

# Tartalomjegyzék

<b>1. Bevezetés</b>	<b>3</b>
1.1. UML Komponens fogalmak . . . . .	4
1.1.1. Enkapszulált elem . . . . .	4
1.1.2. UML Interfész . . . . .	4
1.1.3. UML Port . . . . .	4
1.1.4. UML Connector . . . . .	5
<b>2. Áttekintés</b>	<b>6</b>
2.1. A probléma rövid meghatározása . . . . .	6
2.2. Keretrendszerek áttekintése . . . . .	6
2.2.1. fUML - ALF . . . . .	6
2.2.2. Umple . . . . .	7
2.2.3. StartUML . . . . .	7
2.2.4. BrigePoint UML . . . . .	8
2.3. Összefoglalás . . . . .	9
<b>3. Reprezentációs lehetőségek</b>	<b>10</b>
3.1. A megfelelő reprezentációk megtalálása . . . . .	10
3.2. Portok reprezentálása . . . . .	10
3.2.1. UML-ben . . . . .	10
3.2.2. C++-ban . . . . .	11
3.3. Portokra való üzenetküldés/fogadás reprezentálása . . . . .	11
3.3.1. UML-ben . . . . .	11
3.3.2. C++-ban . . . . .	12
3.4. Interfészek reprezentálása interfész portok esetén . . . . .	12

3.4.1.	UML-ben . . . . .	12
3.4.2.	C++-ban . . . . .	12
3.4.3.	Szolgáltatott és elvárt interfészek megkülönböztetése . . . . .	17
3.5.	Konnektorok, összekapcsolás művelet reprezentálása . . . . .	18
3.5.1.	Összekapcsolt portok reprezentációja C++-ban . . . . .	19
3.5.2.	Konnektorok reprezentálása C++-ban . . . . .	22
3.6.	Kommunikáció megvalósítása C++-ban . . . . .	22
<b>4.</b>	<b>Program dokumentáció</b>	<b>25</b>
4.1.	Felhasználói dokumentáció . . . . .	25
4.1.1.	Szükséges eszközök . . . . .	25
4.1.2.	Generálás és futtatás . . . . .	26
4.2.	Fejlesztői dokumentáció . . . . .	26
4.2.1.	Szerkezeti felépítés . . . . .	26
4.2.2.	Tesztelés . . . . .	27

# 1. fejezet

## Bevezetés

A szoftverfejlesztési folyamatban egyre nagyobb hangsúlyt kap a tervezés, a rendszer strukturálása a felelősségi körök szeparálásnak elve, az újrafelhasználhatóság, valamint a rendszer szereplőinek izoláltsága szerint. A végrehajtható UML ezen a ponton játszik nagy szerepet, melynek segítségével magas szinten leírhatjuk a rendszer struktúráját, felvehetjük az egyes szereplőket úgy, hogy már a rendszertervünk is tesztelhetővé, a kódgenerálás révén pedig közvetlen felhasználhatóvá válik.

A szereplők izolálása érdekében kommunikációs csomópontokra, portokra és azokat összekapcsoló absztrakt kommunikációs protokollokra, konnektorokra van szükségünk, melynek révén az adott szereplőnek nem szükséges ismernie a kommunikációban részt vevő felet. Valós idejű rendszerek tervezésénél fontos szerepet játszanak a portok és konnektorok ([ftp://ftp.omg.org/pub/umlrtf/UML\\_rt\\_ext\\_F56dist.pdf](ftp://ftp.omg.org/pub/umlrtf/UML_rt_ext_F56dist.pdf))

A portok és konnektorok fordíthatóságához meg kell ismerni azok pontos UML szemantikáját, a szabvány szerinti reprezentációját, valamint kell találni egy olyan C++ reprezentációt, melynek API-ja letisztult, érhető a felhasználó számára, ugyanakkor az egyes műveletek leghatékonyabb megvalósítására törekszik. A következőkben ezeket a megvalósításokat fogjuk elemezni, egymással összehasonlítani, valamint kiérünk a szabványos UML leképezésre, annak problémáira is.

## **1.1. UML Komponens fogalmak**

### **1.1.1. Enkapszulált elem**

UML-ben az enkapszulált elem egy olyan absztrakt fogalom, egyfajta mechanizmust fejez ki, mellyel képesek vagyunk elszigetelni az objektumot a külvilágtól portok segítségével.

Ez a fogalom nem összekeverendő az UML komponenssel, mely egy enkapszulált elem, viszont több klasszifikált elemet fog össze, egy jól definiált interfésszel rendelkezik, mellyel valamilyen szolgáltatást nyújt a külvilág felé. A komponens fontos tulajdonsága, hogy az adott rendszerben könnyen cserélhető.

Az UML Osztályok is enkapszulált klasszifikációnak számítanak, a diplomamunka az osztályok portjainak exportálását járja körül.

### **1.1.2. UML Interfész**

UML-ben az interfész egy olyan klasszifikáció, melynek a szokványos operációs műveleteken kívül speciális kommunikációs műveletei, úgynevezett reception-ei vannak. Ezek olyan speciális műveletek, melyek egy eseményt várnak, tehát egy interfész leírja, milyen eseményekkel kommunikálhatunk az interfészeken keresztül.

### **1.1.3. UML Port**

Enkapszulált elemek egy adattagja, amely egy kommunikációs csomópontot reprezentál. A típusa lehet egy egyszerű primitív is, de gyakoribb eset, amikor a port egy interfészt valósít meg. Ilyen esetben megkülönböztetünk szolgáltatott és elvárt interfészeket. A szolgáltatott interfész a külvilágtól érkező üzeneteket foglalja magában, míg az elvárt interfész a külvilággal történő kommunikációra szolgál. Arra használjuk, hogy egy klasszifikációt függetlenítsünk a környezetétől, mivel így a külvilággal való kapcsolattartás egy bizonyos típusú porton keresztül történik, nem pedig referencia birtoklásával.

#### **1.1.4. UML Connector**

Egy asszociációt realizál két port között. Leírja, hogy milyen portokat köt össze, assembly vagy delegáció kapcsolat van-e két port között, illetve a kommunikációra használt protokollt.

## 2. fejezet

# Áttekintés

### 2.1. A probléma rövid meghatározása

A diplomamunka célja teljes körű támogatást nyújtani az interfész portokkal történő kommunikációra a txtUML keretrendszeren belül. A portokat és azok műveleteit az eszköz által helyesen és teljesen leírhatónak tekintjük, ennek problémájával nem foglalkozunk. A probléma megtalálni ennek a helyes UML reprezentációját, mellyel a kódgenerálás tovább tud dolgozni, valamint a megfelelő C++ kód reprezentációt, mely megfelelő hatékonyságú és tisztaságú.

### 2.2. Keretrendszerek áttekintése

#### 2.2.1. fUML - ALF

A Foundational UML (fUML) egy Object Management Group (OMG) által specifikált végrehajtható UML szabvány, melynek szöveges akciónyelvi szintaxistát az ALF (Action Language for Foundational UML) definiálja. Mivel ALF az UML szabvány egy részhalmazán alapszik, így az akciónyelvi elemekhez és a strukturális elemekhez (osztály, asszociációk) széleskörű támogatást nyújt, de az állapotgépekhez vagy a kompozíciókhoz nem definiál szintaxist.

### 2.2.2. Umple

Az Umple (link) szöveges alapú modellezőeszköz támogatja a komponens struktúrák leírását. Itt a portok egyszerű adattagokként jelennek meg, melyekhez úgynenevezett aktivátor metódusokat rendelhetünk, melynek hatása az attribútum változásakor érvényesül. A megváltozott érték pedig végig csorog a kapcsolatban álló portokon, melyre szintén egy aktivátor segítségével reagálhatunk. A connectorok pedig egyszerűen portok egymáshoz kötésével jelennek meg (binding) interfész kompatibilitási problémák nélkül. Ezzel szemben a txtUML-ben interfész portokat adhatunk meg elvárt és szolgáltatott interfésszel, egyszerű attribútumokat nem, aktivátorok helyett pedig az állapotgép átmenteti vannak kiegészítve a portok dimenziójával. Az Umple nem foglalkozik a szabványos UML2-es reprezentációval, a C++ kódot közvetlen a modell szövege alapján állítja elő.

Előnyei:

- Online eszközénk is elérhető, nem szükséges hozzá semmit telepíteni.
- Letisztult, jól definiált szöveges szintaxisa van, beleértve a portokat is.
- Tartalmaz kódgenerálást, mely kitrejed a portok akciónyelvére.

Hátrányai:

- Nem generál egy szabványos UML modellt a kódgenerálás előtt, közvetlen a szövegre építkezik.
- A portokat egyszerű adattagként képi le a feltípusozás alapján, nem lehet megadni szolgáltatott és elvárt interfészeket.
- Nincs lehetőség testre szabni az üzenetküldés implementációját a porton keresztül, nincs konfigurációs leírás

### 2.2.3. StartUML

Támogatja a kompozit diagramok létrehozását, így a portok, konnektorok és interfészekét is. Vizuális modellezőeszköz, a létrehozott diagramokból C++ kódot tudunk exportálni. A Portok vázát exportálja, ahogy az interfészeket is, de primitív

szinten, a portok féltípusozása után egy adattagként generálja hozzá az osztályhoz. A konnektorokat már nem exportálja, ahogy a portokhoz kapcsolódó műveleteket sem.

Előnyei:

- Az UML szabványos elemeivel dolgozik
- Különböző kiterjesztések révén tartalmaz kódgenerálást

Hátrányai:

- Az eszköz mellett meg kell ismerni precízen az UML szabványos elemeit is, hogy fel tudjuk tölteni az elemek tulajdonságait
- A kódgenerálás csak a vázra terjed ki, mely tartalmazza a portokat, de az azokkal való műveleteket már nem.
- Nehézkes, vizuális használat, az elemeket szabadon létrehozhatjuk, és könnyen invalid modellt generálhatunk, míg a txtUML exportja a szabványos modellt exportálás révén hozza létre, így kisebb lehetőségünk van hibázni.

#### **2.2.4. BrigePoint UML**

A StartUML-hez hasonlóan az UML szabványos elemeiből építhetünk fel vizuálisan egy szabványos UML diagramot. Azonban nem tudunk osztályok között portokkal kommunikálni, csak a komponensek között, így ez a megoldás nem elég általános. Portot, illetve konnektort önmagában nem tudunk felvenni, az interfészek segítségével tudunk egy kapcsolatot húzni két komponens között, mely leképződik egy porttá és konnektorra. Hasonlóan a StartUML-hez, az akciókat aktivitás node-ok révén kézzel kell összerakni, így nehéz szabványos port műveleteket létrehozni. Összességében a BridgePointról is elmondható, hogy kezdetlegesen támogatja a portokkal történő kommunikációt, de kiforratlan, hiányos, a kódgenerálás ellenére nehéz érdemben azok reprezentációjáról beszélni.



## 2.3. Összefoglalás

Összességében az figyelhető meg az egyes exportálási módok között, hogy a portokat mindig valamilyen primitív feltípusozott adattagként exportáljuk. A másik lehetőségünk, melyet mi is követni fogunk, az interfész portokat összetett típusként exportálni, melyben benne van a szolgáltatott és elvárt interfészt. Portok összekapcsolásának, a portokon történő kommunikáció végrehajtási szemantikájával pedig nem mindegyik keretrendszer foglalkozik, még ha részben támogatja is a portok exportálását. Mindebből az figyelhető meg, hogy egy viszonylag új területről beszélünk, melyben egy teljesen körű támogatás és elemzés referencia értékű lehet későbbi megoldások, fejlesztések számára. A diplomamunka kielemez több különböző, teljes körű támogatást a portok és interfészek C++ nyelvre történő exportálásához, mely magában foglalja a szabványos UML modell létrehozását egy jól definiált leírónyelv alapján.

## 3. fejezet

# Reprezentációs lehetőségek

### 3.1. A megfelelő reprezentációk megtalálása

Az eddig leírtakból kiderült, hogy a szabványos reprezentáció megtalálásával is problémákba ütközünk, mely a fellelhető irodalomból és eszközökből is csak nehezen derül ki, így kitérünk ezek ismertetésére is. A C++ reprezentációhoz pedig egy saját Port könyvtárat és generálási módszert választunk, mely a felelhető módszerek finomításából és több különböző ötlet elemzésének következtében hozunk létre.

A továbbiakban a következő elemek reprezentációjával foglalkozunk:

- Interészek
- Portok és üzenet küldés/fogadás művelet
- Connectorok és connect művelet

### 3.2. Portok reprezentálása

#### 3.2.1. UML-ben

A portok reprezentálása értelemszerű, egy speciális Property-t kell felvenni az osztályon belül. Amit fontos megemlíteni, hogy a szolgáltatott interfész segítségével fogjuk feltipúsozni.

### 3.2.2. C++-ban

A portokat minden esetben adattagként reprezentáljuk, ezt mi sem tudjuk másképp megtenni. Többféle képen használhatók kommunikációra, de minden esetben egyfajta ravaszként működik a rajtuk történő állapotváltozás. A port adattag értéke reprezentálja az üzentet. Ennek megfelelően általánosságban kétféle megoldás létezik a portokon történő kommunikációra:

- Hasonlóan az események feldolgozásához, úgy a portokon történő üzenetáramlást is átmeneti függvényekkel dolgozzuk fel.
- Külön aktivátor függvényeket valósítunk meg az osztályon belül az egyes portokra, melyek a portok állapotváltozások aktiválódnak.

A mi esetünkben az előbbi megoldás tűnt kézenfekvőnek, mivel az könnyen adaptálható a meglévő generálási szabályok közé, valamint a forrásnyelv is ezt a konvenciót követi.

## 3.3. Portokra való üzenetküldés/fogadás reprezentálása

### 3.3.1. UML-ben

#### Üzenet küldés

A *SendObjectAction* két paraméterből áll, egy célobjektumból és egy üzenetből. A célobjektum valamilyen *InputPin*, vagyis valamilyen akciónak az eredménye. Mivel a port egy strukturális elem, így a referenciáját egy *ReadStructuralFeatureAction*-nal megszerezhetjük.

- Könnyű reprezentálni, illeszkedik a modellfordító elvébe, aki feltételezi, hogy egy *send* akciónak van egy cél objektuma, ahova az üzentet kell küldeni.
- Nem kell extra logikát beiktatni a C++ fordítóba az üzenetküldés terén.

## Üzenet fogadás

Ha egy objektumnak a portján keresztül szeretnénk üzenetet küldeni a külvilág felől, akkor a *SendObjectAction*-nak a az *OnPort* műveletét használhatjuk.

### 3.3.2. C++-ban

#### Üzenet küldés

Ez esetben már a *send* művelet végrehajtási szemantikája érdekes számunkra. A *send* művelet azt jelenti, hogy a Portra teszünk egy üzenetet, melyet továbbítunk egy másik komponens felé konnektoron keresztül. Ennek szájharmonikájával a [3.5](#) fejezetben majd foglalkozunk részletesen.

#### Üzenet fogadás

Valahogyan ki kell fejeznünk azt is, ha egy üzenetet nem a külvilág felé delegálunk, hanem a külvilág felől érkezett egy üzenet a Portra, melyet fel kell dolgoznunk állapotgéppel összekapcsolt portok esetén, vagy tovább delegálunk a belső komponensek felé. Erre két lehetőségünk van: Vagy egy teljesen új függvényt vezetünk be (nevezzük ezt *receive*-nek), vagy paraméterezzük a *send* függvényünket. Előbbi megoldás tisztább, jobban érhető a felhasználó számára, valamint a szolgáltatott és elvárt interfészek kifejezésénél is előnyösebb lesz. ([3.4](#))

## 3.4. Interfészek reprezentálása interfész portok esetén

### 3.4.1. UML-ben

Az interfészeket UML-ben értelemszerűen reprezentáljuk, annyiban különböznek csak az osztályoktól, hogy megadhatunk neki. fogadási műveleteket, un. recept-ionőket.

### 3.4.2. C++-ban

C++-ban számos lehetőségünk van kifejezni egy UML interfészt.

## Nincs interfész

Egy lehetséges megoldás az is, ha nem vesszük figyelembe a szolgáltatott és elvárt interfészeket akkor, mikor egy adott portot manipulálunk, üzenetet küldünk rá. A forrásnyelv validációja hivatott kiszűrni az ilyen nem helyes üzenetküldéseket, hivatkozásokat.

Előnyök:

- Kódgenerálás szempontjából nagyon triviális megoldás, nem szükséges foglalkozni egyáltalán vele.
- Alacsony szintű, hatékonysági kérdésekben a legjobb megoldás.

Hátrányok:

- Külső komponensek, portok esetén magunk sem tudunk felvenni interfészeket, ilyenkor már nem hagyatkozhatunk a forrásnyelv validációjára, könnyen hibázhatunk.
- Ha nem is vezetünk be új komponenst, külső kommutáció esetén tetszőleges üzenetet küldhetünk a portra, ami szintén hibás lehet.
- A generált kód kevésbé lesz érthető
- Az interfészeket sehol máshol sem tudjuk felhasználni, osztályok esetén sem, mely szintén a modellel való kommunikációban jelent hátrányt.

## Van interfész

Ezen belül meg kell vizsgálni az interfész port szintaxisát, valamint a mögöttes végrehajtási szemantikát, illetve magának az interfésznek a leírási módjait.

Ahogy azt korábban írtuk, a port típusa az UML-ben a szolgáltatott interfész, ezért kézenfekvő megoldás lenne felvenni egy olyan adattagot, melynek ez az interfész a típusa. Ennek azonban több hátránya is van, sokkal nehezebben tudjuk kifejezni, milyen elvárt interfésze van a portnak, valamint nem tudjuk kifejezni a port típuson belül az üzenetküldés végrehajtási szemantikáját, valamilyen külső implementációra kényszerülünk. Ezáltal bonyolultjuk a megvalósítást, veszítünk az absztrakcióból, de a hatékonyságot nem növeljük.

Így a másik lehetőségünk bevezetni a Port típust, melynek sablon paramétere a szolgáltatott és elvárt interfész. Ezáltal a következőképpen vehetünk fel egy portot egy osztályba:

```
// ProvidedInf, RequiredInf Definitios.. PortjProvidedInf, RequiredInf; p;
```

Ezek után technikailag többféle lehetőségünk van megvalósítani a validációt:

1. Generáljunk egy üres osztályt az interfészekből, majd a szignálok származzanak le aszerint, hogy melyik interfész fogadó műveletei között vannak benne:

PL: UML interfész neve: PingPongInterface, a receptionjai pedig a következő szignálokat fogadják: Ping, Pong

Ekkor a generált kód: `class PingPongInterface; class Ping: public PingPongInterface .. class Pong: public PingPongInterface ..`

Majd a portnak legyen egy *send* művelete, ami az elvárt interfész alá sorolt szignálokat várja, vagyis olyan szignálokat, melyek implementálják a megfelelő interfészt. `send(PingPongInterface ..)` Mindezzel fordítási időben biztosítjuk, hogy ne adhassunk át olyan szignált, melyet nem soroltunk az elvárt interfész szignáljai közé. Hasonlóan, a *receive* művelet olyan szignálokat vár, melyek leszármazottjai a szolgáltatott interfésznek.

Előnyök:

- Egyszerű megvalósítás, kevés extra kód generálásra van szükség hozzá, az interfészek lényegében csak üres, jelző osztályok
- Egyetlen *send* függvénnyel meg lehet oldani a validációt, így a kód a lehető legkisebb lesz, nem lesz technikai redundancia

Hátrányok:

- Nem általánosított megoldás. Az interfészt bármilyen szereplő használhat, és le is származhat belőle, azonban ezzel erősen portokra korlátozzuk a megoldást, mivel a küldő és fogadó műveletek a

portok kódjába vannak beleégetve.

2. Ezt a gondolatmenetet folytatva vizsgáljuk meg azt az általános lehetőséget, hogy a Port megvalósítja az interfészt. A szintaxis nem változik, továbbra is sablon paraméterként szeretnénk megadni az interfészeket a portban. Mivel most a szignáloknak nincsen interfész őse, így minden egyes recepciön művelethez kelleni fog egy-egy függvény. Mivel az üzenetküldés végrehajtási szemantikája nem változik, így vezessünk be egy általános *sendAny* védett metódust, mely bármilyen szignált képes fogadni, azonban a publikus interfészen nem látható. Ezt a leszármazott osztályban felül tudjuk definiálni az üzenetküldés szemantikájától függően. Amikor a *sendAny* továbbítja az adott portnak a megfelelő szignált, a túloldalon lévő port már csak egy általános eseményt lát, nem tudjuk ellenőrizni a típuskonzisztenciát. Azonban két port összekapcsolásakor tudunk gondoskodni róla, hogy ne köthessünk össze inkonzisztens interfészű portokat.

Előnyök:

- Általános megoldás, az interfészt ténylegesen megvalósítjuk, így ez akár modell osztályokra is kiterjeszthető
- Funkcionálisan nem különülnek el a *send* függvények egymástól, így a karbantartási idő nem nő

Hátrányok:

- A generált kód mérete azonban jelentősen megnő, mivel több *send* függvény generálásra van szükségünk, melyek csak a várt szignál típusban különböznek.
3. Mint a fentiekből látjuk, az utóbbi megoldásnak egy hátránya van, hogy növeljük az interfész kódját az első megoldáshoz képest. Amíg csak generáljuk az interfészt, addig ez elhanyagolható tényező, azonban, ha kézzel kell hozzáadni a programunkhoz, hogy kiterjesszük a modellel való kommunikációt, már probléma lehet. Megfigyelhetjük, hogy minden interfész implementációja csak abban fog különbözni, hogy különböző szignálokat tudnak fogadni. A C++ template metaprogramozási lehetőségeit kihasználva próbáljuk

meg az interfész kódját egészen odáig szűkíteni, hogy csak fel kelljen sorolni egy típuslistába, hogy milyen szignálokat tud küldeni illetve fogadni.

Cél szintaxis: `using PingPongInf = GeneralInterface<Ping,Pong>;`

Ez azt jelentené, hogy a `PingPongInf` egy olyan interfész lenne, amelynek két fogadó művelete van, a `Ping` és a `Pong`.

Vizsgáljuk meg a szolgáltatott részt, az elvárt ezzel analóg. Vezessünk be egy olyan sablon *send* függvényt, melynek a sablon paramétere az elküldendő szignál típusa. Ezt látjuk a publikus interfészen, eszerint bármilyen szignállal meg tudjuk hívni az adott függvényt. Azonban a függvény delegálja a hívást egy olyan védett sablon függvénynek, mely bármilyen szignált képes fogadni, és rendelkezik egy logikai sablon paraméterrel: ebben adhatjuk meg, hogy a delegált szignál része-e az interfésznek vagy sem. Template specializáció segítségével kétféle implementációt adunk ennek a függvénynek: Az igaz ág, mely végrehajtja a szignál küldést, a hamis ág pedig szemantikai fordítási hibára fut, ezzel ellenőrizve fordítási időben, hogy ne adhassunk át olyan szignált, mely nem része az interfésznek.

A kérdés már csak az maradt, hogyan döntjük el egy adott szignálról, hogy része-e az interfésznek. Az egyik lehetőség bevezetni egy sablon osztályt, melynek egy adattagja megmondja, hogy *reception* szignálról beszélünk-e. Ezt alapértelmezetten hamisra állítani, és specializálni igaz értékekkel a sablon osztályt a megfelelő szignálokkal. Ezzel azonban még nem értük el a kívánt egyszerű szintaxist, de a *send* függvényt legalább nem duplikáltuk fölöslegesen. A *TYPELIST* konstrukciót használva viszont eljutunk a cél szintaxisig, ugyanis felsorolhatjuk a megfelelő szignál típusokat egy listába, és elég arra vizsgálnunk, hogy az adott szignál típus része-e a listának.

A legutolsó megoldás tekinthető a legelőnyösebbnek, tartalmazza a második előnyeit, azonban a hátrányát sikerült kiküszöbölni.



### 3.4.3. Szolgáltatott és elvárt interfészek megkülönböztetése

#### UML-ben

Az UML szabvány szerint egy *konnektor* csak olyan portok között húzódhat, melynek a szolgáltatott és elvárt interfészei kompatibilisek egymással. A port egy kommunikációs csomópontot reprezentál, a típusa azonban adattag révén tetszőleges lehet. A modellosztály számára a port szolgáltatásokat nyújt a szolgáltatott interfész által, így a típusa megfelel a szolgáltatott interfésznek. A modellosztály azonban kifele áramoltathat a porton keresztül információkat, az elvárt interfész által. Húzzunk be egy *Using* reláció a port és az elvárt interfész között, mivel a modellosztály a portot használja üzenetáramoltatásra. A probléma, hogy a *Using* relációk nem kölcsönösen egyértelműek, ha két portnak ugyanaz a szolgáltatott interfésze, így érdemes minden portnak egy saját interfészt generálni, amely leszarmazottja a szolgáltatott interfésznek. Így minden port típusa egyedi lesz, és a relációk is egyértelműek lesznek a modellben.

#### C++-ban

Mivel a portra tudunk üzentet küldeni, de fogadni is tudunk róla, így az interfészeknek két részük lesz, az egyik a fogadó, a másik a küldő műveleteket definiálja. Attól függően, hogy az adott interfészt szolgáltatott vagy elvárt interfésznek definiáltuk, a megfelelő részből fog leszarmazni az adott port. A másik lehetőség az lenne, ha nem bontjuk két részre az interfészt, helyette a recepciön függvényeknek egy plusz paramétere lenne, hogy kívülről vagy belülről küldünk-e üzentet. (Ahogy azt a 3.3 bekezdésben felleztük ) Azonban a portok esetén problémába ütköznénk, mert a validáció nem tudna különbséget tenni aközött, hogy egy portnak szolgáltatott vagy elvárt interfésze-e, melyet implementál.

Interfész kiterjesztetett szintaxisa, konkrét példa portok esetén: `class PingPongInterface` `class RequiredPart` `void send(Ping e); void send(Pong e);` `class ProvidedPart` `void receive(Ping); void receive(Pong);` `class Port {Provided, Required};` `public Required::RequiredPart, Provided::ProvidedPart` `PingPongPort {PingPongInterface, PingPongInterface};` `p;`

## 3.5. Konnektorok, összekapcsolás művelet reprezentálása

Az interfészeket (3.4) és portokat már tudjuk exportálni UML-ben, illetve megfelelő C++ reprezentációt is találtunk. A konnektorokat ezek segítségével már ki tudjuk fejezni.

Mivel *txtUML*-ben 1-1 kapcsolatokat tudunk csak leírni, így a mi megoldásainkat is erre korlátozzuk. Először megvizsgáljuk, mi lehet a szabványos reprezentáció UML-ben, majd kitérünk a különböző megoldásokra C++-ban.

Mielőtt még tovább megyünk, definiáljuk, pontosan mik ezek a konnektorok az UML szerint. A konnektorok alapvetően portokat kötnek össze, de többet fejez ki egy egyszerű asszociációnál. A konnektor egyfajta absztrakciót fejez ki két port között, leírja, milyen két port kapcsolódik össze, delegációval vagy assembly kapcsolattal vannak-e összekötve, illetve a kommunikáció prototípusát is képes leírni. A delegáció szülő-gyerek portok között jöhet létre, ilyenkor egy portra való üzenetküldést delegálunk a túloldalon lévő portra, mely tovább áramoltatja a belső komponens felé, vagy a másik irány esetén a külvilág felé, a komponensen kívülre. Assembly összekapcsolás esetén két egyenrangú enkapszuláció portja között jön létre kapcsolat, kölcsönös, oda-vissza kommunikációval. Fontos, hogy szülő-gyerek enkapszulációk között nem lehet Assembly kapcsolat, illetve egyenrangú szereplők között Delegáció.

### Konnektor reprezentáció UML-ben

A szabvány szerinti reprezentáció viszonylag értelemszerű, egy *Connector* UML elemet kell létre hozni, a végeknek megadni a portokat, beállítani, hogy Assembly vagy Delegált összekapcsolás van-e közöttük. A *Connector* elemnek van egy Asszociáció típusa, mivel a szabvány szerint a *Connector* egy Asszociáció egy konkrét realizációja. Hogy mi legyen ez a típus, azzal később foglalkozunk.

### Összekapcsolás művelet reprezentációja UML-ben

A *connect* akció kifejezése UML-ben már kevésbé egyértelmű, mivel *connect* akció nem létezik explicit a szabványban. Több lehetőséget is megvizsgáltunk, mire eljutottunk a helyes reprezentációhoz.

1. Mivel a Portok adattagok az UML-ben (Property-k) így két portot kölcsönösen értékül adhatunk egymásnak. Ez egy lehetőség lehet kifejezni az összekapcsolást, szabványos modell keletkezik, de mégsem ez a helyes szabványa Portok összekapcsolásának. A probléma ugyanis az lesz, hogy bármilyen két portot összekapcsolhatok így, konkrét konnektor realizáció nélkül.
2. Mivel a *Connector* egy asszociáció példány, így már egy konkrét, létrejött kapcsolatról beszélhetünk. Ebből adódóan a komponensek struktúrájából következtethetünk az összekapcsolt portokra. Azonban ez megszorításokkal járna, ha például egy szülőnek két azonos típusú gyereke van, de csak az egyikkel szeretne delegáció révén kommunikálni, nem köthetem hozzá mindkét gyerek porthoz a szülő portját.
3. Nem maradt más lehetőség, mint a dinamikus összekapcsolás, és szintaktikailag sincs már más lehetőségem, mint a *CreateLinkAction* használata. Ehhez két dologra van szükségem: Végpontokra, valamint arra az asszociációra, melyet használni szeretnék az összekapcsoláshoz. Mint korábban láttuk, a Konnektorok fel vannak típusozva egy Asszociációval. Értelemszerű lenne ezt fölhasználni az összekapcsoláshoz, a kérdés csak az, mi legyen ez az Asszociáció. Az Asszociáció típusok között húzható, a Port önmagában nem típus, azonban adattag révén rendelkezik típussal, a szolgáltatott interfészük típusával. Nem maradt más ésszerű lehetőségünk, mint ezek között az interfészek között létrehozni egy Asszociációt, melyet a Konnektor realizál a Portok összekapcsolásával. Így a *CreateLinkAction*-be be tudjuk tenni Asszociációként a Konnektor típusát, ami mentén összekapcsolunk. Ezzel egy szabványos UML-t generálunk, és a Konnektorokat sem hagyjuk figyelmen kívül két port összekapcsolásánál.

### 3.5.1. Összekapcsolt portok reprezentációja C++-ban

A szabvány szemantikája szerint, amikor egy üzenet küldünk a *send* művelettel egy adott portra, annak továbbítása kell ezt a vele kapcsolatban álló port felé. Ehhez mindenképpen valami referenciát kell tárolni a kapcsolat Portra. Ezt alapvetően kétféle képen érhetük el:

1. Egy globális adatszerkezetben, például egy asszociatív tömbben megmondjuk, melyik porthoz ki kapcsolódik. Ennek előnye, hogy nem a Port kódja semleges marad, nem kell belekódolni a kapcsolatokhoz. Hátránya az, hogy egy párhuzamos környezetben egy globális adatszerkezetet kell manipulálni, így sok szinkronizációt igényel a dolog, minek következtében csökken a hatékonyság
2. A másik lehetőségünk, hogy a port kódjában közvetlen referenciát tárolunk a kapcsolt portra. Mivel az nem lesz érdekes az üzenetküldés szemantikájában, hogy pontosan melyik Konnektoron keresztül kapcsolódnak az adott portok egymáshoz (annak típusa azonban számítani fog), és csak 1-1 kapcsolatok valósítunk meg, így a Port kódját nem kell a modell alapján generálnunk, nem fog bővülni, és hatékonyabb megoldást érhetünk el szekvenciális környezetben is, mint a globális adatszerkezettel.

Az üzenetküldés szemantikájához már megvan a referenciánk, azonban ez még nem elég önmagában. Figyelembe kell vennünk azt is, hogy milyen típusú Konnektoron keresztül kapcsolódtak az adott portok. Másképp kell átadni az üzenetet, ha Delegációról van szó, és másképp, ha Assembly kapcsolatról. Delegáció esetén, ha üzenetet küldünk kívülről egy adott portra (*send* művelet), akkor a szemantika szerint az üzenetküldésünk ekvivalens a kapcsolt portra való belső üzenetküldéssel. Assembly kapcsolat esetén azonban ez az üzenetküldés a kapcsolt portra való külső üzenetküldéssel lesz ekvivalens a műveletünk. (*receive* művelet). Így tudunk kell, hogy a referenciánk milyen kapcsolatban áll a portunkkal. Két lehetőségünk van ennek kifejezésére, a választás nem egyértelmű:

1. Virtualizáció használata, *Port* referencia helyett *Connection* referencia, melynek absztrakt művelet eldönti, a kapcsolt port melyik műveletét kell meghívni. Ez egy tisztább implementáció, azonban a virtualizáció nem elég hatékony.
2. Delegáció jelző használata: Marad a Port referencia, maga a Port dönti el, melyik művelet hívására van szükség a jelző alapján. Ez a megoldás kevésbé letisztult, de hatékonyabb.

Vizsgáljuk meg az üzenet fogadás szemantikáját is. A fogadás művelet végrehajtási szemantikája attól függ, hogy állapotgéppel összekapcsolt Portrol (Behavior Port) beszélünk-e vagy nem. Előbbi esetén egy objektum állapotgépéhez futnak be az üzenetek, a másik esetben pedig a szülő komponens portja delegálja az üzenetet a gyerek klasszifikáció portjára. Ennek megvalósítására szintén a fent említett két lehetőségünk van, a virtualizáció, illetve a behavior jelző. Egy *Behavior Port* esetén nem csak a *receive* művelet törzse különbözik, de más-más adattagokkal dolgozik a két port. Állapotgép összekapcsolás esetén referenciát kell szereznünk az állapotgépet birtokló objektumra, ellenkező esetben pedig a gyerek komponens portjára. Ez különböző adattagokat igényel, az adat redundancia miatt leszámazás egy kézen fekvő választás. Unio típus segítségével azonban el lehetne kerülni a redundáns tárolást, ami egy hatékonyabb, de nehezebben olvasható kódot eredményez.

Hasonlóan az interfészekhez, a kapcsolatok esetén is a kevert megoldás lenne a leghatékonyabb. Ha dolgozhatnánk leszámazással, ugyanakkor nem kéne virtualizációt használni. Erre egy kézenfekvő megoldás az lenne, ha a leszámazáskor megadnánk az interfésznek, ki a leszámazottja.

```
class DelegateConnection: public Connection, DelegateConnection;
```

Majd definiálnánk a speciális implementációkat, az interfész implementáció pedig statikus konverzióval a specializált *Connection* osztályra konvertálná önmagát, és meghívna a megfelelő műveletet. Ehhez fordítási időben tudnom kell, hogy egy adott porthoz milyen típusú kapcsolaton keresztül fog létrejönni a referencia.

A portokat akkor kapcsoljuk össze, mikor létre hozzuk a rendszer struktúráját. Minden port 1-1 kapcsolattal kapcsolódik egymáshoz. Ami azt jelenti, hogy egy állapotgéppel összekapcsolt port esetén egy referenciák lesz egy másik portra, melyet valamilyen ismert kapcsolaton keresztül kötünk majd hozzá. Általános port esetén pedig két referenciánk lesz, az egyik biztosan egy delegáció, a másik pedig ugyanazon az érvek mentén ismert, amit az előbb tárgyaltunk. Ebből következtetve úgy tűnik, hogy ha a pontos összekapcsolások nem is ismertek, a potenciális összekapcsolások típusai igen, így fordítási időben el tudjuk dönteni egy adott kapcsolatról, milyen típusú.

### 3.5.2. Konnektorok reprezentálása C++-ban

A konnektorok reprezentálása is több lehetőségünk van, ezeket fogjuk most megvizsgálni

1. Itt is felmerül a lehetőség, hogy teljesen figyelmen kívül hagyjuk a konnektorokat. Ezt akkor tehetjük meg egyértelműen, ha a referencia birtoklását választjuk a fentiekben az összekapcsolt portok reprezentálásánál. Ennek előnye szintén az egyszerűség lesz, a generált kód mérete kisebb lesz. Hátránya pedig a validáció teljes hiánya. Az interfészeknél arra jutottunk, hogy szükséges a validáció, így ez nem a legjobb megoldás
2. A másik lehetőségünk bevezetni egy *Connector* osztályt, és összekapcsolásánál erre az osztályra hivatkozunk. Ebben a struktúrában leírhatnánk a portokat, melyek potenciálisan kapcsolódnak egymáshoz, és ezt használhatjuk validációra. Ennek előnye egy tiszta reprezentáció lesz, hátránya viszont az, hogy egy redundáns megoldást kapunk, a konnektoroknak semmi szerepük nincsen a validáción kívül.
3. A Konnektorokhoz generált asszociáció típust használjuk fel a validációra. Ennek előnye, hogy az asszociációkat egyébként is eltengethetetlen generálunk a modellben, így nem igényel további kódgenerálást. Azonban a típusvalidációról gondoskodik, nem enged összekapcsolni olyan két portot, melyhez nem tartozik megfelelő típusú asszociáció. Hátránya, hogy nem elég konkrét, nem kell megfelelni a konkrét portoknak a struktúrán belül, elég, ha az interfészeik megfelelnek.
4. TODO: Ha azt a megoldást vesszük alapul, hogy az egyes végpontokba bele tudjuk kódolni a Konnektor típusát, akkor akár konkrétabbak is lehetünk, és a megfelelő Konnektort és szerepeket is bele tudjuk kódolni.

## 3.6. Kommunikáció megvalósítása C++-ban

Végül még érdemes néhány szót ejteni a kommunikációs szemantikáról. Ennek nagy részét már tisztáztuk. Megvizsgáltuk a portra való üzenetküldés szemantikát az enkapszuláción belülről. Ekkor az üzentet kifelé áramoltatjuk, egy másik

port fogja megkapni. Ebben a fejezetben csak az lesz az érdekes, hogy az üzenetáram végén, amikor eljut az üzenet egy állapotgéphez, hogyan fogjuk kezelni. Így csak az állapotgéppel összekapcsolt portok fogadó (*receive*) műveletei lesznek érdekesek számunkra. A port tartalmaz egy referenciát az állapotgépre, így tudja neki továbbítani az üzenetet. Azonban a nyers továbbítás kevés lenne, mivel egy objektum állapotgépe egy adott porton történő állapotváltozásra kell reagáljon. Így az üzenet feldolgozásakor tisztában kell lennünk azzal, melyik porttól kaptuk meg az adott üzenetet. Az UML szabványban egy adott átmenetnél felsorolhatjuk az adott portokat, melyről az adott üzenetet várjuk. *txtUML*-ben csak egy portot adhatunk meg, vagy jelezhetjük, hogy az adott üzenet bárhonnan érkezhetsen, porton kívülről is, vagy bármely portról. Ennek megfelelően megfelelően mindenképpen a meglévő állapot-átmenet táblát érdemes kibővíteni egy új dimenzióval: Megadni, hogy az átmenetben szereplő üzenet potenciálisan mely portokról jöhet, hogy az állapotgép reagáljon rá.

Eddig esemény-állapot táblánk volt, mely megadta, melyik átmenetnek kell végrehajtódnia. Ezt kibővíthetjük az alábbi módokon:

1. Egy darab port típus, és állapot-esemény-port hármasokat fog tartalmazni az átmenet táblánk. Mielőtt egy port továbbítja az állapotgépnek az üzenetet, beállítja az eseményen a port infót, így tudni fogjuk, honnan érkezett. Ez az esetünkben jól fog működni, mivel csak egyféle portról jöhet az üzenet. Azonban, ha azt adjuk meg, hogy bármelyik portról érkezhetsen az üzenet, már gondba leszünk, mivel ilyenkor nincs megadva port dimenzió a táblában, az üzenetben viszont igen. Ezt speciálisan le tudjuk kezelni az üzenetfeldolgozás során, azonban nem elég általános megoldás.
2. Általánosabb megoldás, ha a port helyett megadhatnánk egy maskban azokat a portokat, melyek üzeneteit megfigyeli az adott átmenet. Mivel egy üzenet továbbra is csak egy helyről érkezhetsen, így elég lenne csak azt vizsgálnunk, hogy az állapot-esemény alapján megkeresett átmenet portjai között szerepel-e a az a port, melyről az üzenet érkezett. Fontos, hogy itt a Port dimenzió a táblázat másik végén jelenik meg. A táblából, hasonlóan az eddigiekhez, állapot-esemény pár alapján keresünk, melyből megkapjuk az átmenet függvénymutatót, az őrfeltételt, valamint a megfigyelt portokat. Ekkor azt az

átmenetet hajtjuk végre, melynek portjai között szerepel az üzenet portja vagy a *NoPort* speciális típus. *AnyPort* esetén az összes lehetséges Portot belerakjuk az átmenetbe, valamint a *NoPort* típust is.



## 4. fejezet

# Program dokumentáció

A dolgozatban nem csak elemeztük a lehetséges megoldásokat, hanem kiválasztottunk egy lehetséges megoldást, melyet leprogramozva demonstrálni tudjuk a dolgozat eredményeit. A továbbiakban ennek a demonstrációs programnak a rövid felhasználói és fejlesztői dokumentációja következik.

### 4.1. Felhasználói dokumentáció

A dolgozat a *txtUML* keretrendszerre épül, melynek részletes felhasználói dokumentációja megtalálható az alábbi linken: <http://txtuml.inf.elte.hu/wiki/doku.php>

A dolgozat szempontjából a Portok, Interfészek, Konnektorok leírása a legfontosabb, valamint az exportálás és fordítás.

#### 4.1.1. Szükséges eszközök

Az exportáláshoz az alábbi eszközökre van szükség:

- Java
- Eclipse IDE az export dialog eléréshez, minden szükséges függőség letöltődik a txtUML pluginnal

A fordításhoz az alábbiakra van szükségünk

- Cmake segítségével legenerálhatjuk a számunkra tetsző fordítási környezetet

- GCC vagy Clang
- A generálandó fordítási környezet. (MinGW, Ninja, MSVC, stb..)

### 4.1.2. Generálás és futtatás

Kódot generálni legegyszerűbben Eclipse-ből tudunk egy txtUML modell megírása és egy kihelyezési konfiguráció alapján. A *runtime* mappában az előre megírt fájlokat másoljuk, mely tartalmazza többek között a Portok osztályát is, az összekacsoló műveleteket, és azok szemantikáját. A konkrét port példányokat, konnektorokat, interfészeket pedig a *runtime* mappán kívül generáljuk a modell alapján.

A generált kódot fordítani a *cmake*, valamint a választott build környezet parancsával tudunk. A *cmake* parancs elhagyható, ha az exporter dialóguson kiválasztjuk a megfelelő környezetet. Fordítás után pedig már futtathatjuk a *main* futtatható állományunkat.

## 4.2. Fejlesztői dokumentáció

A program a *txtUML* keretrendszer C++ fordító és exporter komponensét fejleszti tovább, ennél fogja Java nyelven készült. A Java modellt JDT segítségével járjuk be, és az EMF keretrendszert használjuk a szabványos modell létrehozásához.

### 4.2.1. Szerkezeti felépítés

A program négy jól elkülöníthető részből áll, mindhárom részt ki kellett egészíteni a Portok, Interfészek, Konnektorok exportálásához

1. A Java modellt UML modellre leképező algoritmusok. Ezek nem tiszta Java-ban, hanem a Xtend keretrendszerben készültek. A strukturális elemekhez fel kellett venni a **PortExporter.xtend**, **InterfaceExporter.xtend**, **ConnectorExorter.xtend** és a **ConnectorEndExporter.xtend** fájlokat. Az akcióelemekhez pedig felvettük a port referenciát kiolvasó **PortReference-Exporter.xtend** fájlt, az üzenetküldéshez a **SendToPortActionExport-**

**er.xtend**, az összekapcsolásokhoz a **AssemblyConnect.xtend** valamint a **DelegateConnect.xtend** fájlokat.

2. Az UML modellt feldolgozó algoritmusok. Itt hozzá kellett adni a **PortExporter.java** és az **InterfaceExporter.java** osztályokat, melyek ezek struktúrájának az exportálásáért felelnek. A *send* és a *connect* műveletek exportálása végett hozzá kellett nyúlni az **ActivityExporter.java** osztályhoz, mely az akciókódok exportálásáért felelős.
3. C++ kódgenerálási sablonok: Itt bevezettük a **InterfaceTemplates.java** és **PortTemplates.java** fájlokat a strukturális elemek sablonjához. Az **AssociationTemplates.java** az összekapcsoláshoz szükséges sablonokról gondoskodik.
4. Előre megírt C++ fájlok: Itt lényegében a **PortUtils.hpp** fájl tartalmaz minden portokhoz szükséges előre megírt elemet: A portok típusát, a kapcsolatok, az összekapcsolásokat. Az **IEvent.hpp**-be is bele kellett nyúlni az eseményfeldolgozás megváltozása miatt. Valamint az **InterfaceUtils.hpp** tartalmazza az interfészek sablonját és az üres interfészt, a **AssociationUtils.hpp** pedig az összekapcsoláshoz szükséges asszociációk sablonját.

Ezek a modulok jól eltönkretehetőek és cserélhetőek.

#### 4.2.2. Tesztelés

A demonstrációs program néhány tesztelését szedjük össze az alábbi fejezetben.

1. Regresszió tesztelése: Portok nélküli modell létrehozása, melyben azt várjuk, hogy az eredmény maradjon az, ami eddig volt.
2. Egyosztályos modell létrehozása egy darab porttal
  - a) Üres interfészekkel, *send* művelet kipróbálása, mely hibás lesz
  - b) Nem-üres interfésszel, *send* művelet kipróbálása olyan szignállal, melyet tartalmaz az interfész. Nincs fordítási hiba, de nincs hatása sem. Majd egy olyannal, mely nem része az interfésznek.

Az osztály egy átmenete tartalmazza az adott portot. *receive* művelet kipróbálása.

a) Kompatibilis interfészekkel hozzuk létre őket, delegációval összekapcsoljuk őket, majd *send* művelet kipróbálása a belső komponensen. Ezután a *receive* művelet a külső komponensen.

c) Inkompatibilis interfészekkel hozzuk létre őket, majd delegációval megpróbáljuk összekapcsolni, mely szintén fordítási hibát jelez.