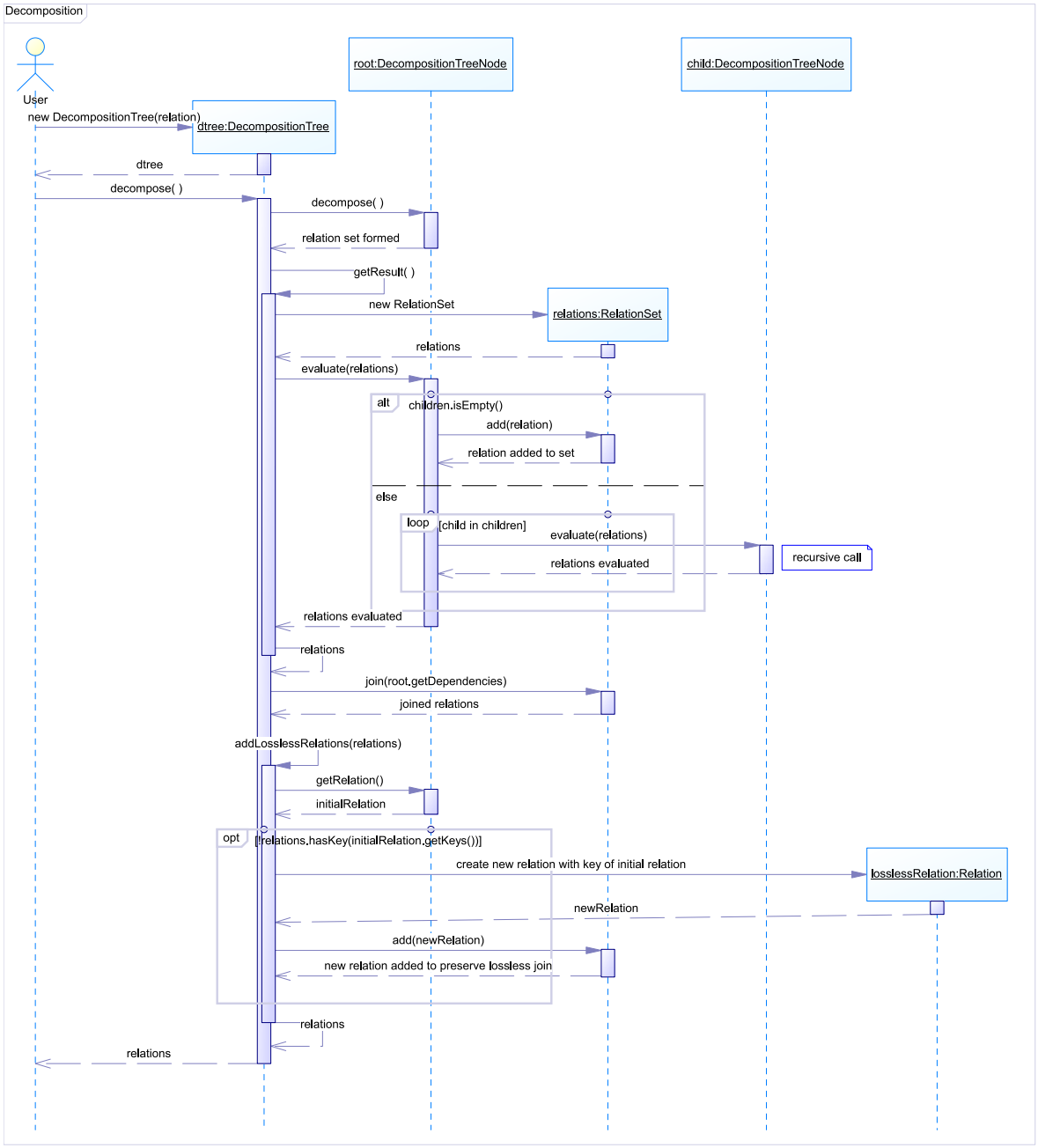
```
mvn package
```

A sikeres csomagolás után a build mappában fog szerepelni a *JAR* csomag, melyet a következőképp tudunk lefuttatni:

```
java -jar rel-norm-1.0-SNAPSHOT.jar <task> <method>
```

A `<task>` megjelölés helyett a feladatsor fájljának az útvonalát kell megadni, a `<method>` megjelölés helyett meg egyikét a következőnek:

- `DECOMPOSITION` – a dekompozíció algoritmus lefuttatásához,



5-5. ábra. A dekompozíciós fa kiértékelésének a szekvenciadiagramja

- `DECOMPOSITION_I` -– a dekompozíció interaktív algoritmusának lefuttatásához és
- `SYNTHESIS` – a szintézis algoritmusának a lefuttatásához.

5.7. Szoftvertesztek megvalósítása

A *Java* programnyelvnek rendeltetésszerű tesztelési munkakeretét használtuk a dolgozatban. A munkakeret neve *JUnit*, ami unit-tesztek írását és lefuttatását teszi lehetővé. A dolgozat 4. fejezetében felsorolt relációs műveletek és algoritmusok mindegyike szerepel a teszt esetekben. A tesztek érvényességét feladatgyűjteményből (Kordić és tsai. 2018) származtatott feladatok és megoldásaik szavatolják. A tesztek elindításához a következő parancsot kell végrehajtani:

```
mvn test
```

A kód lefedettség kiszámolása érdekében külső szoftverelemző eszközt használunk. A *SonarCloud* online kódbázis elemző szoftver, amivel git alapú kódbázisokat tudunk megfigyelni. A *SonarCloud* képes jelenteni programhibákat, biztonsági réseket, nem tiszta kód jellemzőit, kód többszöröződést stb. Mindezek mellett még kód lefedettségi szintet is mér, habár ezt külső tesztjelentésből állítja össze. Ahhoz, hogy a *SonarCloud* külső tesztjelentéseket tudjon olvasni, integrálni kell a *JaCoCo* bővítményt a *Maven* projektbe. Ez a bővítmény jelentéseket készít a lefuttatott tesztekről.

6. fejezet

Eredmények

Az előző fejezetben leírt algoritmusok megvalósításának a helyességét úgy tudjuk leellenőrizni, ha szoftvertesztekkel támasztjuk alá a működésüket. Ezeknek a teszteknek az eredményei a 6-1 táblázatban szerepelnek. Valamennyi algoritmus metódusához rendeltünk legalább 2 tesztet, hogy megbizonyosodjunk arról, hogy a tesztek nem csak véletlenszerűen sikerültek. A teszt eseteket feladatlapból (Kordić és tsai. 2018) szerzett feladatokra alapoztuk, melyeknek ismertek az eredményei. Ezek az ismert eredmények kerültek összehasonlításra az algoritmus által kapott eredményekkel. Amennyiben az eredmények nem egyeztek meg, azokat sikertelen teszteknek vettük; ha hiba keletkezett tesztelés során, azok a hibák száma oszlopban vannak feltüntetve. Ha bármiféle okból kifolyólag nem lehetett lefuttatni a tesztet, az az átugrott tesztek számát növelte. A tesztek mellett feltüntetett eltelt idő az egyes tesztcsoportok lefutásának az idejét jegyzi.

A *SonarCloud* szoftverelemző eszköz jelentésének egy részét a 6-2 táblázat jeleníti meg. Ez a jelentés magában foglal különböző statisztikai adatokat a teljes kódbázisról, amit a *SonarCloud* bizonyos belső algoritmusok és kritériumok alapján számol.

Teszt neve	Tesztek száma	Sikertelen tesztek száma	Hibák száma	Átugrott tesztek száma	Eltelt idő [ms]
LogicalConsequence	5	0	0	0	2
AttributeSetClosure	6	0	0	0	3
Equivalence	2	0	0	0	1
Projection	3	0	0	0	1
Keys	3	0	0	0	5
RelationParser	2	0	0	0	66
2NF	5	0	0	0	25
3NF	2	0	0	0	1
NormalForm	4	0	0	0	5
CanonicalSet	4	0	0	0	39
Decomposition	4	0	0	0	97
Synthesis	6	0	0	0	108
TOTAL	46	0	0	0	353

6-1. táblázat. Szoftvertesztelés eredménye

Jellemző	Érték
Összes kódsor	1161
Osztályok száma	21
Kommentek aránya	1.4%
Code Smell	30
Biztonsági rések	0
Kódtöbbszöröződés	0%
Hibák	0
Teszt lefedettség	76.5%

6-2. táblázat. SonarCloud elemzésének az eredményei

7. fejezet

Tárgyalás

Az előző fejezet szoftvertesztelési eredményei alapján kijelenthető, hogy a *RelNorm* eszköz sikeresen el tudja végezni a relációs adatbázisok normalizációját a szintézis és a dekompozíció algoritmusát felhasználva. Ezek az algoritmusok relatív több időt igényelnek, mint a kisebb összetettségű algoritmusok, és ez a teszteredményeken is meglátszik. A `RelationParser` teszt esetek ugornak ki a sorból a hosszabb futtatási idővel. Ezekben az esetekben fájl beolvasása történik, ezzel magyarázható a hosszabb futtatási idő. A dekompozíció algoritmusánál nem sikerült tesztelni az interaktív függőség megadás módszerét, ezért az interaktív módszer azon pontján, ahol felhasználói bemenet szükséges, automatikusan kiválasztja a program a felkínált függőségek egyikét.

A *SonarCloud* jelentése nem okozott kiugró értékeket. Megemlíthető azonban a *Code Smell* jellemző, ami a nem tiszta kód jellemzőit számlálja. Ez az érték 30 lett, ami egy relatív magas szám. Ennek a 30 észrevételnek túlnyomó többsége a felhasználóval való interakciót jelezte, pontosabban a standard bemenet és kimenet használatát kifogásolta. Mivel a *RelNorm* egy konzol applikáció, ezért egyelőre nincs más módja a felhasználói bemenet megváltoztatására.

A teszt lefedettség 76.5% lett, ami szintén egy jó eredmény. Betekintve az elemzés részleteibe, a lefedettség azért nem lett nagyobb, mert nem került tesztelésre valamennyi konstruktőr, get/set metódus, beépített metódus, valamint a standard bemenettel és kimenettel foglalkozó metódusok sem.

8. fejezet

Összefoglalás

Ebben a dolgozatban bemutatásra került a *RelNorm* nevezetű szoftver, amely relációs adatbázisok normalizálásának a problémáját próbálja megoldani. A *RelNorm* eszközt elsősorban oktatásban ajánlott alkalmazni, normalizálási feladatsorok összeállításánál és a megoldott feladatok leellenőrzésénél. A dolgozat bemutatja a hasonló normalizálási szoftvereket, majd a legalapvetőbb relációs fogalmakra és algoritmusokra tér rá. A *RelNorm* fejlesztési fázisai a(z) 5. fejezetben kerültek be a dolgozatba, ahol kódrészletekkel valamint osztály- és szekvenciadiagramokkal prezentáltuk a szoftver megvalósítását. A *RelNorm* szoftvertesztelésnek vetettük alá, ezek eredményei és kiértékelése a dolgozat utolsó fejezetében történt.

A bemutatott eredmények és azok elemzésének fényében következtethető, hogy a *RelNorm* teljesítette a dolgozatban elé támasztott követelményeket. A gyakorlatban a 2021–2022-es tanévben már helytállt szoftver kielégítette azokat az igényeket, melyeket tanársegédi feladatokként szántunk a szoftverhez.

Jövőbeli terveink között szerepel egy grafikus felhasználói felület kifejlesztése, ami akár web applikációs formát is ölthet: így a *RelNorm* szélesebb rétegekhez is eljuthat. Amennyiben igény van rá, akkor további funkciókkal is bővíthető a program, ami egy tanulási keretet is adhat a hallgatóknak.

Irodalomjegyzék

2019 Database Trends (2019). ScaleGrid.io. URL: https://scalegrid.io/blog/2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use/ (elérés dátuma 2022. 03. 30.).

Adatbázisok 2 (2021). Műszaki Tudományok Kara, Újvidéki Egyetem. URL: http://ftn.uns.ac.rs/571482529/databases-2 (elérés dátuma 2022. 03. 30.).

Bahmani, Amir, Mahmoud Naghibzadeh és Behnam Bahmani (2008). "Automatic database normalization and primary key generation". *2008 Canadian Conference on Electrical and Computer Engineering* (Niagara Falls, Kanada). DOI: 10.1109/CCECE.2008.4564486.

Čeliković, Milan és tsai., szerk. (2021). *Adatbázisok 2 - labor gyakorlatok gyűjtemény*. Újvidék: Műszaki Tudományok Kara.

Gajdos, Sándor (2019). *Adatbázisok*. Budapest: Műegyetemi Kiadó. URL: https://db.bme.hu/~gajdos/Adatbazisok2019.pdf (elérés dátuma 2022. 01. 01.).

Georgiev, Nikolay (2008). *A Web-Based Environment for Learning Normalization of Relational Database Schemata*. research. Umeå University.

Hongbo, Du és Laurent Wery (1999). "Micro: A normalization tool for relational database designers". *Journal of Network and Computer Applications* 22 (4), 215–232. old. DOI: 10.1006/jnca.1999.0096.

Khorikov, Vladimir (2020). *Unit Testing: Principles, Practices and Patterns*. New York: Manning Publications Co.

Kordić, Slavica és tsai. (2018). *Baze podataka - zbirka zadataka*. Újvidék: Műszaki Tudományok Kara, Újvidéki Egyetem.

Loukides, Mike (2022). *Technology Trends for 2022*. technical report. USA: O'Reilly Media, Inc. URL: https://www.oreilly.com/radar/technology-trends-for-2022/ (elérés dátuma 2022. 03. 30.).

Mitrovic, Antonija (2002). "NORMIT: a Web-enabled tutor for database normalization". *International Conference on Computers in Education* (Auckland, Új-Zéland). DOI: 10.1109/CIE.2002.1186210.

Mogin, Pavle és Ivan Luković (1996). *Principi baza podataka*. Újvidék: Műszaki Tudományok Kara, Újvidéki Egyetem.

Mogin, Pavle, Ivan Luković és Miro Govedarica (2004). *Principi projektovanja baza podataka*. Újvidék: Műszaki Tudományok Kara, Újvidéki Egyetem.

Obtaining a Power Set of a Set in Java (2020). Baeldung. URL: https://www.baeldung.com/java-power-set-of-a-set#3-binary-representation (elérés dátuma 2021. 11. 01.).

Ramel, David (2015). *Relational Databases Still Reign in Enterprises, Survey Says*. URL: https://esj.com/articles/2015/04/23/database-survey.aspx (elérés dátuma 2022. 03. 30.).

Tárgymutató