Eötvös Loránd Tudományegyetem

Informatikai Kar

Algoritmusok és Alkalmazásaik Tanszék

Membránrendszerek vizualizációja és szimulálása

Témavezető:
Kolonits Gábor
egyetemi tanársegéd

Szerző:
Buzga Levente
programtervező informatikus BSc

Budapest, 2022

A membránrendszer vagy más néven P-rendszer olyan biológiailag inspirált számítási modell, amely képes a biokémiai folyamatok törvényszerűségeinek modellezésére. A rendszer több belső térből, úgynevezett régióból áll, amelyek hierarchikus membránstruktúrát alkotnak. Ez lehetővé teszi a sejten belüli és a sejtek közötti információáramlás formális modellezését.

Az egyes régiókban különböző nyersanyagok fordulhatnak elő, amelyek egymással reakcióba léphetnek. A reakció következtében ezek a nyersanyagok átalakulhatnak, régiók között átvándorolhatnak, illetve membránok is feloldódhatnak. Ezeket a reakciókat evolúciós szabályoknak hívjuk. A felépítésükből adódóan a membránrendszerek nagy mértékű párhuzamosításra is alkalmasak. A membránrendszer számítása közben kialakuló állapotát egy konfigurációja írja le, amelyet minden membrán esetén objektumokból álló multihalmaz reprezentál.

A számítás eredményét többféleképpen is definiálhatjuk, alapértelmezetten egy kitüntetett membránban a megálláskor jelen lévő objektumok mennyiségét jelenti.

Szakdolgozatom célja egy olyan szoftver elkészítése, amely képes különböző típusú membránrendszerek grafikus megjelenítésére, betöltésére és exportálására, illetve egy megadott szabályrendszer mentén történő számításaik szimulálására. A felhasználónak lehetősége van a membránstruktúra vizuális megkonstruálására, illetve egy megadott formátum segítségével szabályok rögzítésére. A számítás végrehajtásánál támogatott a lépésenkénti futtatás.

Tartalomjegyzék

1. fejezet

Bevezetés

1.1. A számításelmélet rövid története

A számításelmélet az 1930-as évek közepén Alan Turing és Alonzo Church munkássága révén vált külön tudományággá, amelynek központjában a számítások és a számítógépek formális modellezése és elemzése áll [**savage2008modelsofcomp**]. A terület azóta is kulcsfontosságú szerepet kap az informatikában. Az előbb említett matematikai logikával foglalkozó tudósok és társaik arra a kérdésre szerettek volna választ adni, hogy mit jelent maga a számítás? Ezzel az algoritmus matematikai modelljének megalkotását tűzték ki célul. Természetesen a számításelmélet létrejötte előtt is születtek algoritmusok, de azok nem törekedtek formalitásra, hanem egyszerűen csak rögzítették a végrehajtandó utasítások sorrendjét. A kérdés megválaszolása érdekében Turing bevezette a Turing gépet, Church pedig a lambda kalkulust. Ezen eszközök segítségével elkezdték vizsgálni a kiszámíthatóságot. Azt fedezték fel, hogy vannak olyan számítási feladatok, amelyekre nem adható algoritmikusan válasz, illetve, hogy vannak olyan univerzális számítógépek, amelyek minden másik számításra alkalmas gépet szimulálni tudnak. Ez vezetett a Church-Turing tézis megfogalmazásához, amely szerint minden olyan matematikailag formalizálható probléma, ami megoldható valamilyen algoritmussal, az Turing géppel is megoldható. Később kiderült, hogy léteznek a Turing géppel azonos számítási erejű modellek, viszont azóta sem találtak ennél nagyobb számítási erővel rendelkező absztrakt modellt. Ezek szerint ha elfogadjuk a tézist, akkor a Turing gép tekinthető az algoritmus formális modelljének.

A tudományág fejlődését ösztönözte és elősegítette a digitális számítógépek meg-

jelenése. A tézis után pár évvel Konrad Zuse elkészítette az első általános célú programozható elektromechanikus számítógépet, amelyet elektromágneses relékkel működtetett. 1945-ben készült el az első teljesen elektromos számítógép, az ENIAC, majd Neumann János 1947-ben megfogalmazta a mai számítógép működésének alapjául szolgáló Neumann-elveket. Azóta a számítógépek számítási kapacitása rohamos ütemben növekedik, ám az akkor fontosnak vélt számítási elemzések és korlátok a mai napig fontos szempontot adnak algoritmusok megalkotásánál.

A mai számításelmélet nagyon változazos témák tárházával rendelkezik, magába foglalja az algoritmusok, adatszerkezetek, elosztott és párhuzamosított számítások, kvantumszámítások, biológiai számítások vizsgálatát, illetve nem utolsó sorban a bonyolultságelméletet és az információelméletet. Mindezekből látszik, hogy milyen szerteágazó tematikája és problémaköre van a számításelméletnek és hogy ezek az irányvonalak folyamatosan képesek fejlődni és gazdagodni.

1.2. Membránrendszerek

A 20. század második felében egyre nagyobb figyelmet kaptak az olyan számítások, amely a természetben vagy valamilyen természeti jelenséghez kapcsolódóan mennek végbe [**paun2002membrane**]. Ezen megfigyelések eredményeképpen megszületett a biológiai számítások területe, amelynek számos új módszert és algoritmust köszönhetünk. Ide tartoznak például a genetikus algoritmusok (általánosabban véve az evolúciós algoritmusok), illetve a neurális hálók. Ezen irányvonal kiterjesztésének tekinthetőek az 21.század legelejére kibontakozó területek mint a membránszámítások és a DNS számítások, amelyek közül az utóbbit a molekuláris biológia alapjaira modelleztek.

A membránszámítások legnagyobb inspirációját a biológiai sejtek jelentik, amelyek az élet legelemibb építőelemei. A sejtek működésében és struktúrájában pedig elengedhetetlenek a membránok, amelyek a sejten belüli rétegződést határozzák meg. Egy sejtet a környezetétől a legkülső membránja (úgynevezett skin) határol el, a rétegződésért pedig a belső membránok felelősek. Egy-egy ilyen membrán pedig egy saját belső teret, *régiót* határol el. Ezen régiókban pedig kémiai reakciók mehetnek végbe, amely hatására különböző molekulák átvándorolhatnak a membránokon keresztül új régiókba. Ezzel a térbeli struktúráltsággal egészítik ki a membránszámítások a DNS számításokat.A membránszámítás motivációja, hogy egy olyan modellt

tudjuk konstruálni, amely képes a sejtek közötti és sejten belüli molekulák által közvetített információáramlás mechanizmusának leírására, szemléltetésére.

Ezen célkitűzéssel vezette be Gheorge Paun a membránrendszereket 2000-ben. Később az ilyen modelleket P-rendszereknek is elnevezték az ő tiszteletére. Ahhoz, hogy a biokémiai folyamatokat modellezni tudja, figyelembe vette ezen folyamatok törvényszerűségeit. Nevezetesen, hogy az ilyen reakciók végbemeneteléhez kellő számban szükség van a reakcióhoz szükséges kezdeti molekulákra és ha ezek rendelkezésre állnak, akkor a reakció biztosan meg is fog történni. Emellett egyszerre több reakció (vagy akár egy reakció többször) is végbemehet, ha van hozzá(juk) elegendő nyersanyag. Speciális esetekben egy reakció hatására membránok feloldódhatnak, ilyenkor a benne rejlő molekulák és belső régiók az őt körülvevő régióba kerülnek, illetve az is előfordulhat, hogy a reakciók között valamilyen prioritási sorrend is fennállhat; azaz az egyik reakció nem hajtódhat végre, amíg egy a másik reakciónak lehetősége van rá. Ezen reakciók úgynevezett *evolúciós lépésekbe* szerveződnek, amelyek során a régiók molekulái és a membránrendszer struktúrája is átalakulhat.

A számítás központi szereplői az előbbi tulajdonságokkal rendelkező reakciók, amelyek a membránrendszerekben szabályoknak nevezünk, a résztvevő molekulákat pedig objektumoknak. Ezek a szabályok régiókhoz vannak rendelve. A szabályok alkalmazására a maximális párhuzamosság elve érvényes, azaz ha a jelenleg alkalmazott szabályok után még alkalmazható lenne egy újabb vagy egy eddig már alkalmazott szabály, akkor az a szabály végre is lesz hajtva az adott evolúciós lépésben. Ennek megfelelően a membránrendszer evolúciós lépések sorozatán megy keresztül, egészen addig, amíg már nincsen egyetlen alkalmazható szabály sem. Ilyenkor a számítás eredményét alapértelmezetten a környezetbe kijutó objektumok számának tekinjük, de ez egyes rendszerekben másképpen is megadható (például egy kijelölt kimeneti régió objektumainak számával).

Érdeklődésemet a téma iránt az SAT eldöntési probléma aktív P-rendszerekkel (amely típust a dolgozatban nem részletezem) való lineáris időben történő megoldása keltette fel. Ez elsőre egy hihetetlen eredmény, hiszen ez az egyik legismertebb NP-teljes probléma. Az ellentmondást a membránrendszerek masszív párhuzamosítási képességével lehet megmagyarázni, amelynek segítségével a változók és a klózok számának függvényében igaz, hogy lineáris időben leáll, viszont ezt a membránrendszert szimuláló Turing gép már nem tudna minden bemeneti példányra lineáris időben leállni. Ezután kezdtem el foglalkozni membránrendszerekkel és úgy éreztem, hogy egy

szimulációs szoftver hasznosnak bizonyulna kutatási és oktatási célokra egyaránt.

1.3. Megvalósítandó feladat

A dolgozatom célja egy olyan szoftver elkészítése, amely lehetővé teszi a felhasználó számára különböző felépítésű membránrendszerek megkonstruálását, grafikus módon való megjelenítését, illetve számításainak végrehajtását. A támogatott számítási modellek magukba foglalják a membránrendszerek alapmodelljét (amelyben egyes régiók feloldódhatnak, illetve a szabályok között lehet prioritási sorrend) és a szimport-antiport rendszereket. A membránrendszer létrehozását a felhasználó egy (megfelelő formátumú) karakterlánc megadásával kezdeményezheti, ami egyszerűen és szemléletesen írja le a membránrendszer struktúráját és opcionálisan tartalmazhatja régiokra bontva a benne található objektumokat. A megalkotott membránrendszer ilyenkor még nem rendelkezik evolúciós szabályokkal, azokat egy külön menü segítségével tudja beállítani. Ilyenkor a kiválasztott régió objektumai is szerkeszthetőek.

A szoftver különösen hasznos tud lenni oktatási célokra, hiszen biztosítja a szabályok elemzésének lehetőségét a lépésenkénti futtatás funkcióval, illetve a teljes szimuláció során megkapott eredmények segítségével tanulmányozható, hogy a létrehozott modell a kívánt nyelvet (ebben az esetben a természetes számok valamely részhalmazát) generálja-e, ezzel segítve a helyes és helytelen működés (nem teljeskörű) kiszűrését. A program biztosít a felhasználó számára egy használati útmutatót, melyben megtalálja a legfontosabb információkat ahhoz, hogy korábbi ismeretek nélkül is képes legyen egy működő membránrendszert megalkotni és számítását szimulálni.

Lehetőség van a membránrendszer aktuális állapotának lementésére, illetve egy korábbi mentés betöltésére. A membránrendszer *JSON* formátumban kerül mentésre, így azt a felhasználónak lehetősége nyílik arra, hogy a fájlban is módosítson, majd a módosított állapotot töltse be. Ezáltal a szoftver biztosítani tudja azt, hogy egy bonyolult vagy gyakran vizsgált membránrendszert nem szükségszerű a kezdetektől megalkotni, elegendő egyszer a kívánt konfigurációját előállítani, majd később betölteni a már elmentett verziót.

A feladatot *Python* programozási nyelven, a *Qt* keretrendszer felhasználásával valósítottam meg.

2. fejezet

Felhasználói dokumentáció

2.1. Az alkalmazás célja

Az alkalmazás célja, hogy segítségével a felhasználó membránrendszereket hozzon létre majd szimulálja számításaikat. A membránrendszer egy olyan biológiailag inspirált számítási modell, amely az eukarióta sejtek működését és felépítését követve evolúciós lépéseken keresztül történő információáramlást ír le membránok között. Minden membrán által körbezárt *ún.* régió tartalmaz evolúciós szabályokat, amelyek nem változnak a membránrendszer működése közben. Az információt a rendszerben a régiókban található molekulák, *ún.* objektumok hordozzák. Egy szabály csak akkor tud végbemenni, ha rendelkezésre állnak a szükséges objektumok kellő számban. Ilyen helyzetekben a szabályoknak végre is kell hajtódnia, tehát nem fordulhat elő, hogy minden objektum hozzáférhető, de nem kerül a szabály alkalmazásra. Egy evolúciós lépésben a maximális párhuzamosság elve érvényesül, azaz a szabályok véletszerűen kerülnek kiválasztása, egészen addig, amíg van alkalmazható szabály. Az egyik szimulálható típusú rendszerben megadhatóak olyan speciális szabályok, amelyek alkalmazásának hatására egy membrán feloldódhat, ilyenkor teljes tartalma (benne levő objektumok és régiók) az őt körbevevő régióba kerül. A szabályok között prioritási sorrend is felállítható. A számítás legfontosabb tulajdonsága annak kimenete, amely általában a legkülső régión kívülre (azaz a környezetbe) kijutó objektumok számát jelenti.

2.2. Hardver és szoftver követelmények

A szoftver futtatásához Linux környezetre van szükség, amely támogatja az `ELF` formátumú bináris állományok értelmezését. Ezen felül a futtatási környezetnek rendelkeznie kell *Python interpreterrel*, illetve a *Qt* keretrendszerhez való hozzáférés érdekében `PySide6` modullal. Az utóbbi könnyen megtehető shell környezetben a `pip install PySide6` paranccsal . A program teljes funkcionalitásának kihasználásához a felhasználó számítógépének a bemeneti perifériák közül egérrel és billentyűzettel kell rendelkeznie. A szoftver hardverigénye nem igényel részletesebb specifikációt.

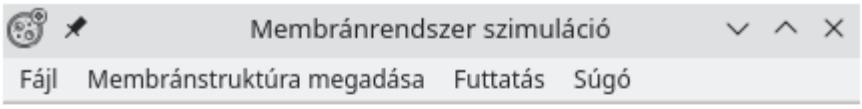
2.3. Futtatás

Mivel a program futtatható állományban kerül a felhasználóhoz, ezért annak az indításhoz elegendő megnyitni a fájlt tartalmazó mappát, majd duplán kattintani a fájlt reprezentáló ikonra. Ugyanez parancssori környezetben is elvégezhető, ilyenkor a terminálban a megfelelő mappába való elnavigálás után a `./MembraneSimulator` parancs megadásával futtatható a program.

2.4. Grafikus felhasználói felület

A felhasználó az alkalmazással a grafikus felhasználói felületen keresztül tud kommunikálni, amely a főablakot és az igény szerint megjelenő dialógusablakokat foglalja magába.

2.4.1. Főablak



2.1. ábra. A főablak az alkalmazás megnyitásakor

A főablak az alkalmazás megnyitásakor még tartalmaz egyetlen grafikus elemet sem, viszont a menüsorban található menüpontok segítségével könnyedén változtatni lehet ezen. Ha a felhasználónak nincs korábbi tapasztala a program használatával, akkor érdemes a *Súgó* menüpont kiválasztásával kezdenie, amelyről részletesen szó esik a 2.5 fejezetben.

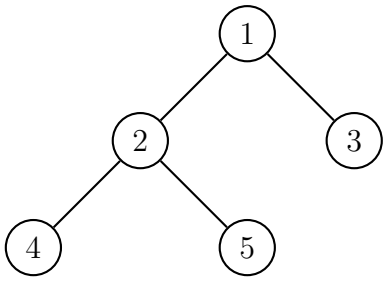
2.4.2. Dialógusablakok

A legfontosabb interakció a felhasználó és a szoftver között a dialógisablakokon keresztül történik.

Membránstruktúra megadása

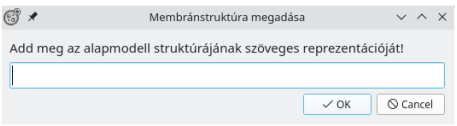
Egy membránrendszer megalkotásának kezdeti módja a struktúrájának megadásával kezdődik. Mivel egy membránrendszerben a membránok hierarchikusan helyezkednek el, ezért a teljes rétegződést nagyon jól lehet ábrázolni fa alakban, ahol mindenkinek a szülő csúcsa az őt legszűkebben tartalmazó régió.

Ezzel egyenértékű az a felírás, amikor egyetlen karakterláncban fejezzük ki ugyanezt, azáltal, hogy egy régiót megfeleltetünk egy nyitó-csukó zárójelpárral. Ilyenkor a két zárójel között elhelyezett objektumok jelentik a régió tartalmát. Azonban nem

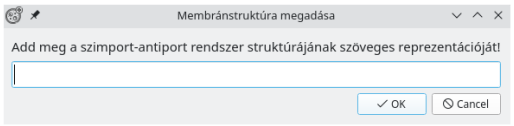


2.3. ábra. Membránrendszer hierarchikus felépítése[1]

csak objektumok, de más régiók is helyet kaphatnak, ezzel kifejezve azt, hogy a már említett régió közvetlen gyerekét szeretnénk megadni. Az alkalmazásban a struktúra megadása a menüsorban található *Membránstruktúra megadása* gombbal kezdeményezhető.



(a) Dialógusablak alapmodell létrehozásához



(b) Dialógusablak szimport-antiport rendszer létrehozásához

2.2. ábra. Dialógusablakok membránrendszer létrehozásához

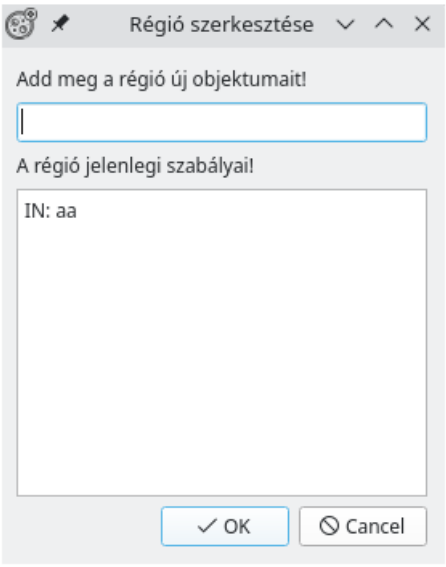
A 2.2 ábra a különböző típusú membránrendszerek struktúrájának megadásához használt dialógusablakot mutatja. A megadott karakterláncban a régiók kezdetét és végét jelző zárójelpárok, illetve a bennük előforduló objektumok szerepelhetnek. A helyes formátum feltétele, hogy a zárójelpárok karakterein és az szimport-antiport rendszereknél a kimeneti régió jelzésére használt speciális # karakteren kívül csak az angol ábécé kisbetűi szerepelhetnek a bemenetben, illetve annak meg kell felelnie a helyes zárójelezés szabályainak. Ezt azt jelenti, hogy nem lehetnek átfedések régiók között, azaz olyan karakterláncok, amikor egy külső régió hamarabb kerül lezárásra, mint bármelyik benne lévő régió. Ha ezen feltételeknek megfelel a felhasználó által megadott karakterlánc, akkor a főablak teljes területét lefedő vásznon megjelenik a létrehozott membránrendszer.

Ha a megadott karakterlánc nem a megfelelő formátumú, akkor a felhasználó figyelmeztetésére egy hibaüzenet jelenik meg a képernyőn.

[1]Az ábrán látható struktúra szöveges megfelelője a $[_1[_2[_4]_4[_5]_5]_2[_3]_3]_1$ karakterlánc

Objektumok módosítása

Miután a felhasználó megkonstruálta a membránrendszer szerkezetét, utána lehetősége nyílik arra, hogy az egyes régióinak tartalmát szerkessze. Mivel minden egyes régióhoz tartozhatnak objektumok és szabályok is egyaránt, ezért ezek módosítása közös dialógusablakban hajtható végre. A dialógusablak megjelenése a szerkeszteni kívánt régió által lefedett területre történő dupla kattintással kezdeményezhető.

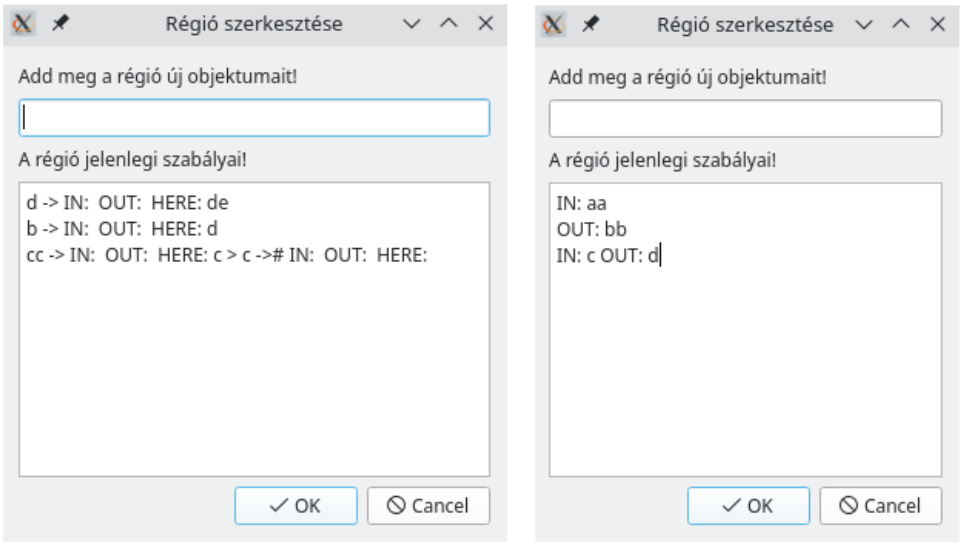


2.4. ábra. Dialógusablak egy régió tartalmának szerkesztéséhez

A régió objektumai közé tetszőleges számban írható a szóköz, mint elválasztó karakter, amely nem kerül figyelembevételre a régió új objektumainak feldolgozásakor. A megengedett karakterek ezen felül továbbra is az angol ábécé kisbetűi. Ha az adott régió a dialógusablak megnyitásakor már rendelkezik objektumokkal, akkor azok alapértelmezett szövegként fognak szerepelni a szövegdobozban. Tehát ha a régió szerkesztésekor az a felhasználó célja, hogy a kiválasztott régió tartalmát újabb objektumokkal egészítse ki, abban az esetben elegendő a hozzáadni kívánt karakterek begépelése.

Szabályok módosítása

A 2.2 segítségével alkotott membránrendszerek még nem fognak szabályokat tartalmazni, ám ezek nélkül nem beszélhetünk még szimulációról. A szabályok fontos szerepet kapnak az információáramlás vezérlésében, hiszen a hatásukra tudnak objektumok átalakulni, be- és kivándorolni a régiók között.



(a) Régió szabályainak szerkesztése alapmodellben

(b) Régió szabályainak szerkesztése szimport-antiport rendszerben

2.5. ábra. Membránrendszerek különböző formátumú szabályokkal

A szabályok felépítése függ a membránrendszer típusától. Minden szabály megadása során először fel kell tüntetni, hogy mik azok az objektumok (olyankor ugyanolyan objektumból akár több is egyszerre), amelyek szükségesek a szabály alkalmazásához.

Szimport-antiport rendszereknél ez a feltétel teljes mértékben elegendő ahhoz, hogy a szabály létrejöjjön, hiszen abban a modellben nincs lehetőség új objektumok létrehozására; a hangsúly inkább az objektumok régiók közötti mozgatására helyeződik. Ahogyan a 2.5b alsó szövegdobozában is látható, szimport-antiport rendszerek esetében két különböző típusú szabály hozható létre. Az első a szimport szabály, amelynek két formája van. Az *IN:* kezdetű szabály alkalmazása esetén a kijelölt objektumok a membránt körülvevő tartományból a jelenleg szerkesztés alatt álló tartományba vándorolnak; az *OUT:* kezdetű szabálynál pedig a szerkesztés alatt álló régióból a membránt körülvevő tartományba kerülnek az objektumok. A másik típushoz az ún. *antiport szabályokat* soroljuk, amelynél egyszerre történik meg az előbbiekkel analóg módon be- és kivándorlás. A bevándorló objektumok előtt az *IN:* prefix szerepel, a kivándorló objektumok pedig az *OUT:* után találhatóak. Ezen két rész a karakterláncon belül felcserélhető, illetve köztük és az objektumok között tetszőleges számú szóköz karakter szerepelhet, amelyek a szabály megalkotásakor nem játszanak szerepet.

Ahogy a 2.5a ábra is mutatja, az alapmodell esetében a szabályok komplexebb felépítéssel rendelkezhetnek. Első nagy különbség, hogy ebben a modellben objektumok átalakulhatnak, illetve eltűnhetnek vagy megsokszorozódhatnak. Ebben a modellben a szabály alkalmazásához szükséges objektumokat a szabály *bal oldalának* nevezzük. A szabály meghatározása ezen objektumok felírásával kezdődik. A szabály alkalmazásának köszönhetően létrejövő objektumok halmazát a nevezzük a szabály *jobb oldalának*, amelyben az objektumok mozgásának figyelembevételével 3 csoportra bonthatunk:

1. Kivándorló objektumok, amelyek a szülő régióba fognak jutni. Ha a legkülső régióban alkalmaztuk a szabályt, akkor az ilyen címkéjű objektumok a környezetbe kerülnek. A szabály megalkotásánál ezen objektumok elé az *OUT:* prefix kerül.
2. Bevándorló objektumok, amelyek véletlenszerűen valamelyik gyerek régióba fognak kerülni (az alapmodell ennél specifikusabban, címke alapján is megengedi az objektumok mozgását, ezt azonban az alkalmazás nem támogatja). A szöveges formátumban ezen objektumokat az *IN:* szöveg előzi meg.
3. Régión belül létrejövő objektumok, amelyek a szabályhoz tartozó régióba kerülnek. Ezen objektumok előtt a karakterláncban a *HERE:* kifejezés kell álljon.

A szabály bal és jobb oldalát a *"->"* szimbólumok választják el, amelyek együttesen egy nyilat reprezentálnak, annak kifejezésére, hogy a bal oldalt a jobb oldal objektumai „váltják fel". Tehát ekkor a *"aa -> IN: bb OUT: cc HERE: dd"* szöveghez tartozó alapmodellbeli szabály két a objektumot igényel a végbemeneteléhez (amelyek felemésztődnek a reakció hatására), a régióból kivándorol két b objektum, az egyik gyerek régióba bevándorol két c objektum, illetve régión belül kialakul két új d objektum.

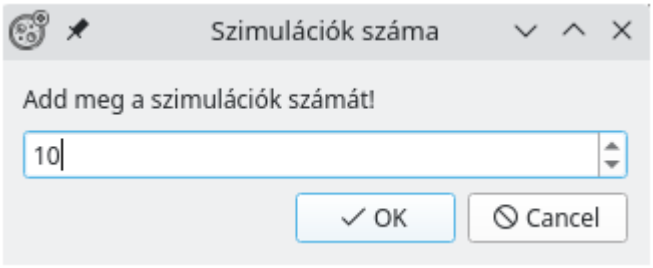
Az alapmodell szabályai kiegészülhetnek többletjelentéssel is. Ebben a modellben ugyanis megengedett lehet egy régió felbomlása, amely egy vagy egyszerre több szabály előidézésére is bekövetkezhet. Ez a szándék jelezhető egy szabály konstruálásakor, ha a bal- és jobb oldalt elválasztó nyilat reprezentáló karakter után közvetlenül egy # karaktert ír a felhasználó. Ilyenkor a régió tartalma és gyerek régiói a szülő régiójába kerülnek.

Szimulációk számának megadása

A korábbi dialógusablakok segítségével a felhasználó lehetőséget szerzett arra, hogy a teljes mértékben a saját elképzeléseinek megfelelő membránrendszer hozza létre és konfigurálja fel objektumokkal és szabályokkal. Ezek után már csak a rendszer számításának szimulálása van hátra. A szimulálás végrehajtására két módot biztosít az alkalmazás:

1. Teljes szimuláció futtatása párhuzamosan tetszőleges számú másolattal
2. Egy szimulációs lépés az aktuális membránrendszeren

Fontos megjegyezni, hogy a teljes szimuláció futtatásának kiválasztása esetén a képernyőn látható membránrendszer nem kerül szimulálásra, annak érdekében, hogy a szimulálni kívánt állapot megőrződjön, így utána változtatások nélkül kezdeményezhető a funkció megismétlése.

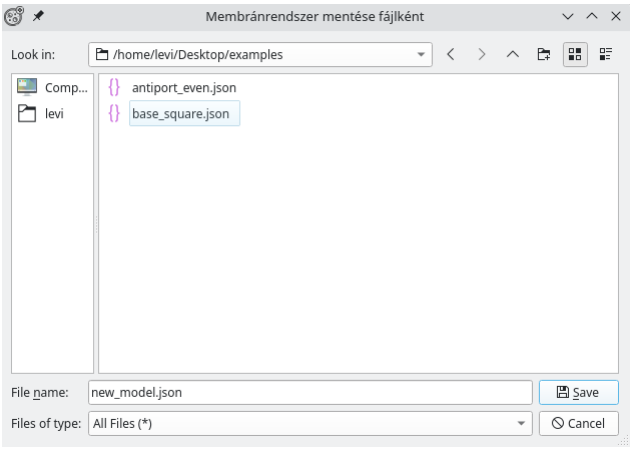


2.6. ábra. Dialógusablak a szimulációk számának kiválasztásához

A szimuláció ismétléseinek számát egy olyan speciális szövegdobozzal lehet beállítani, amely bemenetként csak numerikus értékeket reprezentáló karaktereket fogad el. Alternatívaként a szövegdoboz jobb oldalán található nyilakkal is egyesével állítható az érték, amelynek alsó korlátja *1*, felső korlátja pedig *1000*.

Mentés

Miután a felhasználó kialakított egy olyan membránrendszert, amelyet szeretne elemezni vagy későbbi használat céljából megőrizni, az alkalmazás lehetőséget biztosít a jelenlegi állapot fájlba történő lementésére.



2.7. ábra. Dialógusablak a membránrendszer mentéséhez

A mentés folyamata a *Mentés* gombra történő kattintással vagy a `Ctrl+S` billentyűkombináció lenyomásával kezdeményezhető. A funkció kiválasztása után egy dialógusablak jelenik meg, amelyben a mentés helyére kell navigálnia a fájlrendszerben, majd a fájlnév megadása után az *OK* gomb lenyomására létrejön a *JSON* formátumú fájl a megadott útvonallal.

Mivel a *JSON* formátum könnyen értelmezhető és szerkeszthető, ezért egy jártasabb felhasználója a programnak képes egy lementett membránrendszeren módosításokat végezni a megfelelő szerkesztésekkel.

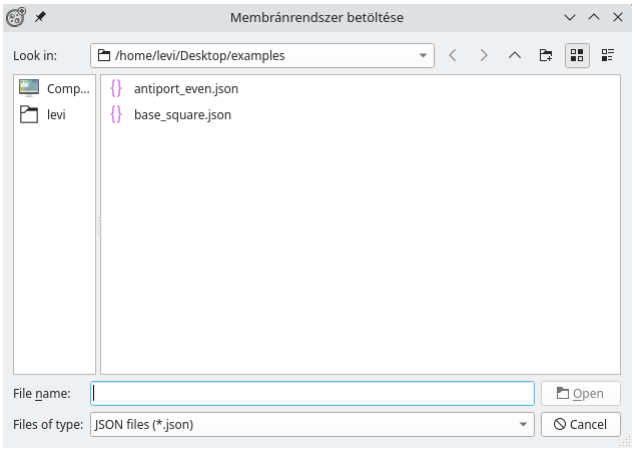
```
{
  "type": "SymportAntiport",
  "structure": "[[[#]]]",
  "rules": {
    "0": [
      "OUT: b",
      "IN: ab"
    ],
    "1": [
      "IN: aa",
      "IN: b OUT: c"
    ],
    "2": [
      "IN: aa"
    ]
  },
  "env_obj": "",
  "env_inf": "a",
  "objects": {
    "0": "b",
    "1": "c",
    "2": ""
  }
}
```

2.8. ábra. Egy lementett membránrendszerhez tartozó JSON fájl [2]

[2]Az ábrán látható konfiguráció a páros számokat generáló szimport-antiport rendszerhez tartozik

Betöltés

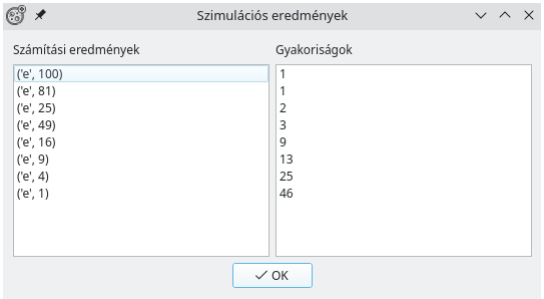
Az alkalmazás biztosítja lementett membránrendszerek betöltését, amelynek hatására betöltődik a kiválasztott fájlhoz tartozó állapot a grafikus felületre. A betöltés nagyon hasonlóan működik a mentéshez, kezdeményezni a *Betöltés* gombra történő kattintással vagy a `Ctrl+O` billentyűkombináció lenyomásával lehetséges, mely után egy dialógusablak segítségével a felhasználó elnavigálhat a megnyitni kívánt fájlt tartalmazó mappához.



2.9. ábra. Dialógusablak egy membránrendszer betöltéséhez

Eredményablak

Miután a felhasználó létrehozott egy membránrendszert és szimulálta azt, nincs más hátra mint értesülnie annak eredményéről. Ezt a funkciót látja el az eredményablak, amely a membránrendszer számításának befejeződése után automatikusan megjelenik és egy listában gyakoriság szerint növekvő sorrendbe rendezve jeleníti meg a különböző számítási eredményeket.



2.10. ábra. Szimulációs eredményeket kilistázó ablak

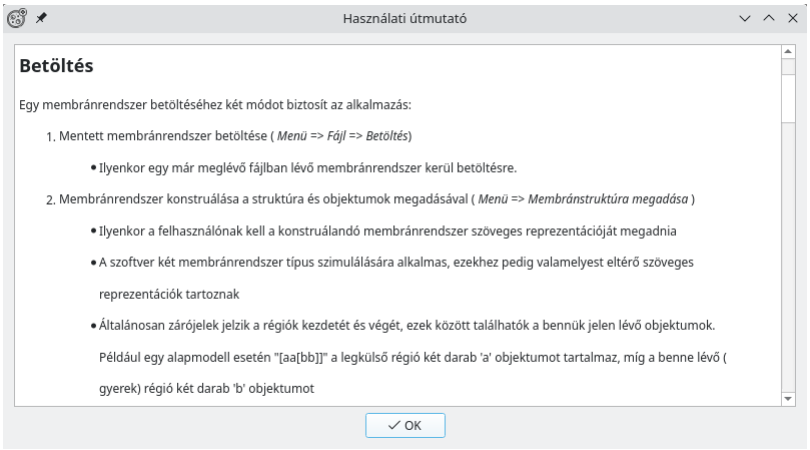
2.5. Használati útmutató

A szoftver használata a membránrendszerek előzetes ismerete nélkül kriptikus tud lenni, ezért a felhasználó útbaigazítására és a szükséges jellemzőkkel kapcsolatos tudnivalók ismertetésére egy súgó funkció is megtalálható a menüsorban.

A *Súgó* gombra való kattintás után jelenik meg az ablak, amelyben a korábbi fejezetekhez hasonló felbontásban találhatóak meg az alkalmazás használatával kapcsolatos információk.

A használati útmutató a *Markdown* formátumnak megfelelő formázásban jelenik meg, amely jól strukturált és könnyen olvasható. A tartalma fejezetekre van bontva, amelyek kitérnek a korábban említett dialógusablakok sajátosságaira, az objektumokkal és szabályokkal kapcsolatos elvárásokra és a szoftver helyes használatának módjára.

A használati útmutató segítségével a felhasználó a membránrendszerekkel kapcsolatos alapvető információkat és a szoftver használatához szükséges tudnivalókat ismerheti meg. A súgó részletes tájékoztatást ad a membránrendszerek létrehozásáról, objektumainak és szabályainak megváltoztatásának módjáról és azok helyes formátumáról, a mentés és betöltés folyamatáról, illetve a számítási eredmény értelmezéséről.



2.11. ábra. Használati útmutató ablak

3. fejezet

Fejlesztői dokumentáció

A szoftver működésének részletezéséhez elengedhetetlen a membránrendszerek formális modelljének ismerete, hiszen ezen rendszerek sajátos viselkedése az architechturális kérdésekben döntő szerepet képez.

A teljes definíció előtt azonban érdemes még pár fogalmat bevezetni. Általánosan tehát egy olyan absztrakt számítási modell áll a szoftver központjában, amely egy biológiai sejthez hasonló felépítéssel rendelkezik, az egymásba ágyazott membránok régiókat különítenek el, amely régiókban különböző objektumok helyezkedhetnek el. A rendszerben az objektumokon felül szabályokat is rendelhetünk régiókhoz, ezen szabályok felelősek a membránrendszer állapotában végbemenő változásokért. A szabályokat a megfelelő feltételek teljesülése esetén kötelesek vagyunk alkalmazni, így egy evolúciós lépés a rendszer számításában addig tart, amíg van legalább egy akalmazható szabály. Érdemes már megjegyezni, hogy egy régió beazonosításához egy egyedi azonosítóra van szükség, illetve, hogy egy régiót könnyen azonosítani lehet az őt legszűkebben tartalmazó membránnal. Így természetes módon adódik, hogy a régiókat fa struktúrába tudjuk szervezni, amelyben minden egyes csúcs egy egyedi azonosítóval rendelkezik. Ezen fában a levelek az elemi membránokat reprezentálják, azaz azokat a régiókat, amelyeken belül nem helyezkedik el régió. Ezen struktúra pedig nagyon könnyen megfeleltethető egy sztringnek.

A másik fontos tervezési szempont az objektumok reprezentálásához kapcsolódik. A membránrendszerek az egy régión belül megtalálható objektumok sokaságát ún. *multihalmazokkal* reprezentálják, amelyben minden objektum rendelkezik egy multiplicitás értékkel, amely számosságát mutatja meg a régión belül. A multihalmaz teljeskörű értelmezéséhez pedig szükség van egy ábécére, amely a lehetsége-

sen előforduló objektumok halmazát tartalmazza. Ezen multihalmazok felett pedig műveleteket kell tudni értelmezni, amelyek az evolúciós szabályok által megkövetelt funkciókat kell magukba foglalniuk. Mivel egy evolúciós szabály felhasználja az alkalmazásához szükséges objektumokat (azaz szintén egy multihalmazt), ezért szükség van egy olyan műveletre, amely a régióhoz tartozó multihalmazból eltávolítja a felhasznált objektumokat. Ezt a funkciót a multihalmazok közötti kivonás művelet fogja biztosítani. Ahhoz, hogy eldöntsük, hogy egy szabály alkalmazható-e egy régióban, elegendő megvizsgálni, hogy a régióhoz tartozó multihalmaz magába fogalalja-e a szabályhoz tartozó bal oldali objektumok multihalmazát. Erre a problémára a halmazok közötti részhalmaz reláció nyújt megoldást. Végül az újonnan keletkező objektumokból álló multihalmaz hozzáadását a régió jelenlegi tartalmához a multihalmazokra nézett unió művelettel lehet modellezni.

Ezen megjegyzések után már bevezethetjük a membránrendszerek formális definícióját [**tichler_slides**].

3.1. Definíciók, formális modell

1. Definíció. Egy $\Pi = \langle O, \mu, \omega_1, \ldots, \omega_m, R_1, \ldots, R_m, i_o \rangle$ rendezett $(2m + 3)$-ast membránrendszernek (alapmodell) nevezzük, ha

1. O egy objektumokból álló ábécé
2. μ egy m membránból álló membránstruktúra. A régiók ekkor a $\{1, 2, \ldots, m\}$ elemeivel injektív módon vannak címkézve. Ilyenkor m-et Π fokának nevezzük.
3. $\omega_1, \ldots, \omega_m$ O feletti multihalmazokat reprezentáló sztringek, amelyek rendre az $1, 2, \ldots, m$ címkéjű régiókhoz vannak rendelve
4. $R_i, 1 \leq i \leq m$ μ i-edik membránjához rendelt O feletti *evolúciós szabályok* véges halmaza. A szabályok $u \to v$ alakúak, ahol $u \in O^+, v \in (O \times TAR)^*$, ahol $TAR = \{here, out\} \cup \{in_j | 1 \leq j \leq m\}$ Ha nem írjuk ki a j indexet, akkor nemdeterminisztikusan történik régió kiválasztása
5. $i_o \in \{1, 2, \ldots, m\}$ egy elemi membrán címkéje, amely a modell kimeneti membránja.

Az előbbi definícióban meghatározott membránrendszerre a dolgozatban alapmodellként fogok hivatkozni. Az alkalmazás az alapmodell esetén csak a nemdeterminisztikus *in* címkét engedi meg a szabályoknál. Ennek oka, hogy a szöveges reprezentációban csak így garantálható az, hogy egy karakter egy objektumot reprezentál, bárhol is szerepeljen a sztringben, illetve a régiók kézzel történő felcímkézésére sincs ezáltal szükség, hiszen ez lenne az egyetlen művelet, amely explicit használja a címkéket.

A számítás formalizálásához fontos bevezetni a konfiguráció fogalmát.

2. Definíció. A $C = (w_1, \ldots, w_m)$ *a* $\Pi = \langle O, \mu, \omega_1, \ldots, \omega_m, R_1, \ldots, R_m, i_o \rangle$ membránrendszer konfigurációja, ha $w_i \in O^*$ és w_i az i-edik régióban lévő objektumokból álló multihalmaz sztring reprezentációja

Megjegyzés. Egy C konfiguráció kezdőkonfiguráció, ha $\forall 1 \leq i \leq m$ esetén $w_i = \omega_i$

3. Definíció. A megállási konfiguráció olyan konfiguráció, amelyre nem lehet már evolúciós szabályt alkalmazni.

4. Definíció. Az egylépéses konfigurációátmenet $C_1 \Longrightarrow_\Pi C_2$, akkor ha C_1-ből egy evolúciós ütemben megkapható C_2 (a maximális párhuzamosság elvének megfelelve).

Megjegyzés. Mivel a szabályok alkalmazása nemdeterminisztikus, ezért a konfigurációátmenet relációban egy C_1-hez több C_2 is létezhet.

5. Definíció. A többlépéses konfigurációátmenet a $\Longrightarrow_\Pi$ reláció reflexív tranzitív lezártja.

A rendszer egy számítása alatt tehát a kezdőkonfigurációból egy megállási konfigurációba történő többlépéses konfigurációátmenet-sorozatot értünk.

Megjegyzés. Ilyenkor a számítás eredményét többféleképpen definiálhatjuk. Alapértelmezetten a kimeneti régióban lévő objektumok számát jelenti. Az alkalmazás az alapmodell esetén ettől eltér és a környezetbe kijutó objektumok számát tekinti a számítás eredményének.

Létezik olyan kiterjesztése a(z) 2. definíciónak, amelyben megengedünk olyan $u \to v\delta$ alakú szabályokat, ahol az $u \to v$ az eddigiekhez hasonló evolúciós szabály, a δ pedig egy speciális szimbólum, amely a membrán feloldódását jelzi a szabály alkalmazásának hatására. Tehát ha egy evolúciós lépésben alkalmazásra kerül egy

ilyen speciális szabály, akkor az ütem végén a szabályhoz tartozó régió felbomlik, ilyenkor a benne lévő objektumok és membránok (a szabályok nem) a membránt közvetlenül tartalmazó szülő régióba kerülnek. A legkülső régió sosem oldódhat fel.

A másik kiterjesztés a szabályok feletti részberendezés lehetőségét adja meg, mely szerint ha r_1 és r_2 relációban állnak (amelyet jelölhetünk $r_1 < r_2$-vel), akkor csak abban az esetben alkalmazható az r_1 szabály, ha r_2 már nem alkalmazható az evolúciós lépésben.

A szoftver mindkét kiterjesztést alapértelmezetten támogatja, működésüket a későbbi fejezetekben részletesebben kifejtem.

Az alkalmazás az alapmodell mellett a szimport-antiport rendszerek szimulálását támogatja. Ebben a modellben az objektumok nem alakulhatnak át, csak a membránstruktúrán belül és a környezetbe vándorolhatnak. Az eddig definiált fogalmak a szimport-antiport rendszereknél is helyt állnak, egyedül a szabályok alakja és működéseigényel módosításokat. Ezen felül a szimport-antiport rendszerek környezete rendelkezhet az O ábécé egy olyan speciális részhalmazával, amelyben előforduló objektumok korlátlanul rendelkezésre állnak az evolúciós lépések során. Ennek megfelelően vezessük be az alábbi definíciót.

6. Definíció. Egy $\Pi = \langle O, \mu, E, \omega_1, \ldots, \omega_m, R_1, \ldots, R_m, i_o \rangle$ rendezett $(2m + 4)$-est szimport-antiport rendszernek nevezünk, ha

1. O egy objektumokból álló ábécé
2. μ egy m membránból álló membránstruktúra. A régiók ekkor a $\{1, 2, \ldots, m\}$ elemeivel injektív módon vannak címkézve. Ilyenkor m-et Π fokának nevezzük.
3. $\omega_1, \ldots, \omega_m$ O feletti multihalmazokat reprezentáló sztringek, amelyek rendre az $1, 2, \ldots, m$ címkéjű régiókhoz vannak rendelve
4. $E \subset O$ a környezetben korlátlanul rendelkezésre álló objektumok halmaza
5. $R_i, 1 \leq i \leq m$ μ i-edik membránjához rendelt O feletti szimport/antiport szabályok véges halmaza
6. $i_o \in \{1, 2, \ldots, m\}$ egy elemi membrán címkéje, amely a modell kimeneti membránja.

Ebben a modellben legyenek $x, y \in O^+$ objektumok multihalmazait reprezentáló sztringek. A velük alkotott szabályok az alábbi alakok egyikét ölthetik:

1. (x, in): Ilyenkor az x multihalmaz a membrán körül elhelyezkedő, őt legszűkebben tartalmazó régióból a membrán által határolt régióna vándorol

2. (x, out): Ilyenkor az x multihalmaz az őt tartalmazó régióból a membránt körülvevő régióba mozog

3. (x, in; y, out): Ilyenkor az x multihalmaz a membrán körül elhelyezkedő, őt legszűkebben tartalmazó régióból a membrán által határolt régióna vándorol, illetve az y multihalmaz a szabályt tartalmazó régióból a membránt körülvevő régióba mozog

Az első két szabályt *szimport szabálynak*, míg az utolsót *antiport szabálynak* nevezzük.

Megjegyzés. Ezen két típuson kívül léteznek még *uniport* szabályok is, amelyek felfoghatók olyan speciális szimport szabályként, amelyekben $|x| = 1$, ahol $|x|$ a multihalmazban szereplő objektumot számát jelöli.

A konfigurációkkal kapcsolatos definíciók megegyeznek az alapmodellben leírtakkal, kiegészítve azt a környezet $O - E$-beli objektumaiból álló ω_0 kezdeti multihalmazt reprezentáló sztringgel, a későbbi konfigurációkban pedig w_0-val.

3.2. Tervezés

A feladat modellezésének legelső mozzanata az alkalmazáshoz szükséges osztályok meghatározása. Elsőként megállapítható, hogy a membránrendszerek egyes típusainak sok közös vonása van, ezért célszerű az absztrakt ősosztály `MembraneSystem` bevezetése . Ez az osztály magába foglal olyan műveleteket, amelyet minden szimulálandó membránrendszer köteles implementálni. Ez az osztály a kód karbantarthatóságát és bővíthetőségét is elősegíti, hiszen esetleges újabb membránrendszerek bevezetése esetén elegendő ezen osztály absztrakt metódusait implementálni. A szoftver esetében két konkrét membránrendszer típusról beszélhetünk, az alapmodellről (`BaseModel`), illetve a szimport-antiport (`SymportAntiport`) rendszerekről, amelyek az ősosztály leszármazottaiként vannak jelen az osztály diagramban.

Általánosan a membránrendszerek régiókból állnak, ezért a `MembraneSystem` osztály az egy régiót reprezentáló `Region` osztályból álló gyűjteményt komponál. A két

osztály közötti kapcsolat aggregáció, mivel egy membránrendszer nem létezne régiók nélkül, illetve absztrakt tekintetben egy régió sem létezhet önállóan, hanem egy csak egy komplexebb entitás (eukarióta sejt) részeként. Ezen régiók pedig valamilyen struktúrába szerveződve helyezkednek el a rendszerben, ennek eltárolására és karbantartására elhivatott a `MembraneStructure` osztály, amely a `Node` osztály segítségével egy *n-áris* fát tárol el és a fa bejárásával begyűjti a membránrendszer által igényelt információkat. A régiók jelentős szerepet játszanak a membránrendszer szimulációjában, hiszen a bennük lévő objektumok és szabályok együttesen határozzák meg a generált nyelvet. Az osztály diagram ennek megfelelően a `Region` osztályhoz az objektumok multihalmazát reprezentáló `MultiSet` osztályt és a szabályokat jelölő absztrakt `Rule` osztályból álló gyűjteményt tartalmazza. A szabályokat összefoglaló ősosztályból a konkrét típusokhoz tartozó szabályok származnak le, így az alapmodell esetén `BaseModelRule` osztály képviseli a létrehozható szabályokat, míg a szimport-antiport rendszer esetében a `SymportRule` látja el ezt a feladatot. Az alapmodellben tovább lehet specializálni a szabályokat, hiszen megadhatóak olyan szabályok, amelyek hatására a hozzá tartozó régió felbomlik, ezen funkciót a `DissolvingRule` tölti be, amely a `BaseModelRule`-ból származik le. A másik ilyen kiegészítése az alapmodellnek a szabályok közötti prioritási sorrend lehetősége, amely a `PriorityRule` osztályban aggregált két `BaseModelRule` típusú objektum segítségével éri el a kívánt viselkedést. A szabályok konstruáláskor valahogyan jelezni kell, hogy a kijelölt objektumokat reprezentáló multihalmaz milyen irányultsággal rendelkezik, azaz a szabály alkalmazása után melyik régióba fog kerülni vagy esetleg kijut a környezetbe. Ezt az információt az alapmodellbeli szabályok esetén a `Direction`, szimport-antiport rendszerek esetében pedig a TransportationRuleType (enum) osztály hordozza Jegyezzük meg, hogy nem csak egy régió hordozhat objektumokat, hanem a membránrendszer környezete is, amely tartalmazhat korlátlan mennyiségben is objektumokat. Ezért a membránrendszer a régiókon felül a `Environment` osztályt is komponálja, amely számomtartja a környezetben korlátlan számban rendelkezésre álló objektumok és a kijutó objektumok multihalmazát.

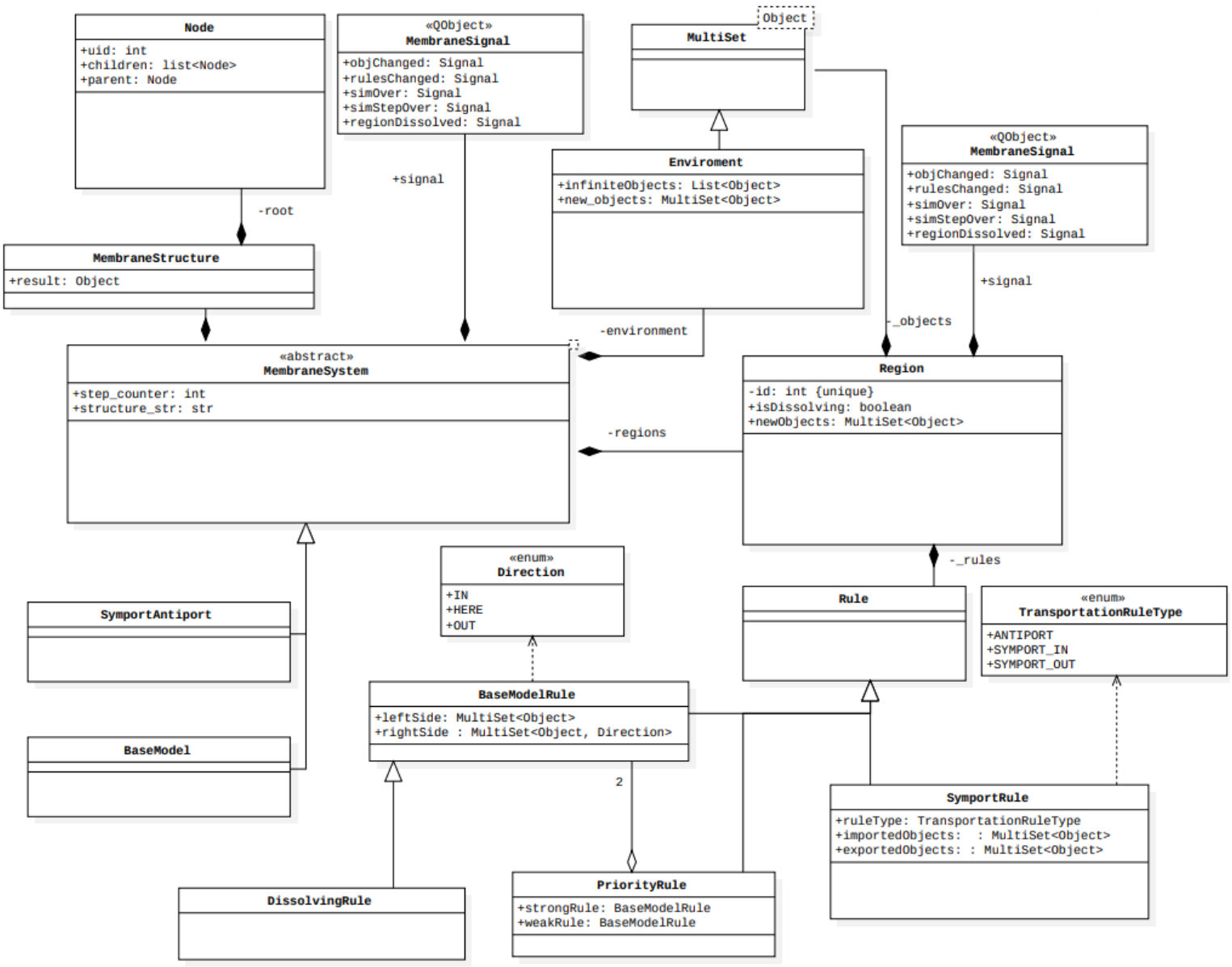
3.2.1. Architektúra

Az alkalmazás *modell/nézet* architektúra elvét követve készült el, ahol a felhasználó mindig a grafikus felhasználói felülettel (nézet) lép interakcióba, amelynek következtében a modell és a nézet közötti kommunikáció eredményeképpen történik változás az alkalmazás állapotában.

A szoftverhez használt eszközök és környezet:

- *Python* programozási nyelv *3.10.4*-es verzió, a *PySide6* modul kiegészítésével
- *Qt* keretrendszer *5.15.3*-as verzió

3.3. Modell



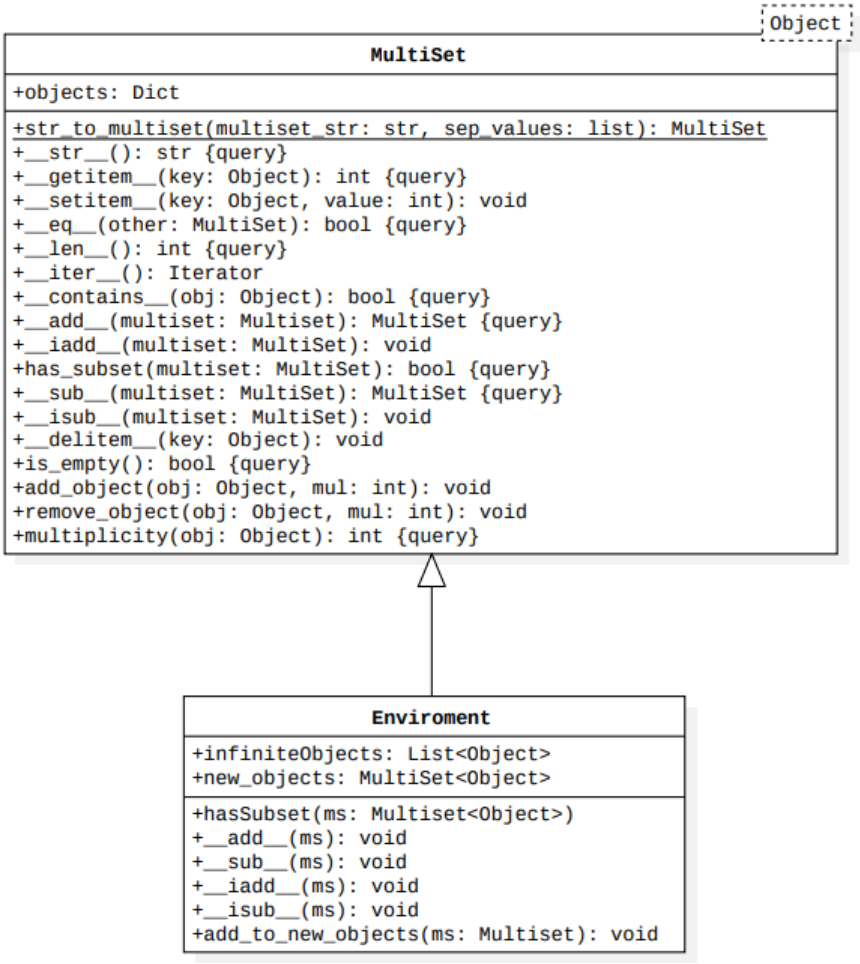
3.1. ábra. A modell osztály diagramja

A 3.1 ábra mutatja az osztály diagramot, amelyben még nem szerepelnek az osztályszintű-és példánymetódusok. Az eddig említett osztályokon felül megjelenik két, a modell és a nézet közötti üzenetváltásért felelős, `QObject`-ből leszármazó osztály, a `MembraneSignal` és a `RegionSignal` . Az előbbi a teljes membránrendszerben előforduló jelzéseket küldi tovább majd a megjelenítésért felelős osztályoknak, míg az utóbbi az egyes régiók tartalmának megváltozását hivatott jelezni. A szoftver működéséhez szükséges objektumok meghatározása még nem elegendő céljának megva-

lósításához, ugyanis szükség van ezen osztályok metódusain keresztül a szimulálás folyamatának megtervezésére is.

3.3.1. Objektumok

Az objektumokat reprezentáló multihalmazokat a `MultiSet` osztály segítségével hozhatjuk létre. Ezen osztály példányai felett értelmezve van az unió, kivonás művelete, a részhalmaz reláció, illetve lekérdezhető egy tetszőleges objektum multiplicitása. A `MultiSet` osztály kiemelt fontosságú a számítás során, hiszen egy szabály alkalmazhatóságának eldöntéséhez két multihalmaz (nevezetesen a régió objektumait tartalmazó és a szabály bal oldala) közötti részhalmaz reláció eldöntésére van szükség. Ha valóban alkalmazható egy szabály, akkor a szabály bal oldala felhasználásra kerül, tehát egy kivonás műveletet kell végezni, a létrejövő objektumok pedig a megfelelő régióhoz az unió műveletével tudnak hozzákerülni. Tehát az osztályt nem csak a régiók, hanem a szabályok is felhasználják működésük során.

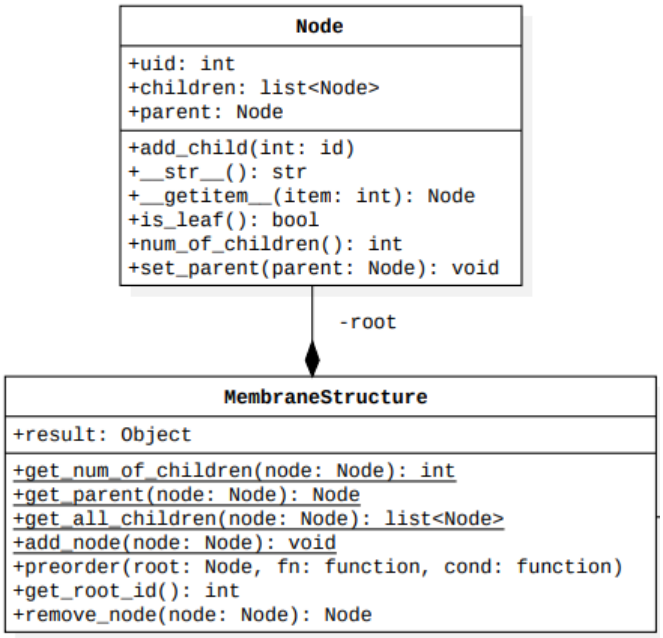


3.2. ábra. Az objektumokat reprezentáló multihalmaz és a környezet

A környezet reprezentálására hivatott `Environment` osztály a korlátlanul rendelkezésre álló objektumokat tartalmazó `infinite_obj` listával egészíti ki a `MultiSet` osztályt, illetve az ősosztályban használt műveletek is felüldefiniálja a helyes viselkedés érdekében. Mivel a környezet is `MultiSet` típusú, ezért a dinamikus típusosság következtében az unió műveletet meghívhatnánk rá `MultiSet` típusú paraméterrel, ennek eredménye szintén `MultiSet` típusú lenne (ami a környezet aktualizálásakor egy evolúciós lépésben nem lenne szerencsés). Ennek elkerülése érdekében ezen műveleteket implementáció nélkül felüldefiniálja az osztályt és csak olyan műveletet biztosít, amely a metódushoz tartozó (`this`) példányhoz unió művelet segítségével hozzáad egy multihalmazt. A hozzáadás során az `infinite_obj`-ban megtalálható objektumok figyelmen kívül hagyhatók. A kivonás esetén is hasonlóan járunk el, illetve ha speciálisan olyan objektumot kell levonni, amely korlátlan számban áll rendelkezésre, olyankor az mellékhatás nélkül végrehajtódhat.

3.3.2. Membránstruktúra

Mivel a membránrendszer struktúrája egyes típusok esetében módosulhat, ezért fontos, hogy a rendszer szerkezetét ne statikusan legyen tárolva, hanem a membránrendszer megkonstruálása után is dinamikusan tudjon változni. Ezeket az elvárásoknak megfelelő n-áris fát láncolt lista adatszerkezettel valósítom meg, amelynek fejelemét (fák esetén inkább gyökerét) a `MembraneStructure` osztály adattagként tárolja el. Minden csúcs egyedi azonosítóval rendelkezik, amely megegyezik a hozzátartozó régió címkéjével. Emellett minden csúcs számontartja a saját szülőjét, illetve gyermekeiből álló `children` listát, amelyhez az `add_child` metódussal lehet új elemet fűzni. Az osztály többi művelete *getter* vagy *setter* metódusként funkcionál.



3.3. ábra. A membránrendszer struktúrájáért felelős osztályok

A `MembraneStructure` osztály a fa struktúra karbantartásáért és a struktúrával kapcsolatos „lekérdezésekért” felelős. Ezen feladatok ellátásához kulcsszerepet kap a `preorder(root, fn, cond)` metódus, amely a bejárja a fát és ha egy n csúcsra teljesül a `cond` függvény feltétele, akkor az `fn` függvény kerül meghívásra.

```
def preorder(self, root, fn, cond):
        if root is None:
            return
        if cond(root):
            self.result = fn(root)
        else:
            iter_count = root.num_of_children()
            for i in range(iter_count):
                self.preorder(root.children[i], fn, cond)
```

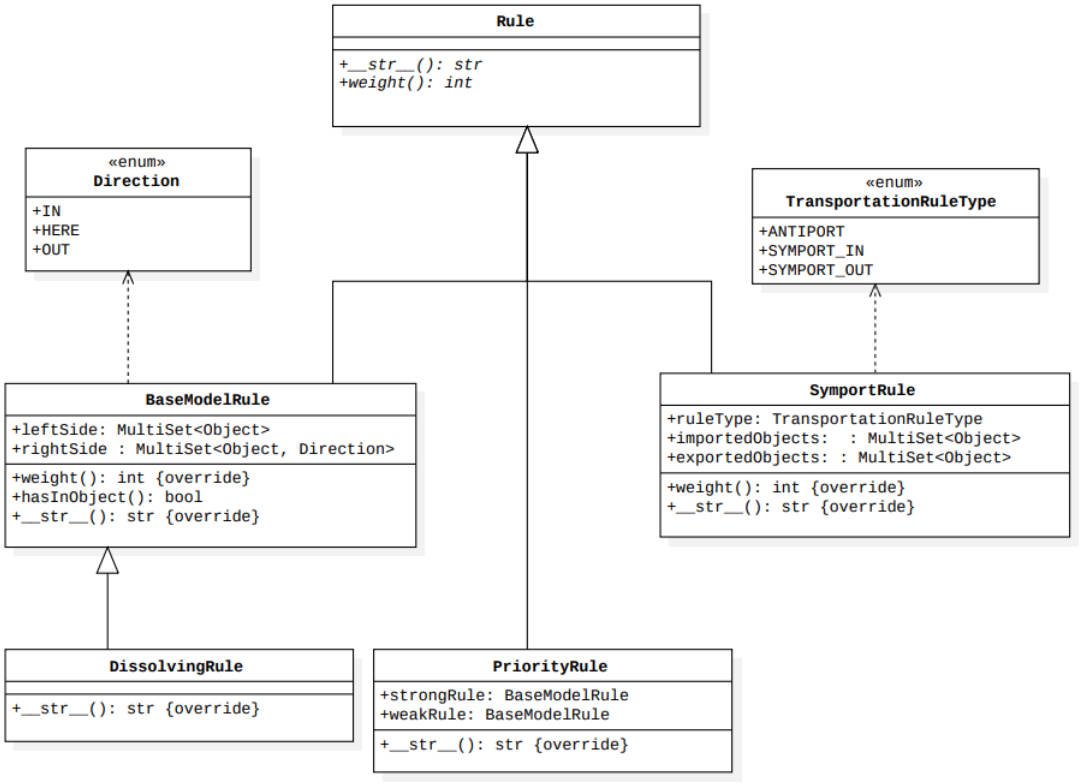
3.1. forráskód. Membránstruktúra preorder bejárása

A `MembraneStructure` kialakítása igazodik ehhez a szemantikához, mivel mind a négy osztályszintű metódus és a `remove_node()` metódus is egy *Node* típusú paramétert vár, amelyen végrehajtják a kívánt műveletet. Tehát ezen metódusok nem járják be a fát és keresik meg a megfelelő címkéjű csúcsot, hanem a `preorder` függvény paramétereként kerülnek felhasználásra, amelyben a feltétel a csúcs egyedi azonosítójára szűr. Mivel a rekurzív függvénynek tetszőleges típusú visszatérési ér-

téke lehetne, ezért a visszaadandó érték a struktúra `result` adattagjába kerül. Az osztály felépítése garantálja, hogy a result lekérdezései nem keresztezhetik egymást, ezáltal elkerülve az inkonzisztens és helytelen értékeket.

3.3.3. Szabályok

A szabályok a membránrendszer számításaiban központi szerepet kapnak, ám kevés olyan tulajdonságuk van, amely minden típusban közös vonásként jelenik meg. Ezért az absztrakt `Rule` ősosztály kevés implementálandó metódust határoz meg, csak a szabályhoz tartozó szöveges reprezentáció, illetve a szabály súlyát meghatározó műveletek meglétét követeli.



3.4. ábra. A membránrendszer szabályait modellező osztályok

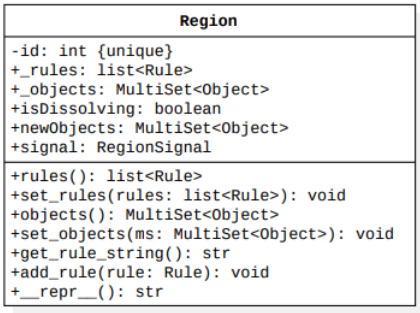
Az alapmodellhez tartozó `BaseModelRule` osztály a leszármazás mellett rögzíti a modellezéséhez szükséges két adattagot. Az egyik a `left_side`, amely a szabály végbemeneteléhez szükséges objektumokból álló multihalmazt tárolja (objektum-multiplicitás párok formájában), a másik pedig a `right_side`, amely a szabály alkalmazásának következtében létrejövő objektumokat és azok mozgásának irányát adja meg. Ezen irányok jelzésére a `Direction` osztályban rögzített `HERE`, `IN` és `OUT` felsorolási típus értékei hivatottak. Tehát a `right_side` olyan multihalmazban a

megfelelő multiplicitással `(objektum, irány)` párok szerepelnek. Tehát egy evolúciós lépésben az alapmodellbeli szabály alkalmazhatóságának eldöntése a szabály bal oldala és a régióhoz tartozó objektumokból álló multihalmaz közötti tartalmazási reláció vizsgálatával ellenőrizhető. Az alapmodell kiegészítéseként beszélhetünk a feloldódás fogalmáról, amely a modellezés során a `BaseModelRule`-ból leszármaztatott `DissolvingRule` osztály segítségével kerül megvalósításra. Ez az osztály semmilyen új metódust vagy adattagot nem vesz fel, az információtartalom a példányosítása mögött rejlik, hiszen ilyen típusú szabály alkalmazása az adott régió `is_dissolving` adattagjának igaz értékre billentését idézi elő. Az alapmodell másik ilyen kiterjesztése a szabályok közötti részbenrendezés lehetőségét nyújtja, amelyet az alkalmazás két szabály közötti prioritás rögzítésével támogatja. Az ilyen szabályok közötti relációk leírására a `PriorityRule` szolgál, amely létrehozásakor két alapmodelli szabályt vár paraméterül. Egy `PriorityRule` típusú szabály alkalmazhatóságának vizsgálata során először a nagyobb prioritással rendelkezdő szabály alkalmazhatóságát kell vizsgálni, majd annak esetleges sikertelensége esetén lehet a kisebb prioritással rendelkező szabály alkalmazhatóságát elemezni.

A szimport-antiport rendszerek esetében a bal-és jobb oldali objektumok helyett *importált* és *exportált* objektumok multihalmazáról beszélhetünk, ahol új objektumok nem jöhetnek létre, csak régiók között vándorolhatnak. Attól függően, hogy ezen két interakció közül melyek hajtódnak végre egy szabályban, három különböző szabálytípust lehet megkülönböztetni, amelyek jelzésére a `TransportationRuleType` felsorolási típus értékei hivatottak. Ezek pedig rendre:

1. `SYMPORT_IN`
2. `SYMPORT_OUT`
3. `ANTIPORT`

3.3.4. Régiók



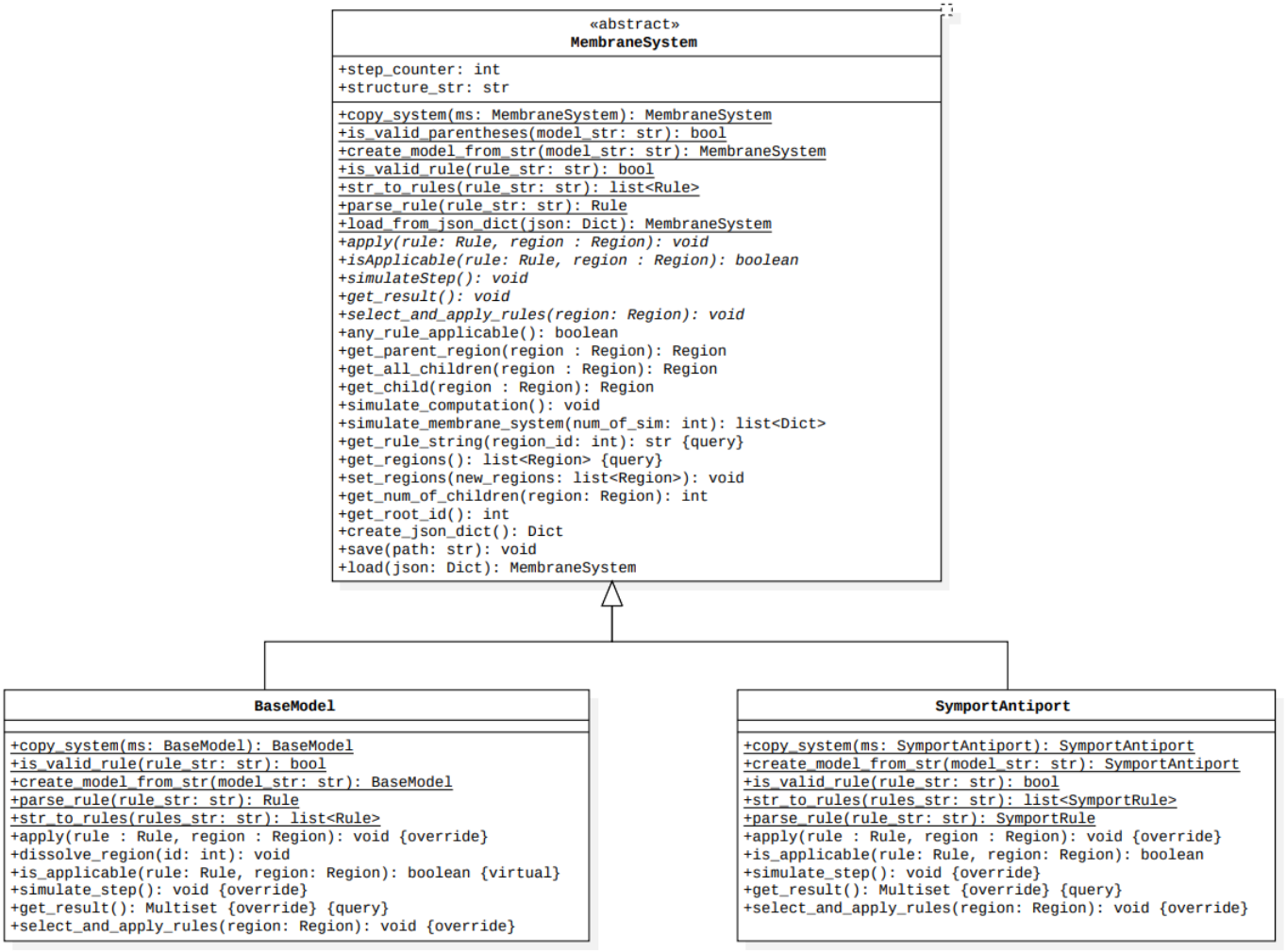
3.5. ábra. A membránrendszer régióit reprezentáló osztály

Egy régió azonosítására a saját egyedi azonosítóját (`id`) használja az alkalmazás, amely megegyezik a membránstruktúrában a hozzá tartozó `Node` objektum `uid` adattagjában tárolt értékkel. Ez az érték a rendszer számítása során nem változik, egyedül egy régió szülőjének címkéje változhat meg (alapmodell esetén feloldódás következtében), ezért ezt az értéket nem lehet statikusan letárolni. A feloldódás jelzésére az `is_dissolving` adattag hivatott, amelynek igaz értékre billenése esetén az evolúciós lépés végén a régió felbomlik. A régió emellett felveszi a `new_objects` adattagot is, amely azon objektumok eltárolására szolgál, amelyek az éppen zajló evolúciós lépésben kerültek a régióba. Ez azért fontos, mert a maximális párhuzamosság elve szerinti szabályok kiválasztása egyidejűleg történik meg az ütem elején, tehát az egyik szabály alkalmazásának mellékhatása nem befolyásolhatja a kiválasztott szabályokat. Ezért amikor alkalmazásra kerül egy szabály, a jobb oldalán található objektumok elrejtésre kerülnek a további szabályok elől, majd az evolúciós lépésben a maximális szabálymultihalmaz után kerülnek a régió `objects` multihalmazába.

3.3.5. Membránrendszer ősosztály és leszármazottai

Az absztrakt membránrendszer ősosztály az eddig taglalt osztályok segítségével képes sablont nyújtani a számítás modellezéséhez. A `MembraneSystem` nem példányosítható (mivel tartalmaz absztakt metódusokat), de konstruktorában a rendszer régiót tartalmazó gyűjteményt, a membránstruktúrát reprezentáló `MembraneStrucutre` objektumot várja paraméterül. A régiókat egy olyan asszociatív tömbben tárolja el a membránrendszer, amelyben a kulcs a régió azonosítója, a hozzá tartozó érték pedig a régió objektum.A leszármaztatott `BaseModel` és

`SymportAntiport` osztályok megöröklik ezt a konstruktort, amelyet a saját konstruktorukban meghívnak. A szimport-antiport rendszerek esetében még a kimeneti régió címkéjét is meg kell adni, ezért ott az további paraméterként jelenik meg. Alapmodell esetében az alkalmazás a környezetbe kijutó objektumok számát tekinti eredménynek, ezért ott nincs szükség további specifikálásra.



3.6. ábra. A membránrendszert modellező absztrakt osztály és leszármazottai

A `MembraneSystem` implementálja a membránrendszer felépítésével kapcsolatos lekérdezéseket. Így például a `get_parent_region()` metódus a paraméterként kapott régió szülő régióját adja vissza. A megvalósításnál a `MembraneStructure` osztály `preorder()` metódusát használja fel, amely a fa gyökeréből indítva a megfelelő `MembraneStructure` osztályszintű metódusával kiegészítve a kívánt viselkedést eredményezi.

```
1 def get_parent_region(self, region):
2         region_id = region.id
```

```
3         self.tree.preorder(self.tree.skin, self.tree.get_parent,
4                            lambda x: x.id == region_id)
5         result_node = self.tree.result
6         return self.regions[result_node.id]
```

3.2. forráskód. Szülő régió lekérdezése

Ez a lekérdezés akkor kerül használatra, amikor egy szabály jobb oldalán `OUT` címkéjű objektum található, ugyanis ilyenkor a szülő régióba fog kerülni az evolúciós lépés során. Ezt a felépítést követi a többi struktúrával kapcsolatos metódus is.

3.3.6. Számítás algoritmusa

Ezen metódusok segítségével minden rendelkezésre áll ahhoz, hogy a membránrendszer számítását modellezni tudja a szoftver. Vannak olyan metódusok, amelyek a membránrendszerek definíciója szerint minden típusban meg kell, hogy egyezzenek. Ilyen például az, hogy mindegyik rendszer számításának kell legyen eredménye, illetve, hogy addig történnek a konfigurációátmenetek ameddig nem jut megállási konfigurációba. Tehát a `MembraneSystem` osztályban bevezetésre kerülhet a `simulate_computation()` metódus, amely megadja a számítás vázát.

```
1 def simulate_computation(self):
2   while self.any_rule_applicable():
3       self.simulate_step()
4   return self.get_result()
```

3.3. forráskód. Membránrendszer számításának általános algoritmusa

A felhasznált metódusok közül csak a `any_rule_applicable()` metódushoz rendelhető általános implementáció, a többi metódus eltérően viselkedhet a különböző típusok esetében. Tehát az `any_rule_applicable()` metódus feladata, hogy megvizsgálja, hogy van-e alkalmazható szabály a rendszerben. Annyiszor fut le tehát szimulációs lépés, ahányszor ez a feltétel teljesül; ennek hiányában a megállási konfigurációba jutott előre definiált eredményével tér vissza a metódus.

```
1 def any_rule_applicable(self):
2   for region in self.regions.values():
3     for rule in region.rules:
4       if self.is_applicable(rule, region):
5         return True
6   return False
```

3.4. forráskód. Membránrendszer leállási feltétel ellenőrzése

A metódus végigiterál az összes régión, azon belül a régió egyes szabályain, és ha talál egy alkalmazható szabályt az `is_applicable()` metódus segítségével, akkor *igaz* értékkel tér vissza, egyébként *hamissal*. Az `is_applicable()` metódus viselkedése eltér az alapmodell és a szimport-antiport rendszerek esetében, ezért ehhez a metódushoz a `MembraneSystem` osztályban nem tartozik implementáció.

Ezen kívül az absztrakt ősosztály több metódus vázát is tartalmazza, ilyen a szabály alkalmazását elvégző `apply()` metódus és a szabályok nemdeterminisztikus kiválasztására szolgáló `select_and_apply_rules()`. Ezek után az alapmodell és szimport-antiport rendszerek sajátosságait figyelembe véve adható algoritmus a számítás leírásához.

Alapmodell

Az alapmodell esetén fontos megjegyezni, hogy a számítás nemdeterminisztikusságának garantálásához nem szükséges megengedni, hogy a szabályok kiválasztása a teljes membránrendszer egészére nézve történjen, hanem elegendő valamilyen sorrendben végighaladni a rendszer összes régióján és lokálisan nézve alkalmazkodni a maximális párhuzamosság elvének, hiszen az adott ütemen belül egy szabály alkalmazásának nincs kihatása egy másik régióra. Így a `simulate_step()` metódus végigiterál a régiókon, és mindegyikre meghívja a `select_and_apply_rules()` metódust. Miután az összes alkalmazható szabály végrehajtása megtörtént, utána történik az egyes régiók tartalmának frissítése. Ez azt jelenti, hogy az ütem során a `new_objects` multihalmazba kerülő objektumok hozzáadódnak az `objects` adattaghoz, majd a `new_objects` tartalma üresre állítódik. A frissítés közben ellenőrzésre kerül, hogy egy régióban végbement-e egy feloldódást előidéző szabály, ugyanis ilyenkor meghívódik a `dissolve_region()` metódus a szóban forgó régióra.

A `dissolve_region()` metódus feladata, hogy a felbomló régióban lévő objektumokat és egyéb régiókat a szülő régióhoz adja, illetve, hogy kitörölje a `regions` asszociatív tömbből a régióhoz tartozó bejegyzést. A `select_and_apply_rules()` metódus feladata tehát egy adott régión belül a maximális párhuzamosság elvének megfelelve szabályok nemdeterminisztikus kiválasztása, amely kiválasztása után meghívja az adott szabályt (a kiválasztott régióra) az `apply()` metódus segítségével.

```
def select_and_apply_rules(self, region):
        indices = list(range(len(region.rules)))
        while indices:
            idx = random.choice(indices)
            if self.is_applicable(region.rules[idx], region):
                self.apply(region.rules[idx], region)
            else:
                indices.remove(idx)
```

3.5. forráskód. Szabályok kiválasztása az alapmodell esetében

A metódus minden régióbeli szabályhoz rendel egy sorszámot, amelyet az `indices` listában elhelyez. Ebből a listából nemdeterminisztikusan választ a metódus egy elemet a beépített `random.choice()` metódus segítségével, amely a listabeli elemek közül egyenletes eloszlás szerint választ. Ezután ha a kiválasztott szabály alkalmazható, akkor meghívja az `apply()` metódust a kiválasztott indexhez tartozó szabállyal, ellenkező esetben pedig törli a sorszámot a listából. Ezen mechanizmus segítségével képes egy régión belül ugyanazon objektumok és szabályok esetén más végeredmény kialakulni.

A `select_and_apply_rules()` metódus működéséhez szükség volt az `is_applicable()` metódusra, amely egy adott szabályról eldönti, hogy alkalmazható-e a jelenlegi konfigurációban. Ez a metódus figyelembe veszi a paraméterben megkapott `rule` változó dinamikus típusát és ez alapján határozza meg a visszatérési értékét. Ha ugyanis a paraméterben kapott változó típusa `PriorityRule`, akkor azt kell megvizsgálnia, hogy a két szabály közül akárcsak az egyik alkalmazható-e, hiszen ez azt jelenti, hogy az erősebb vagy ha az erősebb nem, akkor a gyengébb szabály végrehajtható. Ezzel a `PriorityRule` szabályok esetét a másik két típus esetére vissza lehet vezetni. A `DissolvingRule` osztálybeli szabályok nem alkalmazhatóak a legkülső (skin) régióban. Ezek után a hátralévő két típusnál elegendő azt megvizsgálni, hogy a szabályhoz tartozó bal oldal részhalmaza-e a régió objektumainak (nem figyelembe véve az evolúciós lépés során újonnan bejutó objektumokat). Ez a feltétel a `region.objects.has_subset(rule.left_side)` metódushívással eldönthető, ahol a `region.objects` az adott régió jelenlegi állapotához tartozó multihalmaz, a `rule.left_side` pedig a vizsgált szabály bal oldala (amely szintén egy multihalmaz).

A teljes számítás modellezésének leírásához már csak az `apply()` metódust kell

definiálni. Az `apply()` metódus meghívása csak olyan esetben történik meg, amikor a szabály alkalmazhatósága bebizonyosodott, ezért ez a függvényben nem kerül ellenőrzésre. A metódus feladata kivonni a szabály végrehajtásához szükséges objektumokat a régió objektumaiból, majd végigmenni a szabályhoz tartozó jobb oldalból álló multihalmazon és a megfelelő régió vagy környezet új objektumaiból álló multihalmazhoz hozzáadni a keletkező objektumokat. A szabály jobb oldala egy olyan multihalmaz, amelyben az értékek `(objektum, irány)` párokból állnak. Az `irány` alapján tehát lehet:

1. `IN`: Ilyenkor az objektum az egyik véletlenszerűen kiválasztott gyerek régióba kerül (ilyen biztosan létezik)
2. `OUT`: Ilyenkor az objektum szülő régióba vándorol. Ha a legkülső régióban kerül alkalmazásra ilyen szabály, akkor a környezethez jut.
3. `HERE`: Ilyenkor az objektum régión belül jön létre.

Ha az alkalmazott szabály típusa `DissolvingRule`, akkor a régió `is_dissolving` adattagja *igaz* értékre állítódik.

Szimport-antiport rendszer

Szimport-antiport rendszerek esetén nincs szó objektumok keletkezéséről, hiszen ebben a modellben az objektumok csak régiók között vándorolhatnak. Ebben a modellben viszont számít az, hogy régiónként milyen sorrendben választunk szabályokat, hiszen egy antiport típusú szabály a szülő régió objektumait is igényli végbemeneteléhez.

Ennek következtében több metódusnál is más megközelítést kell alkalmazni. A `select_and_apply_rules()` metódus globálisan fog szabályokat kiválasztani a teljes membránrendszert lefedve, ez garantálja a teljes nemdeterminisztikusságot ennél a modellnél.

```
1 def select_and_apply_rules(self):
2   rule_indices = {}
3   idx = 0
4   for region in self.regions.values():
5     for rule in region.rules:
6       rule_indices[idx] = (rule, region)
7       idx = idx + 1
8   while rule_indices:
9     rand_idx = random.choice(list(rule_indices.keys()))
```

```
10      rand_rule = rule_indices[rand_idx][0]
11      rand_region = rule_indices[rand_idx][1]
12      if self.is_applicable(rand_rule, rand_region):
13        self.apply(rand_rule, rand_region)
14      else:
15        del rule_indices[rand_idx]
```

3.6. forráskód. Szabályok kiválasztása a szimport-antiport rendszer esetében

A metódus elején a `rule_indices` asszociatív tömb sorszám-szabály párosokkal kerül feltöltésre a membránrendszerben megtalálható összes régión keresztül történő iterálással. Ezek után az alapmodellhez hasonlóan nemdeterminisztikusan egy index kerül kiválasztásra, amelyhez tartozó szabály alkalmazhatóságát vizsgálja az `is_applicable()` metódus. Amennyiben ez sikeres, a szabály végrehajtódik, egyébként törlődik a szótárból a hozzá tartozó bejegyzés.

A `simulate_step()` metódusban az előbb említett `select_and_apply_rules()` metódus meghívása után nincs csak a régiók állapotának frissítése van hátra. Ilyenkor az alapmodellhez hasonlóan minden régióban és ebben az modellben a környezetben is a `new_objects` multihalmaz tartalma hozzáadásra kerül az `objects` multihalmazhoz, majd az előbbi kinullázásával befejeződött a szimulációs lépés. Emellett a metódus még eggyel megnöveli a lépések számát tartalmazó `step_counter` adattag értékét.

Egy szabály alkalmazhatóságának kérdése szimport-antiport rendszerek esetében egy fokkal összetettebb, hiszen a környezet rendelkezhet korlátlan mennyiségben objektumokkal, amelyek külön vizsgálatatot igényelnek. Ezen speciális eseteken kívül minden alkalommal azt kell vizsgálni, hogy az exportált objektumokból álló multihalmaz részhalmaza-e a régióhoz tartozó objektumok multihalmazának és/vagy az importált objektumokból álló multihalmaz részhalmaza-e a szülő régiónak (esetlegesen a környezetnek).

Ha az `is_applicable()` metódus úgy ítélte, hogy az adott régió egyik szabálya alkalmazható, akkor az `apply()` metódus kerül meghívásra. Mindhárom szabálytípusnál ilyenkor újból meg kell vizsgálni, hogy a legkülső régióban történik-e a reakció, hiszen az importálásnál és az exportálásnál a korlátlan számban meglévő objektumok multiplicitásán nem kell módosítani. A többi esetben a beérkező objektumok a régió `new_objects` multihalmazába kerülnek, a kijutó objektumok pedig a `objects` multihalmazból kerülnek levonásra.

3.3.7. Párhuzamos számítások

Egy membránrendszer nemdeterminisztikussága miatt érdemes a kezdőkonfigurációból több szimulációt is indítani, hogy megfigyelhetőek legyenek az esetleges mintázatok a számítás folyamatában. Ezt a funkciók a `MembraneSystem` osztály a `simulate_parallel()` metóduson keresztül biztosítja. Ezen metódus meghívásakor előre meghatározott számú szimuláció indul el az aktuális membránrendszer konfigurációjából, ám ezek egymást semmilyen módon nem befolyásolják.

```
def simulate_membrane_system(self, num_of_sim=100):
  def compute(model):
    model_copy = model.__class__.copy_system(model)
    model_copy.simulate_computation()
    return model_copy.get_result()

  cpu_count = multiprocessing.cpu_count()
  futures = []
  results = []
  with ThreadPoolExecutor(max_workers=cpu_count) as executor:
    for i in range(num_of_sim):
      futures.append(executor.submit(compute, self))
    for i in range(num_of_sim):
      results.append(futures[i].result())
  self.signal.sim_over.emit(results)
  return results
```

3.7. forráskód. Párhuzamos szimulációt végrehajtó metódus

A párhuzamos számítások megkezdése előtt fontos, hogy lehessen másolatot készíteni a membránrendszerről. Mivel a Python nyelv alapértelmezetten referencia szerint másol le egy objektumot, ez pedig azt eredményezné, hogy a különböző számítások ugyanazon az membránrendszeren történnének, amely számos problémát idézne elő. A kívánt másolatokhoz létrehozásához mély másolást kell alkalmazni, amelyhez a `copy` modul ad támogatást. A `MembraneSystem` osztály abból következően, hogy leszármazik a `QObject` osztályból, nem szerializálható. Emiatt a mély másolást a létrehozásához elengedhetetlen adattagokon keresztül kell megvalósítani. Ehhez az absztakt ősosztályban definiált `copy_system()` metódus az egyes leszármazott típusoknál való felüldefiniálással valósítható meg.

Tehát a `compute()` metódus először készít az aktuális állapotról egy másola-

tot, majd arra meghívja a számítást végző `simulate_computation()` metódust és visszatér annak eredményével. Az egyes számítások egy `ThreadPoolExecutor` osztály segítségével kerülnek külön logikai szálakra.A számítások feldolgozásának rendelkezésére álló szálak száma megegyezik a futtató számítógépben található processzormagok számával. A paraméterben megadott `num_of_sim` változó hordozza az ismétlések számát, amellyel megegyező számó szál fog indulni. Az egyes szálakon elindított `compute()` metódusok visszatérési értéklt a `futures` listába elhelyezett `Future` típusú objektumok fogják tartalmazni. Az `executor.submit(compute, self)` metódus segítségével kiadjuk a metódus ütemezését a `ThreadPoolExecutor` típusú `executor` objektumnak. A `result` listába ezek után elhelyezhetőek az eredmények, a `futures[i].result()` metódushívással. Ez egy blokkoló művelet, tehát amíg nem tér vissza az i-edik számítás, addig várakozik a vezérlési szál. Ezzel lehet garantálni, hogy a `simulate_parallel()` metódus csak akkor tér vissza a `results` listával, amikor már minden számítás befejeződött.

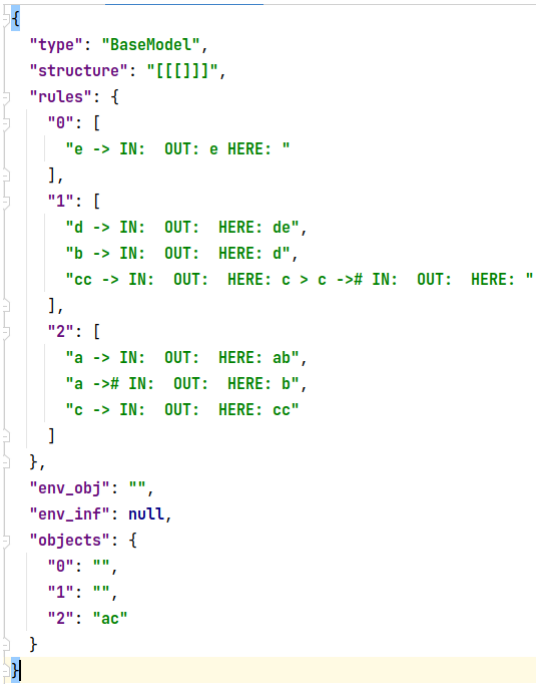
3.3.8. Mentés

Az alkalmazás biztosítja egy membránrendszer lementését, ezért a modellnek támogatnia kell állapotának olyan formátumba való exportálását, amely alkalmas arra, hogy azt az állapotot vissza lehessen állítani. Ehhez kapcsolódóan fontos megjegyezni, hogy a felhasználó egy sztring megadásával is létrehozhat egy membránrendszert, amely rögzíti annak struktúráját és esetlegesen objektumait. Ezt a sztringet a modell inicializálásakor a `structure_str` adattagjába lementi, hiszen az az algoritmus, amely a legelső alkalommal ebből előállítja a megfelelő típusú `MembraneSystem` osztályból leszármazó objektumokt, az az újbóli előállításnál is ugyanazon algoritmussal vissza tudja állítani ezt az állapotot. Viszont ez az állapot eltérhet az eredeti konstruálás pillanatától, objektumok és régiók hozzáadásának következtében. Ezért a lementés előtt ezt a kezdeti `structure_str` változót módosítani kell.

```
1 def save(self, path):
2   json_dict = self.create_json_dict()
3   with open(path, 'w') as save_file:
4     json.dump(json_dict, save_file)
```

3.8. forráskód. Mentésért felelős metódus

A mentés helyét a felhasználó adja meg, majd validitásának ellenőrzése után meghívódik a `save()` metódus, amelynek kulcsmozzanata egy *JSON* stílusú szótár előállítása a `create_json_dict()` metóduson keresztül.Ebben a szótár minden fontos paraméter megtalálható a membránrendszer rekonstruáláshoz. Itt kerül előtérbe az, hogy az objektumok és a szabályok is támogatták a sztring reprezentációt, mert így azokat könnyen rögzíteni lehet a megfelelő címkével.



```
{
  "type": "BaseModel",
  "structure": "[[[]]]",
  "rules": {
    "0": [
      "e -> IN:  OUT: e HERE: "
    ],
    "1": [
      "d -> IN:  OUT:  HERE: de",
      "b -> IN:  OUT:  HERE: d",
      "cc -> IN:  OUT:  HERE: c > c ->#  IN:  OUT:  HERE: "
    ],
    "2": [
      "a -> IN:  OUT:  HERE: ab",
      "a ->#  IN:  OUT:  HERE: b",
      "c -> IN:  OUT:  HERE: cc"
    ]
  },
  "env_obj": "",
  "env_inf": null,
  "objects": {
    "0": "",
    "1": "",
    "2": "ac"
  }
}
```


3.7. ábra. Egy alapmodellbeli membránrendszerhez tartozó *JSON* szótár

A *JSON* szótár részletes tartalmát a 3.8 ábra szemlélteti:

- `type`: A membránrendszer típusa.
- `structure`: A membránrendszerhez tartozó struktúrát reprezentáló sztring (az esetleges objektumok elhagyásával).
- `rules`: A membránrendszer szabályait tartalmazó szótár. A kulcsok a régió egyedi azonosítói, az értékek pedig a szabályok sztring reprezentációjából álló listák.
- `env_obj`: A környezetben nem korlátlanul rendelkezésre álló objektumokból álló multihalmaz sztring reprezentációja.
- `env_inf`: A környezetben korlátlanul rendelkezésre álló objektumokból álló lista.

- `objects`: A membránrendszer objektumait tartalmazó szótár. A kulcsok a régió egyedi azonosítói, az értékek pedig az adott régióhoz tartozó objektumokból álló multihalmaz sztring reprezentációi.

Ekkor a *JSON* szótár minden információt tartalmaz mindkét támogatott típus újbóli betöltéséhez.

3.3.9. Betöltés

A betöltés az előző alfejezetben említett `JSON` szótárat fogja igénybe venni. Miután a felhasználó kiválasztotta a betölteni kívánt membránrendszert tartalmazó fájlt, meghívásra kerül a `MembraneSystem` ősosztályban implementált `load()` metódus, amely a betöltés után a szótárban található `"structure"` kulcshoz tartozó sztringgel meghívja a `create_model_from_str()` metódust. Ez a metódus pedig azon osztály implementációjával lesz meghívva (az ősztály definiálja, de nem implementálja), amely típusnak a neve található a `"type"` értéknél. Így a dinamikus típusosság következtében mindig a megfelelő típusú membránrendszer lesz példányosítva. Ekkor még sem szabályt sem objektumokat nem fog tartalmazni, ám a szótárban található többi információ segítségével ez is elvégezhető. Ehhez azonban fontos, hogy a szabályok és objektumok sztring reprezentációját képes legyen mindegyik membránrendszerből leszármazott típus átalakítani a megfelelő `Rule` vagy `MultiSet` típusra. Ezen paraméterek feldolgozása után előáll a kívánt konfigurációjú membránrendszer. A felhasználó által megadott bemenet feldolgozásáról a következő alfejezetben esik részletesen szó.

3.3.10. Felhasználói bemenet feldolgozása

Az alkalmazás használata során számos helyen a felhasználó által megadott input segítségével hajt végre módosításokat a membránrendszeren. A felhasználó szöveges módon adja meg a régiók objektumait és szabályait, illetve a membránrendszer struktúráját is egy sztring segítségével teheti meg. Ezen interakciók aktualizálására szükség van olyan metódusokra, amelyek ezt a sztring reprezentációt a modell architechtúrájában a tényleges felhasználási formájára konvertálják.

Struktúra megadása

A struktúra megadásánál kihagyhatatlan a megadott bemenet helyességének ellenőrzése. Ennek feltételei:

- A bemenetben a zárójelezésre használt szimbólumok és a szimport-antiport rendszerek esetében kimeneti régió jelzésére használt # karakteren kívül csak a szóköz és az angol ábécé kisbetűi szerepelhetnek.
- A megadott kifejezésnek meg kell felelnie a helyes zárójelezés szabályainak. Ennek eldöntéséhez verem adatszerkezetet használ az algoritmus.

```
1  def is_valid_parentheses(cls, m_str):
2    open_paren = set(['(', '{', '['])
3    close_paren = set([')', '}', ']'])
4    stack = []
5    pairs = {'}': '{', ')': '(', ']': '['}
6
7    for c in m_str:
8      if c in close_paren:
9        if not stack:
10         return False
11     elif pairs[c] != stack.pop():
12         return False
13     else:
14       continue
15     if c not in open_paren.union(close_paren):
16       continue
17     if c in open_paren:
18                 stack.append(c)
19   if not stack:
20     return True
21   else:
22     return False
```

3.9. forráskód. A helyes zárójelezést eldöntő algoritmus

Ha a függvény igaz értékkel tér vissza, akkor létrehozható a struktúrához tartozó membránrendszer a `create_model_from_str()` metódus meghívásával (a megfelelő osztály implementációja szerint).

Szimport antiport rendszerek esetében a legkülső régióhoz tartozó zárójelpáron kívülre írhatóak a korlátlanul rendelkezésre álló objektumok. Megengedett a karakterek ismétlődése, hiszen ilyenkor az objektum többszörös jelenléte nem befolyásolja a környezetet viselkedését. A szóköz karakter itt is megengedett és nem kerül figyelembe vételre.

Objektumok megadása

Az objektumok megadása történhet a struktúra megadásánál és a már létrehozott membránrendszer szerkesztésénél is. A struktúra megadásánál a megfelelő régióhoz tartozó zárójelpár között kell szerepeljenek azok az objektumok, amely a régió kezdeti tartalmát fogják alkotni. A membránrendszer szerkesztésénél a vizuális felületen a szerkesztendő régióra való dupla kattintás után megjelenő felső szövegdobozban kerülnek rögzítésre a módosított objektumok. Mindkét esetben a következő feltételek érvényesek:

- A bemenetben csak szóköz és az angol ábécé kisbetűjei szerepelhetnek.
- A szóköz karakterek nem kerülnek figyelembe vételre az objektumok feldolgozásánál.

megadott kifejezésnek meg kell felelnie a helyes zárójelezés szabályainak. Ennek eldöntéséhez verem adatszerkezetet használ az algoritmus. A bemeneti sztring feldolgozásáért a `MultiSet` osztály osztályszintű `string_to_multiset()` metódusa felel, amely karakterenként dolgozza fel a bemenetet és a nem szóköz karaktereket az eredmény multihalmazhoz a `add_object()` metódus segítségével adja hozzá.

Szabályok megadása

A szabályok megadása egyedül már a létrehozott membránrendszer szerkesztésekor támogatott. A szabályok alakja és viselkedése is eltér az egyes membránrendszer típusok esetén, ezért ezek különböző módszerekkel kerülnek feldolgozásra.

Alapmodell esetén a következőknek kell megfelelni a bemeneti szövegnek:

- A szabályok alakja:
  `"szükséges ->(#) IN: bevándorló OUT: kivándorló HERE: helyben_keletkező"`
- Az objektumokat reprezentáló részsztringben tetszőleges számban szereplhet szóköz karakter, azon kívül csak az angol ábécé kisbetűi szerepelhetnek.

- A szabály végbemeneteléhez szükséges objektumok száma legalább egy kell legyen
- Ha az opcionális # karakter szerepel a megfelelő pozícióban, akkor a megadott szabály feloldódást fog eredményezni
- Ha egy sorban szerepel két, az előbbi formátumnak megfelelő, > szimbólummal elválasztott szabály, olyankor a megadott páros egy `PriorityRule` típusú objektumként lesz értelmezve.

A szabály megfelelő formátumának ellenőrzését egy reguláris kifejezés végzi, amely egyben azt is ellenőrzi, hogy a szabály bal oldala legalább egy objektumból áll.

Szimport antiport rendszerek esetén a három szabálytípusnak megfelelően három formátum is támogatott a feldolgozás során:

1. `"IN: importált_objektumok"`
2. `"OUT: exportált_objektumok"`
3. `"IN: importált_objektumok OUT: exportált_objektumok"` vagy
   `OUT: exportált_objektumok IN: importált_objektumok "`

Itt is elmondható, hogy az objektumokat reprezentáló részsztringben tetszőleges számban szereplhet szóköz karakter, azon kívül csak az angol ábécé kisbetűi szerepelhetnek.

A helyes formátummal megadott szabályok feldolgozásáért a `MembraneSystem` osztályban definiált `parse_rule()` metódus felelős, amely a különböző modelltípusoknál más implementációval rendelkezik. Egy membránrendszer betöltésekor is ez a metódus alakítja át a szöveges reprezentációt a megfelelő `Rule` altípusú példánnyá.

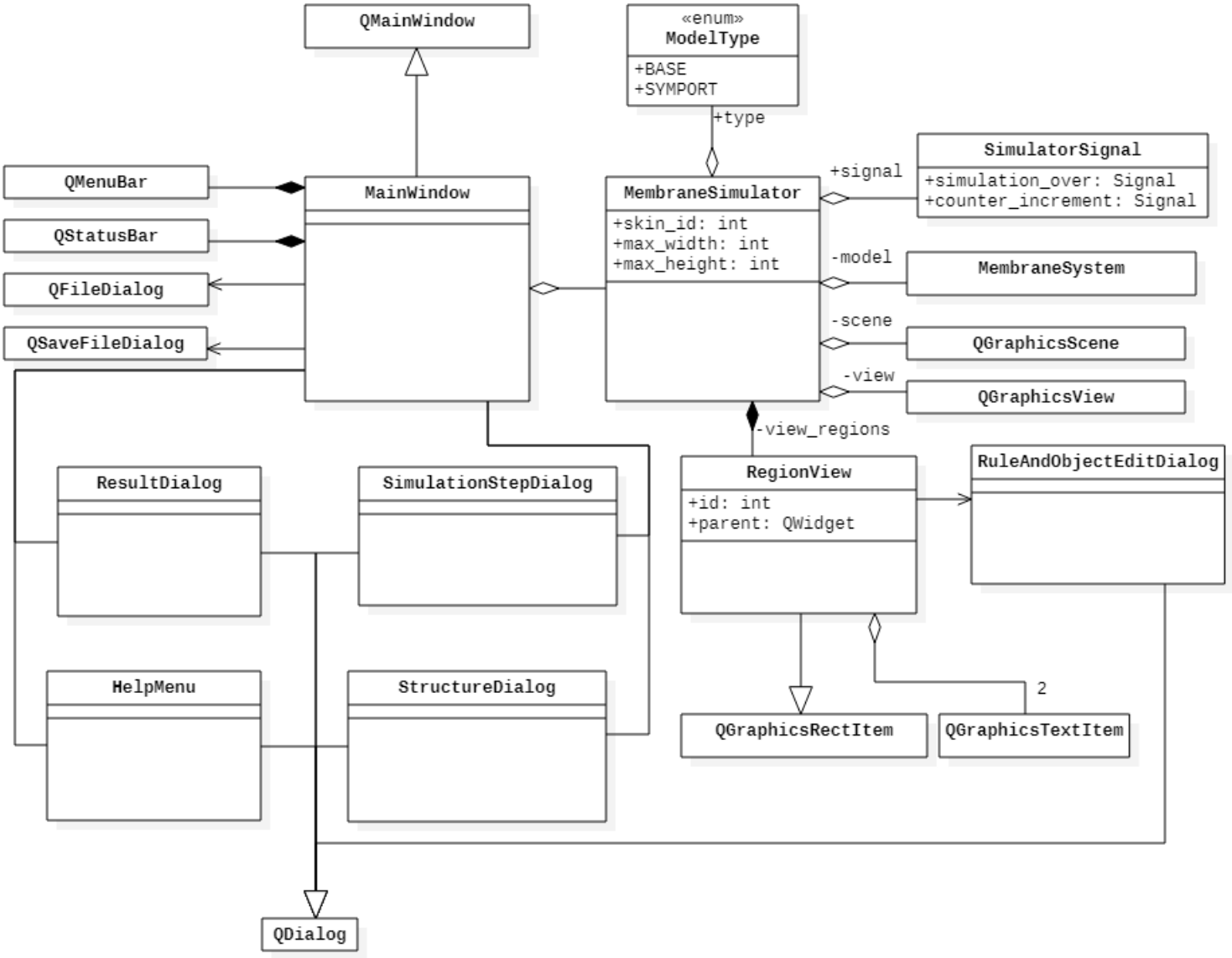
3.4. Nézet

A membránrendszerek vizuális megjelenítését a grafikus felület biztosítja, amely a felhasználóval való kommunikáció során bekövetkezett eseményekről tájékoztatja a modellt.

A tervezés során fontos szempont a *Qt* keretrendszer grafikus eszközei által nyújtott lehetőségek feltárása, hiszen a membránrendszer vizualizálásának megvalósításához egy olyan vászonra van szükség, amely egy tetszőleges membránstruktúrához tartozó sejtszerű ábrát tud megjeleníteni, amelyben az egyes régiók mozgathatóak.

Maga a felület a `QGraphicsScene` és a `QGraphicsView` osztályok közös felhasználásával valósítható meg, ahol az előbbi törődik a "vásznon" található objektumok eltárolásával és manipulálásával, az utóbbi pedig mindezek megjelenítésével. A membránrendszer egyes régióinak kirajzolásához és a bennük található objektumok és szabályok vizuális megjelenítéséért a `QGraphicsRectItem` osztályból leszármaztatott `RegionView` osztály felelős. Mindezen funkciók és megjelenített régiók összefogására a `MembraneSimulator` osztály hivatott, amely a modell felől érkező szignálokra eseménykezelő metódusokkal feliratkozik. Ez az osztály az egyes régiókhoz tartozó `RegionView` példányokat a modellben látottakhoz hasonlóan egy szótárban tárolja, ahol az egyes objektumokhoz tartozó kulcsot a modell adott régiójának egyedi azonosítója jelenti. Így biztosítható az, hogy a nézet oldalán egyértelműen beazonosítható egy régió és a hozzá tartozó megjelenítés.

A szofver főablakát tartalmazó `MainWindow` osztály a `QMainWindow` osztályból származik le, így megörökli az alkalmazás számára hasznos menüsor és státuszsor felhelyezésének lehetőségét. A felhasználóbarát és könnyen átlátható felület elérésének érdekében az adatok bekérése dialógusablakok segítségével történik, amelyekben a megadott bemenet ellenőrzésre kerül. Hiba esetén a felhasználó figyelmeztetésére egy üzenetablak jelenik minden esetben.

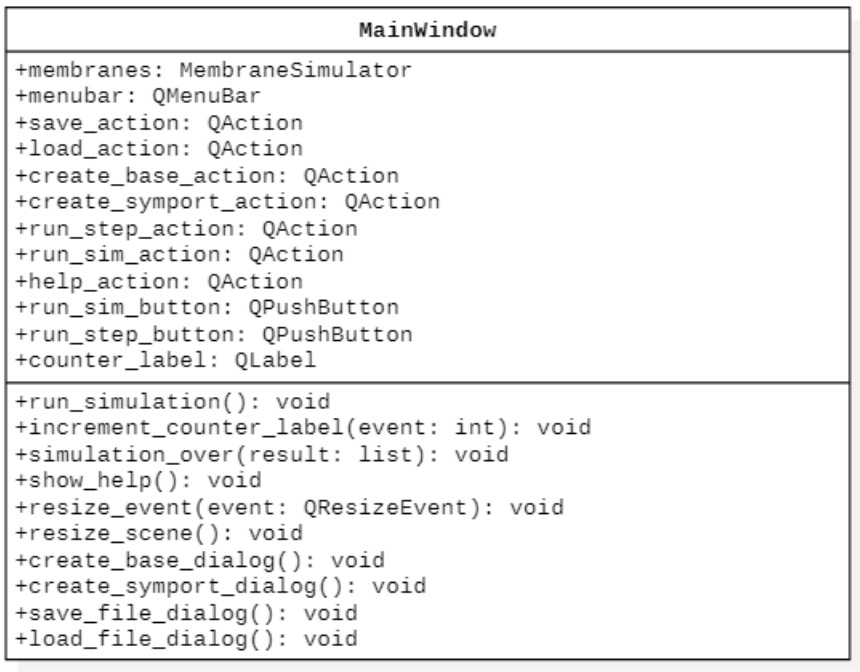


3.8. ábra. A szoftver nézetének osztály diagramja

3.4.1. Főablak

Az alkalmazás főablaka tartalmazza a felhasználóval való interakció megteremtéséért felelős `QAction` típusú objektumokat, amelyek a menüsorban találhatóak. Ezen funkciók almenükre vannak osztva, a megfelelő gombra való kattintással vagy a funkcióhoz tartozó billentyűzetkombináció lenyomásával kezdeményezhető a dialógusablak megnyitása. Ezen dialógus ablakok közé tartoznak a két típusú membránrendszer struktúrájának megadását, illetve a mentés-betöltés funkciókat ellátó ablakok. Emellett ha már egy membránrendszer betöltésre került, akkor a főablak alján egy `QStatusBar` is megjelenik, amely két gombot tartalmaz a szimuláció két

futtatási módjához, illetve a lépések számát megjelenítő `QLabel` objektumot. A menüsor helyet ad a súgó megjelenítését előidéző `QAction` objektumnak is.

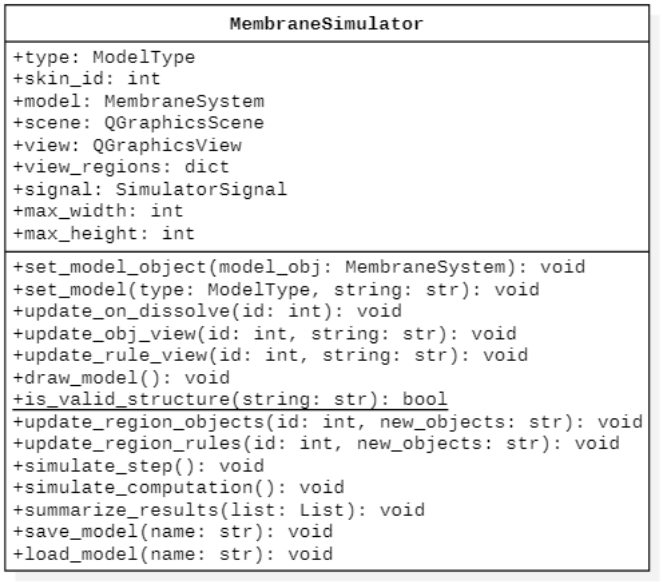


3.9. ábra. Az alkalmazás főablakának osztály diagramja

A főablak legfontosabb komponense a központi *widget*-ként funkcionáló (a membránrendszer szimulálásáért felelős) `MembraneSimulator` osztály `QGraphicsView` típusú adattagja, amely a modellben található membránrendszer vizualizációját jeleníti meg.

3.4.2. Membránrendszer szimulátor osztály

A `MembraneSimulator` osztály az üzleti logikát magába foglaló `MembraneSystem` osztályt és annak vizualizálásához szükséges objektumokat aggregálja. Ez az osztály felelős a modell által kommunikált szignálok feldolgozásáért, illetve a felhasználó által megadott utasítások továbbításáért. A dialógusablakok biztosítják azt, hogy a felhasználó csak a megfelelő formátumú input leírásával tud a modell számára szánt változtatásokat kezdeményezni. A `MembraneSimulator` osztály letárolja a jelenleg szimulálás alatt álló membránrendszer típusát, a legkülső régiójának egyedi azonosítóját (erre az alakzat kirajzolása miatt van szükség), illetve a megrajzolt sejtszerkezet által elfoglalható maximális méretet.



3.10. ábra. Membránrendszer szimulátor osztály diagramja

A `MembraneSimulator` osztály az általa aggregált `QGraphicsScene` és `QGraphicsView` objektumok segítségével jeleníti meg a membránrendszert, amelyben minden régió a hozzá tartozó `RegionView` osztálybeli objektum megrajzolásával kerül megjelenítésre.

Az osztály feliratkozik a `MembraneSystem` által kibocsájtott szignálokra, ezáltal képes az egyes régión az objektumok és szabályok új állapotát befrissíteni. Fontos ilyen szignál még a `region_dissolved()`, amely fellépésekor paraméterként hordozza a felbomlás alatt lévő régió sorszámát. Ilyenkor a hozzá tartozó alakzatot törölnie kell a grafikus felületről, illetve a gyerek régiók szülő referenciáját is frissítenie kell.

A felhasználó által megadott, illetve a szűrő által validnak vélt sztringeket az `update_region_objects()` és `update_region_rules()` metódusok továbbítják a modell felé, amely feldolgozza a sztringeket és a megfelelő módosításokat hajtja végre saját állapotában.

Ezen osztály feladata még a párhuzamos számítások indítása esetén az eredmények összegzése, azaz gyakoriságok alapján rendezni a különböző eredményeket, majd egy szignál segítségével ezt továbbítani a főablaknak.

Régiók vizualizálása

A teljes membránstruktúra megrajzolásáért a `MembraneSimulator` osztály `draw_model()` metódusa felelős. Az osztály rendelkezik a teljes rendszer számára

biztosítható maximális szélességet és maximális magasságot meghatározó adattagokkal, amelyek a megrajzolt alakzat teljes láthatóságát garantálják. Ezek után kezdődhet a régiók legenerálása, amely a következő algoritmust követi:

1. algoritmus A régiók megrajzolására használt algoritmus

1: Legyen $genChildList := []$
2: Generáljuk le a legkülső régióhoz tartozó alakzatot, majd helyezzük a $genChildList$ végére
3: **while** ($genChildList \neq []$) **do**
4: $currentNode := genChildList.pop()$
5: Generáljuk le a $currentNode$ régiót
6: $numOfChildren := currentNode.get_num_of_children()$
7: **if** $numOfChildre \neq 0$ **then**
8: **for** $i = 1, \ldots, numOfChildren$ **do**
9: Az i-edik gyerek régiót a $genChildList$ végére helyezzük
10: **end for**
11: **end if**
12: **end while**

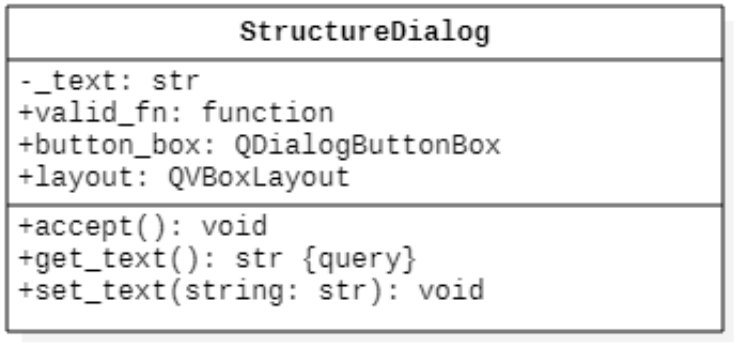
Az algoritmus garantálja, hogy egy gyerek régió csakis a szülő régió legenerálása után kerül feldolgozásra, illetve, hogy minden régió pontosan egyszer kerül kiterjesztésre. Ez az információ fontos ugyanis akkor, amikor a régióhoz megrajzolt lekerekített oldalú téglalap mérete kerül meghatározásra. Egy régió méretét az határozza meg, hogy hány testvére van, illetve, hogy a membránstruktúra fa szerkezetében milyen mélységben helyezkedik el.

Egy adott régió kirajzolása pedig az általa határolt membrán, illetve a benne található objektumok és szabályok megjelenítésével történik. Miután a `draw_model()` metódus kiszámolta a kialakuló régió méretét, ezt átadja a `RegionView` típusú objektum konstruktorának paraméterként. Ekkor a `RegionView draw()` metódusának feladata magát a lekerekített oldalú téglalapot megrajzolni, illetve vízszintesen középre zárva a benne található objektumok és szabályok megjelenítését szolgáló `QGraphicsTextItem` típusú objektumokat létrehoznia. Ezek automatikusan megjelenítik a bennük található szöveget.

3.4.3. Dialógusablakok

Az alkalmazás számos dialógusablakot biztosít a felhasználó számára, amelyen keresztül manipulálni tudja a szimulált membránrendszer belső állapotát. Ezen dialógusablakok nagyrészt egy funkció ellátásra szolgálnak.

Membránstruktúra



3.11. ábra. Membránstruktúra megadásához használt dialógusablak diagramja

A dialógusablakon egy szövegdoboz rögzíti a felhasználó által megadott struktúrát, amely a `MembraneSystem` ősosztályban definiált `is_valid_parentheses()` metódus segítségével kerül validálásra. A szövegdoboz mellett még egy `QDialogButtonBox` típusú osztály biztosítja az ablak elfogadására és bezárása használt gombokat. Ha a felhasználó a `Cancel` gombra kattint, akkor a dialógusablakba írt tartalom nem kerül feldolgozásra. A dialógusablak elrendezése a `QVBoxLayout` osztály segítségével kerül beállításra. A dialógusablak bezárása után a begépelt szöveg a `text` adattag getter metódusán keresztül érhető el.

Használati útmutatót megjelenítő dialógusablak

A dialógusablak elrendezését hasonlóan az előző dialógusablakhoz a `QVBoxLayout` osztály alakítja ki, illetve a dialógusablak elrejtéséhez a `QDialogButtonBox` osztályt használja. Ezen kívül az egyetlen szöveg a `help_md` `QTextEdit` típusú változó segítségével *Markdown* formázással jelenik meg.

A párhuzamos számítások számát beállító dialógusablak

A párhuzamos számítások számát beállító dialógusablak a korábbiakban ismertett technikákat használja az elrendezésre és a dialógusablak elfogadására, illetve elutasítására. A számítások számának beállítására a `QSpinBox` típusú `spin_box` változót aggregálja, amelynek értékét a felhasználó `OK` gombra kattintása után a `get_number()` metódus segítségével lehet lekérdezni.

Egy régió szerkesztéséért felelős dialógusablak

Mivel ez a dialógusablak mindig egy adott régió tartalmához kötődik, ezért megjelenítése és valamilyen formában egyhez kell kapcsolódjon. Így az ilyen dialógusablakok megjelenítését a szerkeszteni kívánt régió által lefedett területre való dupla kattintással lehet kezdményezni. Ezen dialógusablak két szövegdobozt nyújt a felhasználó bemenetének fogadására, az egyik az objektumok, a másik pedig a szabályok rögzítéséért felelős. Miután a felhasználó a `QDialogButtonBox` típusó dobozban az `OK` gombot választja, a megadott bemenet ellenőrzésre kerül. Az objektumok esetén elegendő egy reguláris kifejezéssel szűrni, hogy a megadott sztring kizárólag csak szóközt és/vagy az angol ábécé kisbetűit tartalmazza. A szabályok esetében viszont a bemenetet soronként kell ellenőrizni. Ilyenkor az üres sorokat figyelmen kívül hagyva a dialógusablak paraméterként megkapott `is_valid_rule()` metódus segítségével ellenőrzi a szabályok helyességét. Ha legalább egy szabály vagy az objektumok formátuma nem megfelelő, akkor egy `QMessageBox` típusú ablak jelenik meg a felhasználó értesítésére.

3.5. Tesztelési terv és eredmények

A program egyes komponensei egységtesztek segítségével kerülnek ellenőrzésre. Ezen egységtesztek az egyes metódusok esetében a teljes lefedettségre törekednek, ahol különös figyelmet kapnak a szélsőséges esetek és a hibás viselkedés lekezelése.

3.5.1. Multihalmaz osztály tesztelése

Tesztesetek leírása	**Teszteset eredménye**
Üres multihalmazra meghívott `is_empty()` metódus igaz értékkel tér vissza	Sikeres
Nemüres multihalmazra meghívott `is_empty()` metódus hamis értékkel tér vissza	Sikeres
A multihalmazhoz egyetlen elemet adva eggyel nő meg a mérete	Sikeres
A multihalmazhoz egynél nagyobb multiplicitással rendelkező objektum hozzáadásánál pontosan annak elemszámmal nő meg a multihalmaz mérete	Sikeres
A multihalmazból egy elemet törölve eggyel csökken annak mérete	Sikeres
A multihalmazból egy meglévő objektum multiplicitásánál kevesebb elemet törölve a megfelelő mértékben csökken mérete	Sikeres
A multihalmazból egy meglévő objektum multiplicitásánál több elemet törlése esetén `NotEnoughObjectsException` kivétel kerül kiváltásra	Sikeres
A multihalmazból egy nem benne szereplő objektum törlésének próbálkozása esetén `ObjectNotFoundException` kivétel kerül kiváltásra	Sikeres
Egy multihalmazból néhány objektum (teljes vagy részleges multiplicitással való) törlésével kapott multihalmaz részhalmaza az eredetinek	Sikeres
Egy multihalmazhoz néhány objektum hozzáadásával kapott multihalmaz nem részhalmaza az eredetinek	Sikeres
Egy multihalmazhozból egy annak nem részhalmazát jelentő multihalmaz kivonása `InvalidOperationException` kiváltását eredményezi	Sikeres
Egy multihalmaz inicializálásakor negatív multiplicitású objektum esetén `AssertionError` kerül kiváltásra	Sikeres
Egy multihalmazra meghívott `multiplicity()` metódus visszatérési értéke megegyezik a paraméterként kapott objektum multihalmazbeli számosságával	Sikeres

3.1. táblázat. A tesztesetek leírása

3.5.2. Membránstruktúra tesztelése

Tesztesetek leírása	**Teszteset eredménye**
Egy csúcs létrehozása után az `is_leaf()` metódus igaz értékkel tér vissza	Sikeres
Egy csúcs gyerekeihez való beszúrás után az `is_leaf()` metódus hamis értékkel tér vissza	Sikeres
A membránstruktúrában a skin régiónak nincs szülője	Sikeres
A membránstruktúrában egy csúcs törlése esetén annak gyerekeinek szülő lekérdezése az eredeti csúcs szülőjével tér vissza	Sikeres
A membránstruktúrában egy csúcs törlése esetén szülőjének gyerekeinek száma eggyel kevesebb fog nőni a törlés után, mint amennyi gyereke volt a törölt csúcsnak	Sikeres
A membránstruktúrában egy csúcs gyerekeinek számának lekérdezése a `children` listában tárolt csúcsok számával egyezik meg	Sikeres

3.2. táblázat. A tesztesetek leírása

3.5.3. Szabályok tesztelése

Tesztesetek leírása	**Teszteset eredménye**
Egy alapmodellbeli szabály súlya megegyezik a bal oldalán található objektumok számával	Sikeres
Egy prioritásos szabály nem alapmodellbeli szabályok megadása esetén `InvalidTypeException` kivételt vált ki	Sikeres
Egy szimport-antiport rendszerhez tartozó antiport szabály súlya megegyezik az importált- és exportált objektumok elemszámának maximumával	Sikeres
Egy szimport-antiport rendszerhez tartozó szimport szabály súlya megegyezik az importált/exportált objektumok elemszámával	Sikeres

3.3. táblázat. A tesztesetek leírása

3.5.4. Feloldódás tesztelése

Tesztesetek leírása	Teszteset eredménye
Ha egy felbomló régió egy gyerek régióval rendelkezik, akkor felbomlás után a felbomló régió szülője tartalmazni fogja a felbomlott régióban található objektumokat és a felbomlott régió egyetlen gyerekét	Sikeres
Ha egy felbomló régió több gyerek régióval rendelkezik, akkor felbomlás után a felbomló régió szülője tartalmazni fogja a felbomlott régióban található objektumokat és a felbomlott régió összes gyerekét	Sikeres
Ha a feloldódás a legkülső régióban kerül kezdeményezésre, akkor nem kerül meghívásra a `dissolve_region()` metódus	Sikeres
Ha egyszerre bomlik egy szülő-gyerek pár, akkor az eredeti szülő szülője örökli meg mindkét bomló régió objektumait és a bomló gyerek régió összes gyerekét	Sikeres

3.4. táblázat. A tesztesetek leírása

3.5.5. Mentés és betöltés tesztelése

Tesztesetek leírása	Teszteset eredménye
Egy `save_model()` metódus segítségével lementett membránrendszerhez tartozó *JSON* fájl régiónként a megfelelő objektumok és szabályok szöveges reprezentációját tartalmazza	Sikeres
Egy `save_model()` metódus segítségével lementett membránrendszerhez tartozó *JSON* fájl a membránrendszer struktúráját helyesen reprezentáló sztringet tartalmazza	Sikeres
Egy `save_model()` metódus segítségével lementett membránrendszerhez tartozó *JSON* fájl a membránrendszer típusának nevét tartalmazza	Sikeres
Egy `save_model()` metódus segítségével lementett membránrendszer `load_model()` metódussal történő betöltése esetén visszakapjuk az eredeti membránrendszer konfigurációját	Sikeres
Egy `save_model()` metódus segítségével lementett membránrendszer `load_model()` metódussal történő betöltése után történő módosítások mentése esetén a lementett *JSON* fájl is megfelelő módon változik meg	Sikeres

3.5. táblázat. A tesztesetek leírása

3.5.6. Felhasználói bemenetet feldolgozó komponensek tesztelése

Tesztesetek leírása	Teszteset eredménye
Nem megfelelő karaktert tartalmazó sztring esetén az `is_valid_parentheses()` metódus hamis értékkel tér vissza	Sikeres
Átfedő zárójelpárokat tartalmazó sztring esetén az `is_valid_parentheses()` metódus hamis értékkel tér vissza	Sikeres
Csak megengedett karakteret és helyes zárójelpárokat tartalmazó sztring esetén az `is_valid_parentheses()` metódus igaz értékkel tér vissza	Sikeres
A `create_model_from_str()` metódus a megfelelő membránstruktúrával rendelkező membrárendszerrel tér vissza A `create_model_from_str()` metódus nem veszi figyelembe a szóköz karaktereket	Sikeres Sikeres
A `create_model_from_str()` metódus által létrehozott membránrendszer az egyes régiókban a megadott sztringben szerepeltett objektumokat tartalmazza	Sikeres
Az `is_valid_rule()` metódus a szabályok szöveges reprezentációjánál részletezett reguláris kifejezésre illeszkedő sztringek esetén igaz értékkel tér vissza	Sikeres
Az `is_valid_rule()` metódus a szabályok szöveges reprezentációjánál részletezett reguláris kifejezésre nem illeszkedő sztringek esetén hamis értékkel tér vissza	Sikeres
Az `is_valid_rule()` a nem megengedett karaktereket tartalmazó sztringek esetén hamis értékkel tér vissza	Sikeres
Az `is_valid_rule()` metódus által helyesnek vélt sztring feldolgozása után létrehozott szabály a sztringben szereplő összes objektumot tartalmazza	Sikeres
Alapmodell esetén ha az `is_valid_rule()` metódus által helyesnek vélt sztring tartalmaz `#` karaktert, akkor a létrehozott szabály `DissolvingRule` típusú	Sikeres
Az `is_valid_rule()` metódus által helyesnek vélt sztring feldolgozása során a `parse_rule()` metódus nem veszi figyelembe a szóköz karaktereket	Sikeres

3.6. táblázat. A tesztesetek leírása

4. fejezet

Összegzés

A megvalósított szoftver a membránrendszerek alapmodelljének és szimport-antiport rendszerek tetszőleges példányát képes a felhasználó által megadott konfiguráció segítségével megalkotni, majd szimulálni. A szimulálás során biztosítja a nemdeterminisztikusságot, amelynek szemléltetésére a membránrendszer aktuális állapotából kiinduló párhuzamos számítások esetén egy összegzést készít a felhasználó számára.A lépésenkénti futtatás során is tanúja lehet a felhasználó a rendszer véletlenszerű fejlődésének. A membránrendszerekkel való ismerkedés és az alkalmazás teljes funkcionalitásának elsajátítása céljából egy használati útmutató is található az alkalmazásban, amely elegendő információt nyújt az előbb említett mindkét területen való kezdeti útbaigazításhoz. Az absztrakt modellhez és a természet által inspirált struktúrához a membránrendszer a grafikus felületen vizuálisan jelenik meg, így az a használat során egy felhasználóbarát és intuitív interaktív interfészt biztosít a mögöttes reprezentáció manipulálásához. Így a matematikailag nehezen leírható konfigurációk és konfigurációátmenet-sorozatok egyszerűen és szemléletesen mutathatók be, amely intuíció könnyebb megértést ad a definícióhoz. A modellezés során kulcsfontosságú metódusok az alkalmazás tervezési fázisában majdnem teljes mértékben elkészültek, ám az eredeti modell megvalósítása után felbukkant pár kezeletlen eset, illetve módosítandó algoritmus. Ezen hibák kiszűrésében és a helyes implementáció megalkotásában központi szerepet játszottak az egyes osztályokhoz és funkciókhoz tartozó egységtesztek.

Az alkalmazás továbfejlesztésére számos lehetőség adott, amelyek implementálását elősegíti a bővíthetőségre és karbantarthatóságra tekintettel megalkotott modell architektúra. Az egyik legérdekesebb kiegészítése a programnak az aktív P-

rendszerek szimulálásának támogatása, amely esetén a régiók töltéssel rendelkeznek és bizonyos szabályok segítségével osztódni is képesek.

Irodalomjegyzék

[1] O. J. Dahl, E. W. Dijkstra és C. A. R. Hoare, szerk. *Structured Programming*. London, UK, UK: Academic Press Ltd., 1972. ISBN: 0-12-200550-3.

[2] Thomas H. Cormen és tsai. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[3] Glenn E. Krasner és Stephen T. Pope. „A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80". *J. Object Oriented Program.* 1.3 (1988. aug.), 26–49. old. ISSN: 0896-8438. URL: `http://dl.acm.org/citation.cfm?id=50757.50759`.

[4] E. Dijkstra. „Classics in Software Engineering". Szerk. Edward Nash Yourdon. Upper Saddle River, NJ, USA: Yourdon Press, 1979. Go to Statement Considered Harmful fej., 27–33. old. ISBN: 0-917072-14-6. URL: `http://dl.acm.org/citation.cfm?id=1241515.1241518`.

Ábrák jegyzéke

Táblázatok jegyzéke

Forráskódjegyzék