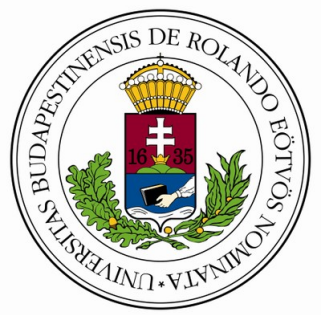


Eötvös Loránd Tudományegyetem
Informatikai Kar
Fordítóprogramok és Programozási nyelvek

Komponens-alapú UML modellek fordításának vizsgálata

Dévai Gergely
adjunktus

Nagy András
Programtervező Informatikus MSc

Budapest, 2019

Tartalomjegyzék

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. Az UML [1], ezen belül is a végrehjatjható 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 közvetlenül 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 tervezésnél fontos szerepet játszanak a portok és konnektorok [2]

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 kitérünk a szabványos UML leképezésre, annak problémáira is.

1.1. txtUML keretrendszer

A diplomamunka a *txtUML* keretrendszerben készült, mely egy szöveges, végrehajtható és fordítható modellező eszköz. A project célja a UML modellezés megkönnyítése azáltal, hogy a felhasználó nem grafikusan, hanem egy jól definiált, Java szerű nyelvben írhat le modelleket, melyek a végrehajthatóság kapcsán helyben verifikálhatóak, valamint szabványos UML exportálás révén képes együtt működni más eszközökkel is. A projekt részét képezi még egy vizualizációs eszköz, valamint egy C++ fordító komponens.

1.2. A diplomamunka célja

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. A diplomamunka tehát egy korábbi diplomamunkát fejleszt tovább[4], mely megalapozta a C++ generátort, valamint egy szakdolgozatot [5], mely konfigurálható párhuzamosítási lehetőségekről, valamint az akciókódok exportálásáról szól. A diplomamunka részben ezt fejleszti tovább, kiszélesíti a strukturális és akciókódok támogatását a kompozíciós elemekre. Az UML kompozíciós szabvány bonyolultsága miatt felmerül az igény a szabvány alapos elemzésére, a megfelelő reprezentációk megtervezésére, a kódgenerálási stratégiák elemzésére, összehasonlítására. Portok révén lehetőséget teremt az egyes osztályok teljes izolációára, így jobb párhuzamosítási lehetőségeket tesz lehetővé.

1.3. A diplomamunka főbb eredményei

A diplomamunka széles körben átjárja az UML kompozíciós elemeket, azok precíz szabványát és C++ kódgenerálását. Ennek megfelelően az alábbi főbb eredményeket lehet felsorolni:

– Az UML szabvány bonyolultsága miatt érthető magyarázatot ad a kompozíciós modellezés eszköztárára.

– Elemzi a szabványt, hogy a lefordítás elég precíz legyen.

– Áttekintést ad hasonló céllal készült keretrendszerekről.

– Lehetőségeket vizsgál meg és értékel ki a C++-ra fordítás tervezési döntéseivel kapcsolatban.

– Az elemzés eredményeinek implementációval történő validációja.

1.4. UML Komponens fogalmak

1.4.1. Strukturális elemek

Egységbe zárt elem

UML-ben az egységbe zárt elem egy olyan absztrakt fogalom,mely egyfajta mechanizmust fejez ki, mellyel képesek vagyunk elszigetelni az objektumot a külvilágtól portok segítségével.
Ez a fogalom nem összekeverendő az UML komponenssel, mely egy speciális fogalom az UML-ben: egy egységbe zárt elem, és több logikai egységet elemet fog össze (mely lehet egy osztály vagy egy másik komponens), egy jól definiált interfésszel rendelkezik, mellyel valamilyen szolgáltatást nyújt a külvilág felé. A komponens fontos tulajdonsága, hogy az adott rendszerben könnyen cserélhető, azonban ez bármely egységbe zárt elemre igaz, így a komponensek nincs különösebb jelentőségük, a továbbiakban nem is foglalkozunk velük.
Az UML osztályok is egységbe zárt elemnek számítanak, a diplomamunka az osztályok portjainak exportálását járja körül, így mostantól egységbe zárt elemek helyett osztályokról beszélünk.

UML Interfész

UML-ben az interfész egy olyan elem, melynek a szokványos operációs műveleteken kívül speciális kommunikációs műveletei, úgynevezett fogadó végpontjai vannak.

Ezek olyan speciális műveletek, melyek egy eseményt várnak, tehát egy interfész leírja, milyen eseményekkel kommunikálhatunk az interfészeken keresztül. (Az idegen nyelvű szakirodalom reception-nek szokta nevezni)

UML Port

Osztályok 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 üzenteteket 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.

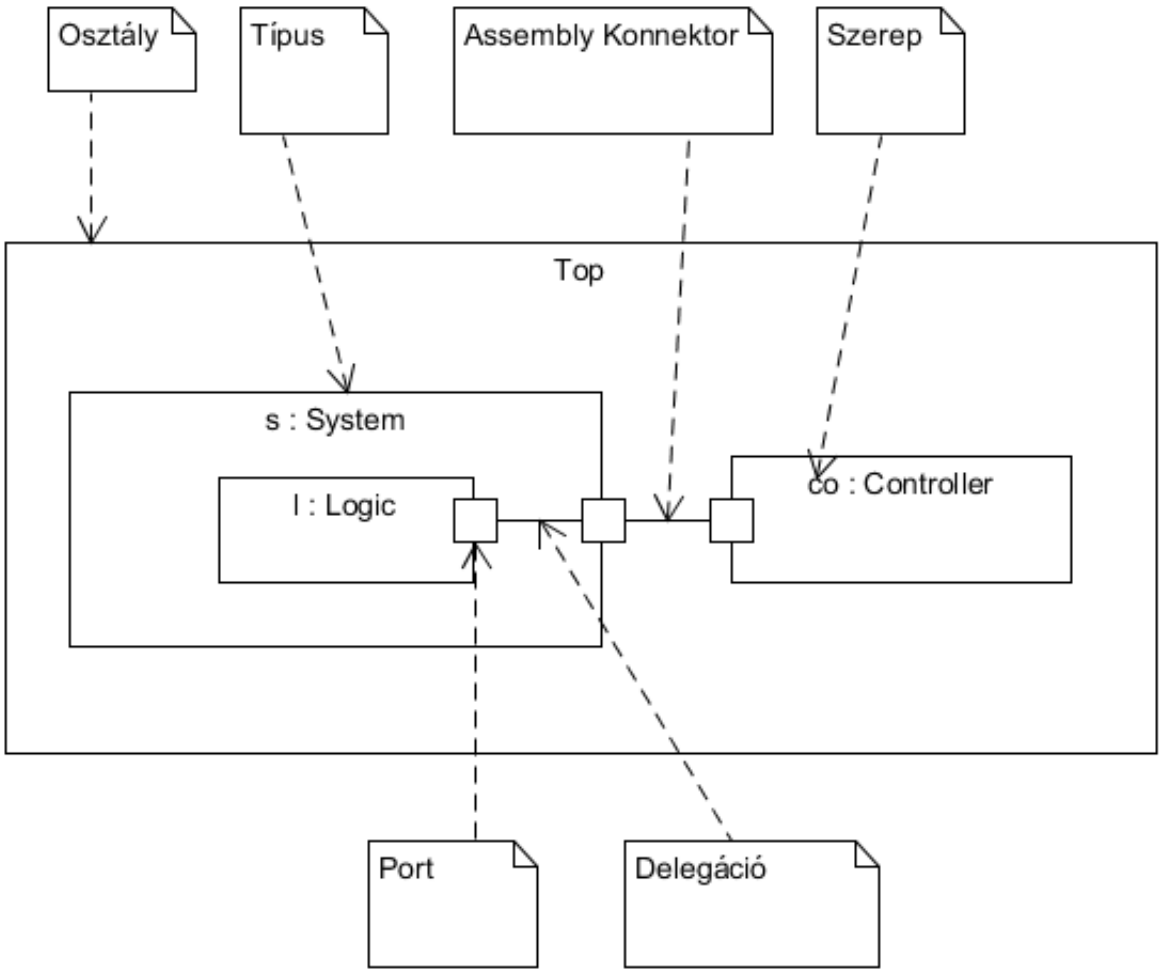
Kompozit struktúra

UML-ben a kompozíció erős tartalmazást jelent, mely azt jelenti, hogy az osztályoknak szerves részei lehetnek további elemek. Ez azt jelenti, hogy az élettartamuk összefügg, a szülő elem hozza létre a gyerek elemet, és ő szünteti meg. Egy kompozícióban mindig van pontosan egy tartalmazó, és véges számú tartalmazott egy adott osztályból, melynek minden tartalmazott objektumához külön szerepet rendelhetünk.

UML Konnektor

Egy asszociációt realizál két port között, melyeknek a tartalmazó osztályai valamilyen szereppel fel vannak tüntetve egy kompozit struktúrában. A konnektornak meg kell adni, milyen két szereplőt köt össze, milyen szerepekkel, a szereplők mely portján keresztül jön létre a kapcsolat. A Konnektoroknak kétféle fajtája van, Assembly vagy Delegációs kapcsolat. A különbség abban van, hogy egy Assembly kapcsolat két azonos szinten lévő szereplőt köt össze a kompozit struktúrában, míg a Delegáció a szülőt köti össze egy gyerek osztállyal a kompozit hierarchia mentén. Interfész portok esetén Assembly kapcsolódás olyan portok között lehet, melyek interfészei egymás inverzei (ami az egyiknek a szolgáltatott interfésze, az a másiknak

az elvárt), delegáció esetén pedig az interfészeknek meg kell egyeznie (az elvárt és a szolgáltatott interfészek megegyeznek). Leírhatjuk benne továbbá a kommunikációra használt protokollt.



1.4.2. Akció elemek

Üzenet küldés/fogadás

Interfész portok esetén az elvárt interfész leírja, hogy belülről milyen üzeneteket tudunk küldeni a portnak, mely a külvilág felé áramlik tovább, ezt nevezzük portra való üzenet küldésnek.

A szolgáltatott interfész pedig azt mondja meg, milyen üzeneteket tudunk fogadni az adott portról, tehát azt, hogy az osztályon kívülről milyen üzeneteket tudunk fogadni az adott portján. Ezt nevezzük üzenet fogadásnak.

Connect művelet

A *connect* művelet alapvetően nem létezik UML-ben, *txtUML*-ben azonban ezt használjuk két port összekötésére a konnektoron keresztül. Ahogy később látni fogjuk, a *CreateLink* műveletet fogjuk használni ugyanerre a célra, mivel a szabvány szerint dinamikusan csak ezen művelet mentén tudunk összekötni két elemet.

1.5. A txtUML kompozit elemei

1.5.1. Interfészek megadása

```
public interface ExampleInterface extends Interface {
  public void reception(ExampleSignal1 signal);
  public void reception(ExampleSignal2 signal);
}
```

Ebben az esetben az *ExampleSignal1* és az *ExampleSignal2* eseményeket képes fogadni az *ExampleInterface* a fogadó végpontjain keresztül.

1.5.2. Portok felvétele

```
public class ExamplePort extends Port<Interface1,Interface2> {}
```

Állapotgéppel összekapcsolt portok:

```
@BehaviorPort
public class ExamplePort extends Port<Interface1,Interface2> {}
```

Ebben az esetben az *ExamplePort* interfészei az *Interface1* és az *Interface2*

1.5.3. Konnektorok

Konnektor

Mivel a portok kompozíciók szerepei között húzódnak, nézzük meg, hogyan néz ki egy kompozíció leírása:

```
class ExampleComposition extends Composition {
  class a extends ContainerEnd<A> {}
  class b extends End<One<B>> {}
```

```
}
```

Itt *A* a tartalmazó osztály *a* szereppel van feltöltetve a kompozícióban, aki tartalmaz egy *B* típusú osztályt *b* szereppel.

Assembly konnektor

A-nak legyen két része: *Comp1.b* egy *B* típusú része és *Comp2.c* egy *C* típusú. Legyen *P* *B*-nek egy portja és *Q* *C*-nek egy portja. Ekkor összeköthetjük *P*-t *b* és *Q* *b* és textitc szerepek mentén egy assembly konnektor segítségével.

```
public class ExampleConnector extends Connector {
  public class connEnd1 extends One<Comp1.b, B.P> {}
  public class connEnd2 extends One<Comp2.c, C.Q> {}
}
```

Delegációs konnektor

Ha *Comp3.d* egy*D* típusú része *A*-nak, *A*-nak a szerepe *Comp3.a*, *R* *A*-nak egy portja, *S* *D*-nek egy portja, akkor egy delegációs konnektor *R* és *S* között a következőképpen nézhet ki:

```
public class ExampleDelegation extends Delegation {
  public class connEnd1 extends One<Comp3.a, A.R> {}
  public class connEnd2 extends One<Comp3.d, D.S> {}
}
```

Átmenetek kibővítése

Az állapotátmeneteket *Trigger*-ét kibővítettük egy *port* dimenzióval, mely azt mondja meg, hogy az *value* értéken szerepelő üzenetnek melyik port legyen a forrása, hogy az átmenet végbemenjen.

```
@From(State1.class) @To(State2.class)
@Trigger(port = ExamplePort.class, value = ExampleSignal.class)
class ExampleTransition extends Transition {}
```

Ez esetben, ha *State1*-ben vagyunk, és az állapotgépnek küldjünk egy *ExampleSignal*-t, akkor, ha ennek forrása a *ExamplePort*, átlépünk a *State2* állapotba.

2. fejezet

Áttekintés

2.1. Szabványok és keretrendszerek áttekintése

2.1.1. fUML - ALF

A Foundational UML (fUML) [7] egy Object Managment Group (OMG) által specifikált végrehajtató UML szabvány, melynek szöveges akciónyelvi szintaxisát az ALF (Action Language for Foundational UML) [6] 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.

2.1.2. PSCS szabvány

A Precise Semantics of UML Composite Structures (PSCS) szabvány [8] szintén az OMG által specifikált UML szabvány, mely kiegészítve az fUML szabványt, a kompozit struktúrák elemihez definiál szemantikát. Ebben található meg többek között az osztályok, interfészek pontos szerepeinek leírása a kompozit struktúrákban, valamint a portok, konnektorok és ahhoz kapcsolódó akciókódok szemantikája. A PSCS szabvány szorosan kapcsolódik a 3 fejezethez, melyben megvizsgáljuk az egyes kompozit elemek UML szabványát a megfelelő modell létrehozásához, valamint a kódgenerálás szemantikájához.

2.1.3. Umple

Az Umple [9] szöveges alapú modellezőeszköz támogatja a komponens struktúrák leírását. Itt a portok egyszerű adattagonként jelennek meg, melyekhez úgynevezett aktívator metódusokat rendelhetünk, melynek hatása az attribútum változásakor érvényesül. A megváltozott állapot kihat a kapcsolatban álló portokra is, a konnektor másik végén álló portok állapota is meg változni, melyre szintén egy aktivátor függvény segítségével reagálhatunk. A konnektorok pedig egyszerűen portok egymáshoz kötésével jelennek meg (binding), konnektor struktúrák és 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. Valamint saját akciónyelvvel sem rendelkezik, hanem azt a célnyelvben adhatjuk meg.

Előnyei:

- Online eszközként is elérhető, nem szükséges hozzá semmit telepíteni.
- Letisztult, jól definiált szöveges szintaxisa van, beleértve a portokat is.
- Tartalmaz kódgenerálást

Hátrányai:

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

2.1.4. StartUML

A 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 portoka, de az azokkal való műveleteket már nem.
- Nehézkes, vizuális használat, az elemeket szabadon létrehozhatjuk, és könnyen nem szabványos modellt generálhatunk, míg a txtUML exportja a szabványos modellt exportálás révén hozza létre, így kisebb lehetőségünk van hibázni.

2.1.5. BrigePoint UML

A StartUML-hez hasonlóan az UML szabványos elemeiből építhetünk fel vizuálisan egy szabványos UML diagramot, melyet kiegészíthetünk akár nem szabványos elemekkel is. Azonban nem tudunk osztályok között portokkal kommunikálni, csak komponensek között (melyről a 1.4.1 fejeztben azt mondtuk, hogy egy kifejezetten speciális elem az UML-ben), 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 konnektorrá. Hasonlóan a StartUML-hez, az akciókat aktivitás node-ok révén kézzel kell összerakni, így nehéz szabványos port műveleteket létrehozni. Összességében

a BridgePoint-ról [10] is elmondható, hogy kezdetlegesen támogatja a portokkal történő kommunikációt,az ezzel kapcsolatos részei eltérnek a szabványos UML-től, így a kódgenerálás támogatása ellenére nehéz érdemben kódgenerálási stratégiáról beszélni.

2.2. 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. Ennek oka a szabvány nehezen érthetőségében és hiányosságában keresendő. 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 exportáláshoz, 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 felellethető módszerek finomításából és több különböző ötlet elemzésének következtében hozunk létre.

A továbbiakban a következő elemek reprezentációjával foglalkozunk:

- Interfészek
- Portok és üzenet küldés/fogadás művelet
- Konnektorok és connect művelet

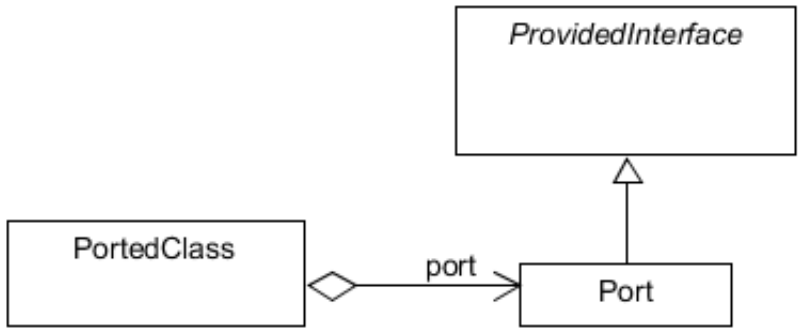
3.2. Portok reprezentálása

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 feltípusozni.

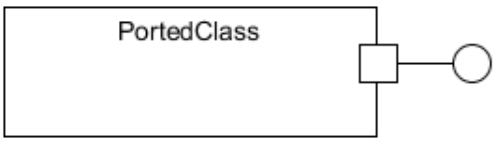
Szemléltetés osztálydiagrammal

A diagramon láthatjuk, hogy a port leszármazik a szolgáltatott interfészből, megvalósítva azt, és az osztály pedig tartalmazza a Portot.



Szemléltetés komponens diagrammal

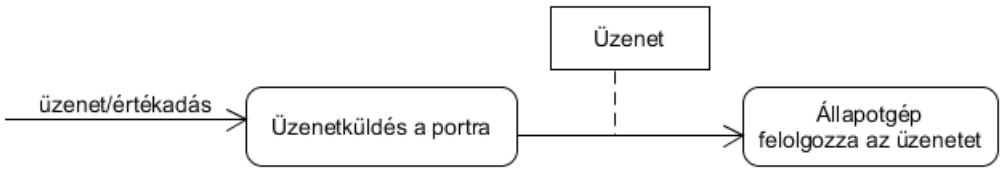
Az ábrán egy osztály látható, melynek a példához hasonlóan van egy portja, melyet kompozit struktúrákon egy négyzet jelöl. Nincs kapcsolatban senkivel, de egy interfész szabványa, melyet a kör jelöl, ez a kapcsolódási pontja.



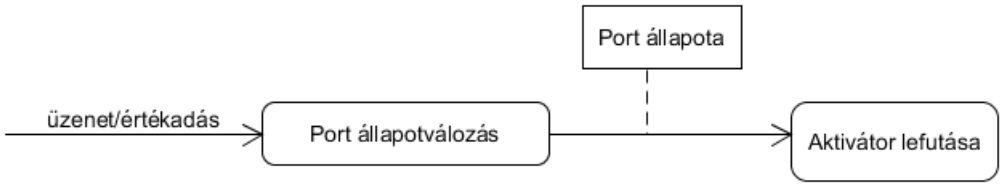
3.2.2. C++-ban

A portokat minden esetben adattagként reprezentáljuk, ezt mi sem tudjuk másképp megtenni. Többféleképpen 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 történű üzenetáramlást is átmeneti függvényekkel dolgozzuk fel. Ez esetben igazán nem érdekes számunka a port állapota, az üzenet az objektum üzenetsorába kerül be, mely majd feldolgozására kerül hasonlóan, mint egy normál üzenet.



– Külön aktivátor függvényeket valósítunk meg az osztályon belül az egyes portokra, melyek a portok állapotváltozásakor aktiválódnak. Ez esetben számít a portnak az állapota, mivel az reprezentálja magát az üzenetet. Ennek megfelelően máshogy dolgozzuk fel, mint az a többi üzenetküldést. Interfész portok esetén a port tartalmazza a neki küldött üzenetet, az reprezentálja az állapotát, primitív esetben pedig a port értéke szolgál erre.



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

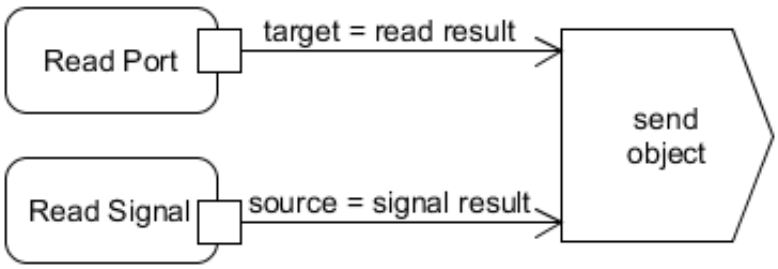
A *SendObjectAction* két paraméterből áll, egy célobjektumból és egy üzenetből. A célobjektum valamilyen *InputPin*, vagyis valamilyen akciónak az eredménye. Mivel a port egy strukturális elem, így a referenciáját egy *ReadStructuralFeatureAction*-nal megszerezhetjük.

– Könnyű reprezentálni, illeszkedik a modellfordító elvébe, aki feltételezi, hogy

egy *send* akciónak van egy cél objektuma, ahova az üzentet kell küldeni.

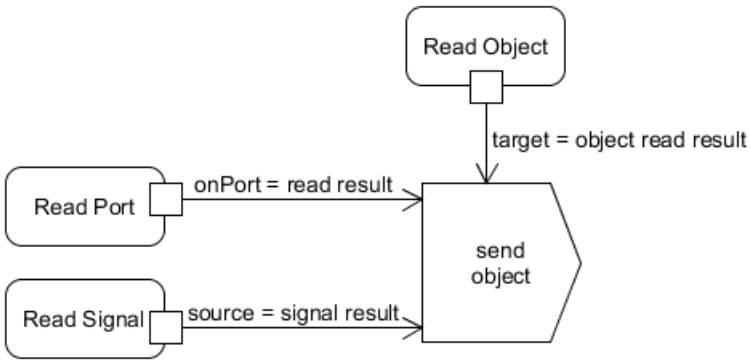
– Nem kell extra logikát beiktatni a C++ fordítóba az üzenetküldés terén.

– Szintaktailag könnyen kivitelezhető, azonban a PSCS szabvány nem ír róla semmit, mi lesz a szemantikája egy ilyen konstrukciónak.



A PSCS szabvány definiálja, hogy szintaktikailag hogyan néz ki egy üzenet küldése, illetve fogadása porton keresztül, és mi lesz ennek a pontos szemantikája. Ha egy osztály példánynak a portján keresztül szeretnénk üzenetet küldeni, akkor a *SendObjectAction*-nak a az *OnPort* adattagját használhatjuk a szabvány szerint. Az *OnPort* adattag egy *Port* típusú referencia adattag, melyet ha nem adunk meg, akkor az üzenetet a célobjektumnak küldjük el, és az objektumnak való üzenetküldés szemantikája nem változik. Azonban, ha ez ki van töltve, akkor az objektumnak lényegében a portján keresztül továbbítunk üzenetet. Ennek szemantikája pedig attól fog függeni, hogy milyen kontextusban hajtottuk végre az akciót.

Ha az akció környezete szerint az üzenetküldés kezdeményezője az az objektum, akinek a portjára üzenetet szeretnék küldeni, vagyis megegyezik a *target* referenciával, akkor üzenet küldésről beszélhetünk.
Ha a *SendObjectAction* kezdeményezése a cél objektumon kívülről történt, akkor pedig üzenet fogadásról beszélhetünk.



3.3.2. C++-ban

C++ exportálás során magának az üzenet küldés fordítását át kell dolgozni, figyelembe kell venni az *OnPort* referenciát, valamint az üzenetküldés kezdeményezőjét a szemantikának megfelelően. *txtUML*-ben az üzenet fogadás nem megvalósított.

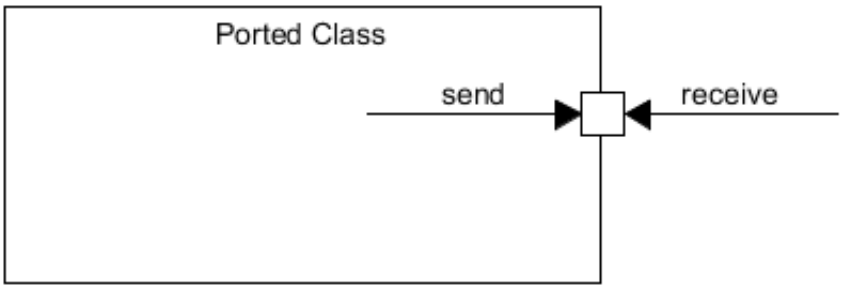
Üzenet küldés

Az üzenet küldés C++-ban egy *send* műveletre fog fordulni, melynek első paramétere az port, a második pedig a küldendő üzenet. A szintaxisa így lesz a legtisztább, inkább ennek a szemantikáját érdemes megvizsgálni. A *send* művelet azt jelenti, hogy a Portra teszünk egy üzenetet, melyet továbbítunk egy másik egység felé konnektoron keresztül. Ennek szemantikájával 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 lehetősünk van: Vagy egy teljesen új függvényt vezetünk be (nevezzük ezt *receive*-nek), vagy paraméterezzük a *send* függvényünket egy környezet változóval, melyben megadhatjuk az üzenetküldés kontextusát a PSCS szabványhoz hasonlóan. Előbbi megoldás azonban tisztább, jobban érhető a felhasználó számára, valamint a szolgáltatott és elvárt interfészek kifejezésénél is előnyösebb lesz. (3.4)

A következő ábra azt szemlélteti, hogy melyik akciót kell lefuttatni attól függően, hogy a portra való üzenetküldés milyen kontextusban történt, melyik irányból.



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. fogadó végpontokat. Egy-egy végpont egy olyan metódusnak felel meg, melynek pontosan egy paramétere van, egy *Singal* típusú paraméter.

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 az interfészek exportálásával C++-ban.

– Alacsony szintű, hatékonysági kérdésekben a legjobb megoldás, mivel se futás idejű se fordítás idejű validáció nincs.

Hátrányok:

– Külső osztályok, portok esetén a felhasználó sem tudunk felvenni interfészeket, ilyenkor már nem hagyatkozhatunk a forrásnyelv validációjára, könnyen hibázhatunk.

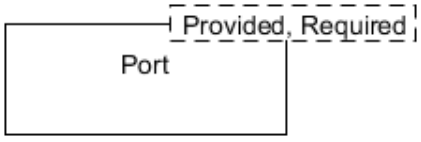
– Ha nem is vezetünk be új komponenst, külső kommunikáció esetén tetszőleges üzentet küldhetünk a portra, ami szintén hibás lehet.

– A generált kód kevésbé lesz érthető

– Az interfészeket sehol máshol sem tudjuk felhasználni, osztályok sem tudják megvalósítani őket, esetén sem, mely szintén a modellel való kommunikációban jelent hátrányt, illetve annak későbbi kiterjesztésében.

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.

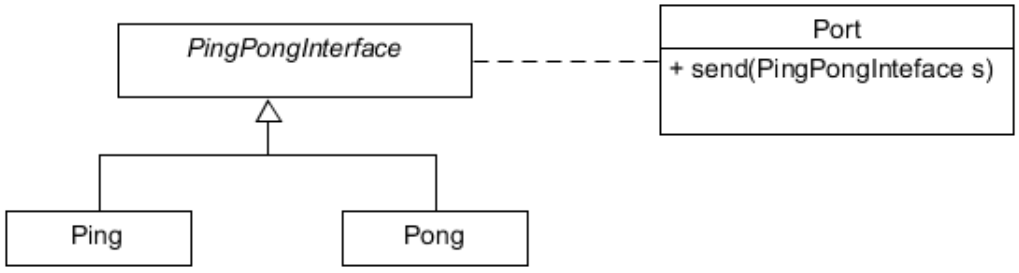


Ezek után technikailag többféle lehetőségünk van megvalósítani a validációt:

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: PingPongInterfce, a fogadó végpontjai pedig a következő szignálokat fogadják: *Ping*, *Pong*

   Majd a portnak legyen egy *send* művelete, ami az elvárt interfész alá sorolt szignálokat várja, vagyis olyan szignálokat, melyek implementálják a megfelelő interfészt. 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.



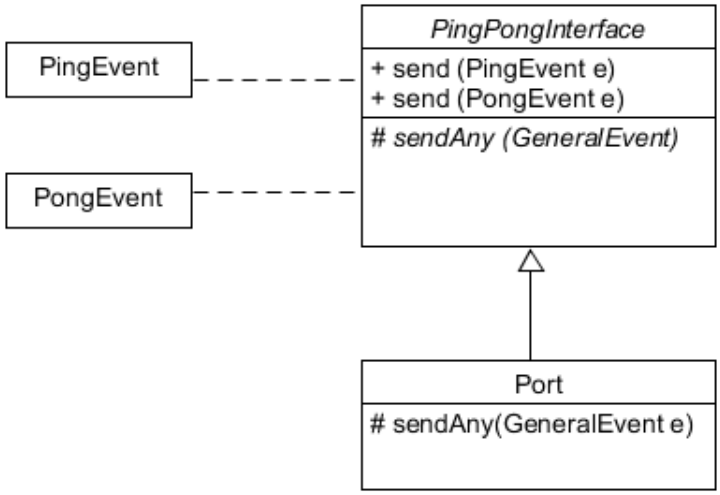
   Előnyök:

   - 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 receptiön 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. 3.5



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:

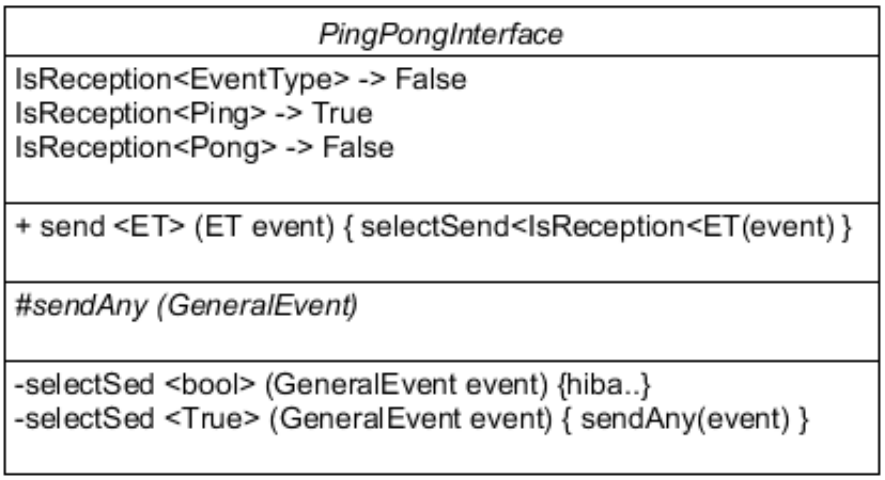
- A generált kód mérete azonban jelentősen megnő, mivel több *send* függvény generálásra van szükségünk, melyek csak a várt szignál típusban

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.
   Összefoglalva az eddigieket egy ábrában:



Ezzel azonban még nem értük el a kívánt egyszerű szintaxist, de a *send* függvényt legalább nem duplikáltuk fölöslegesen. A *TYPELIST* [12] 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

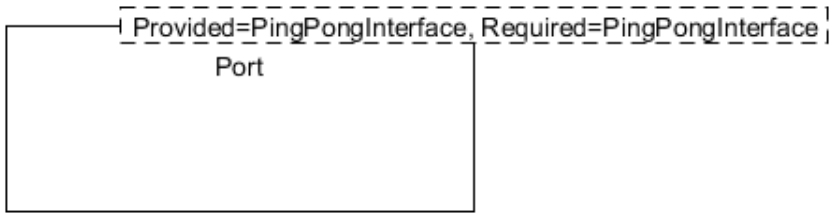
Az UML szabvány szerint egy *konnektor* csak olyan portok között húzódhat, melynek a szolgáltatott és elvárt interfészei kompatibilisek egymással. A port egy kommunikációs csomópontot reprezentál, a típusa azonban adattag révén tetszőleges lehet. A modellosztály számára a port szolgáltatásokat nyújt a szolgáltatott interfész által, Így a típusa megfelel a szolgáltatott interfésznek. A modellosztály azonban kifele továbbít 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 üzenetküldésre. 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 leszármazottja 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 leszármazni az adott port. A másik lehetőség az lenne, ha nem bontjuk két részre az interfészt, helyette a fogadó végpontok függvényeinek egy plusz paramétere lenne, hogy kívülről vagy belülről küldünk-e üzentet. (Ahogy azt a 3.3 bekezdésben felvezettük) 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



Az ábrán a Port-nak a szolgáltatott és elvárt interfésze is a *PingPongInterface* lesz. Az interfész maga nem tartalmaz egyetlen fogadó végpontot sem, helyette két különböző részből áll. Az egyik részben az üzenet küldő függvények, a másik részben pedig az üzenet fogadó függvények vannak felsorolva. Ha egy port szolgáltatott interfészének adjuk meg az adott interfészt, mely azt mondja meg, hogy milyen üzeneteket fogadhat, akkor az üzenet fogadó (*ProvidedPart*) résztől örökli meg az egyes műveleteket. Ha elvárt interfésznek adjuk meg a *PingPongInterface*-t, akkor pedig a *RequiredPart*-tól örököljük meg az egyes üzenet küldő műveleteket. A konkrét példában azt láthatjuk, hogy a *Port* a *PingPongInterface* fogadó és küldő műveleteit is örökli, mivel az elvárt és szolgáltatott interfésze megegyezik.

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, elevenítsük fel, mik is azok a konnektorok és mely részei érdekesek számunkra. A konnektorok egy kompozit struktúrában szerepeket kötnek össze, melyek olyan osztályokat fejeznek ki, melyek rendelkeznek porttal. kötnek össze. A konnektor egyfajta absztrakciót fejez ki két szereplő portja között, leírja, a szereplő mely két portja kapcsolódik össze, delegációval vagy assembly kapcsolattal vannak-e összekötve, illetve a kommunikáció prototípusát is képes leírni. A delegáció szülő-gyerek portok között jöhet lére, ilyenkor egy portra való üzenetküldést delegálunk a túloldalon lévő portra, mely továbbküldi a belső osztály felé, vagy a másik irány esetén a külvilág felé, az osztályon kívülre. Assembly összekapcsolás esetén két egyenrangú osztályok portja között jön létre kapcsolat, kölcsönös, oda-vissza kommunikációval. Fontos, hogy szülő-gyerek osztályok között nem lehet Assembly kapcsolat, illetve egyenrangú szereplők között Delegáció.

Konnektor reprezentáció UML-ben

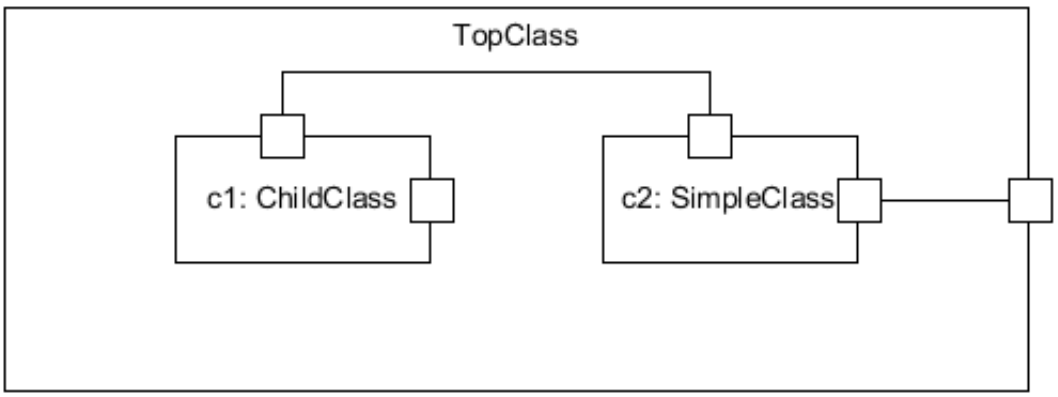
A szabvány szerinti reprezentáció viszonylag értelemszerű, egy *Connector* UML elemet kell létre hozni, a végpontoknak megadni a kompozíció szereplő portjait, beállítani, hogy Assembly vagy Delegált összekapcsolás van-e közöttük. A *Connector* elemnek van egy Asszociáció típusa, mivel a szabvány szerint 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.

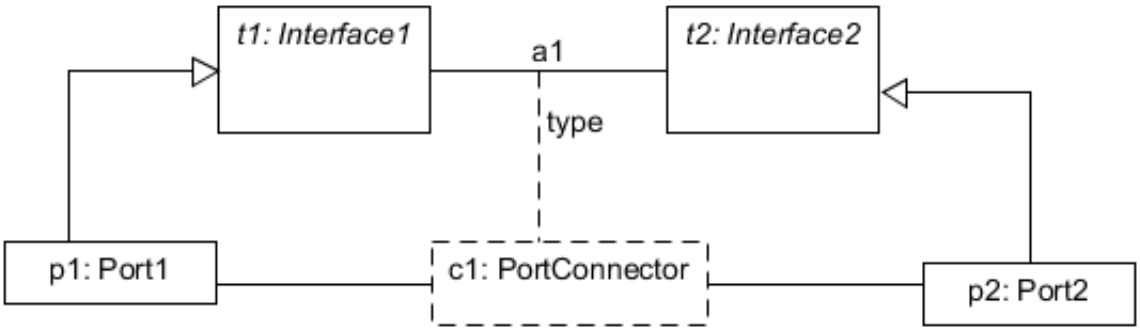
1. Mivel a Portok adattagok az UML-ben (Property-k) így két portot kölcsönösen értékül adhatunk egymásnak. Ez egy lehetőség lehet kifejezni az összekapcsolást, szabványos modell keletkezik, de mégsem ez a helyes szabványa Portok összekapcsolásának. A probléma ugyanis az lesz, hogy bármilyen két portot összekapcsolhatok így, konkrét konnektor realizáció nélkül.

2. A PSCS szabvány szerint, ha egy operáció egy konstruktor hívásnak felel meg (*Create* sztereotípiával van ellátva), és nem adunk meg hozzá konkrét metódustörzset, mint akciót, akkor az osztály kompozit struktúráját valamilyen alapértelmezett konstrukciós stratégia mentén fogja létrehozni. Mivel a *Connector* egy asszociáció példány, így már egy konkrét, létrejött kapcsolatról beszélhetünk, az alapértelmezett konstrukciós stratégia pedig megmondhatja, hogyan kell összekötni a portokat. Ebből adódóan a kompozíció 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, tehát meg kell mondani egy konkrét metódusban, hogyan inicializáljunk fel egy struktúrát.

Szintaktikailag sincs már más lehetőség, mint a *CreateLinkAction* használata, melyet a PSCS szabvány definíciója is kimond. Ehhez két dologra van szükség: Végpontokra (*LinkEndData*), valamint arra az asszociációra, melyet használni szeretnék az összekapcsoláshoz. Az asszociáció végpontok egyértelműek, felhasználhatjuk a konnektor végpontjait. Mint korábban láttuk, a konnektorok fel vannak típusozva egy *Association* típussal. Értelemszerű lenne ezt fölhasználni az összekapcsoláshoz, a kérdés csak az, mi legyen ez az asszociáció. A szabvány biztosít nekünk egy alapértelmezett asszociációt (*Generic Association*), melyet használhatunk abban az esetben, ha a konnektornak nincs típusa. Arra is van lehetőségünk, hogy saját asszociációt generáljunk a modellbe, mivel asszociáció típusok között húzható, a port önmagában ugyan nem típus, azonban adattag révén rendelkeznek típussal, a szolgáltatott interfészük típusával. Vegyük két port interfészét, nevezzük ezeket t1-nek, illetve t2-nek. Megtehetjük, hogy generálunk *t1* és *t2* között egy *a1* asszociációt, melyet a konnektor realizál a portok összekapcsolásával. Így a *CreateLinkAction*-be be tudjuk tenni asszociációként az a1-et, mint a konnektor típusát, ami mentén összekapcsolunk, nem kell használunk a *Generic Association*-t. Ezzel egy szabványos UML-t generálunk, és a konnektorokat sem hagyjuk figyelmen kívül két port összekapcsolásánál.



Az ábrán látható, hogy a *p1* és *p2* portnak a típusa az *t1* és *t2* interfészből származik, melyek között húzódik egy *a1* asszociáció. Ezek után a *c1* konnektor *type* referenciájába az *a1* asszociáció kerül bele.

3.5.1. Összekapcsolt portok reprezentációja C++-ban

A szabvány szemantikája szerint, amikor egy üzenet küldünk a *send* művelettel egy adott portra, annak továbbítania kell ezt a vele kapcsolatban álló port felé. Ehhez mindenképpen valami referenciát kell tárolni a kapcsolt portra. Ezt alapvetően kétféleképpen érthetjük el:

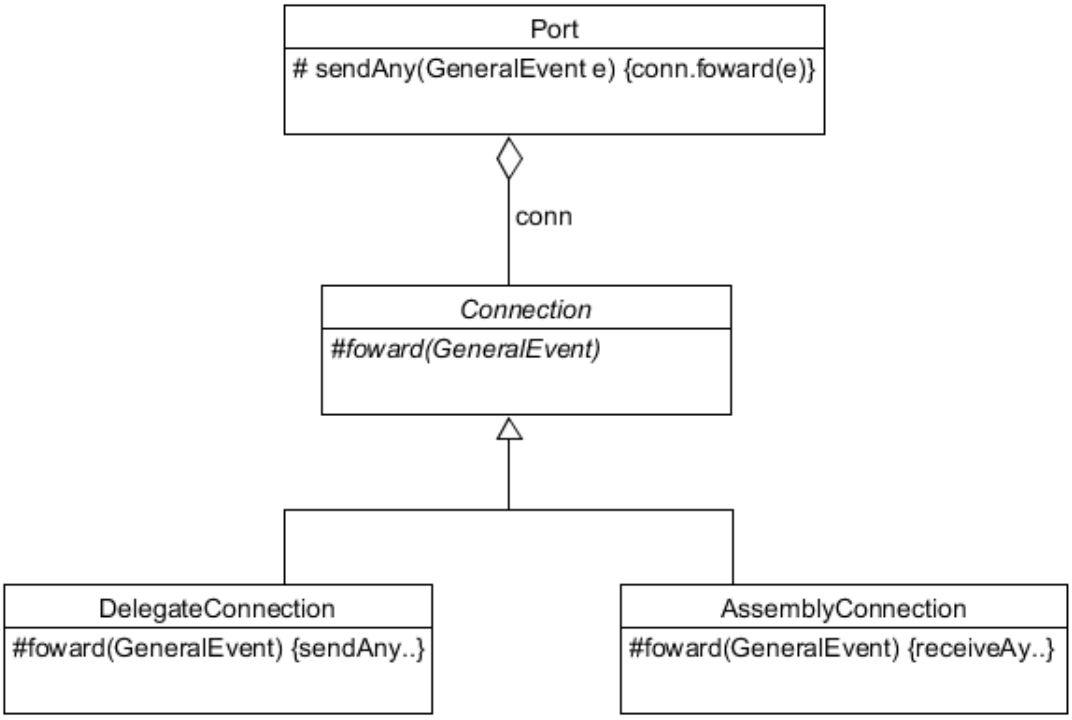
1. Egy globális adatszerkezetben, példásul egy asszociatív tömbben megmondjuk, melyik porthoz ki kapcsolódik. Ennek előnye, hogy nem a port kódja független marad a kapcsolatoktól, nem kell belekódolni a kapcsolatokhoz. Hátránya az, hogy egy párhuzamos környezetben egy globális adatszerkezetet kell manipulálni, 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.

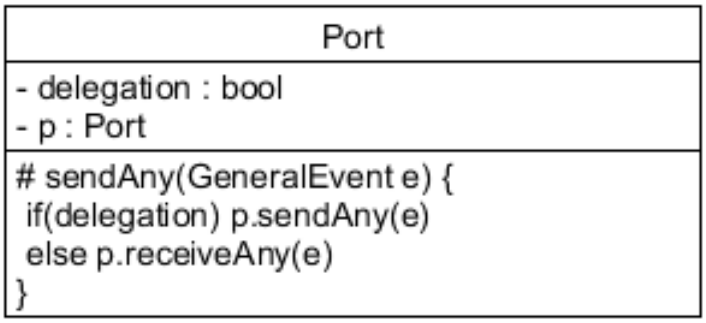
Az üzenetküldés szemantikájához már megvan a referenciánk, azonban ez még nem elég önmagában. Figyelembe kell vennünk azt is, hogy milyen típusú konnektoron keresztül kapcsolódtak az adott portok. Másképp kell átadni az üzenetet, ha Delegációról van szó, és másképp, ha *Assembly* kapcsolatról. Delegációs kapcsolat 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* típusú referencia helyett *Connection* típusú 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* típusú referencia, maga a port dönti el, melyik művelet hívására van szükség a jelző alapján. Ez a megoldás kevésbé letisztult, de hatékonyabb.



Hasonlóan az interfészek megoldásához, itt is az a megoldás lenne hatékonyabb, mely mindkét megoldás előnyeit kombinálja, tehát nem veszítünk a hatékonyságból, de megtartjuk a tiszta implementációt. Ha dolgozhatnánk leszármazással, ugyan-

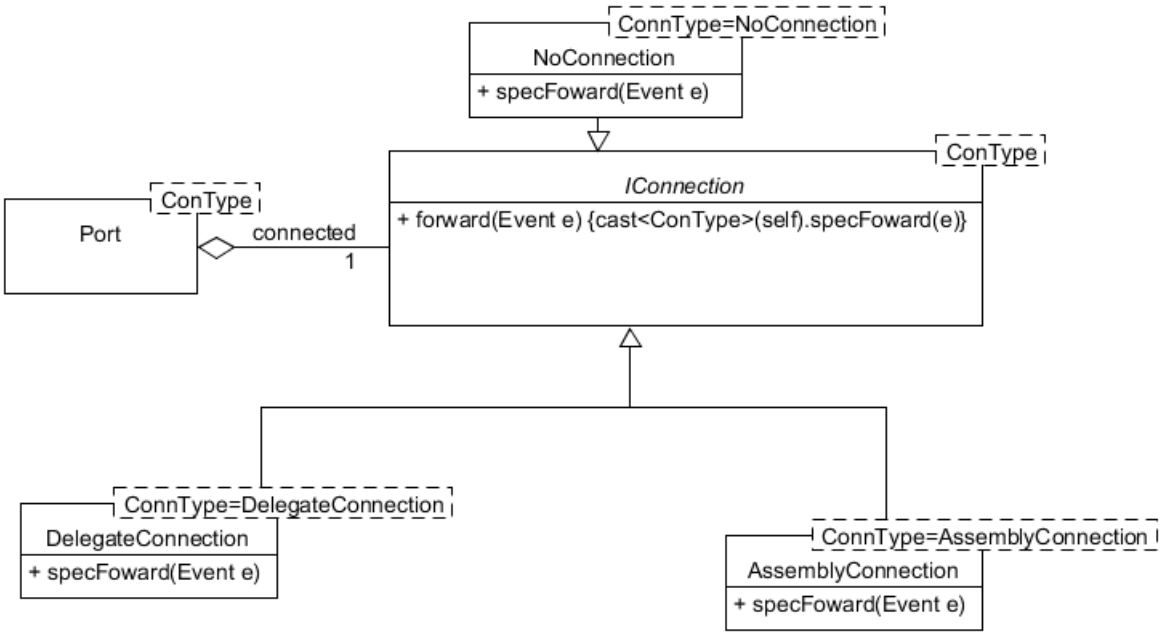
akkor nem kéne virtualizációt használni. Erre egy kézenfekvő megoldás az lenne, ha a leszármazáskor megadnánk az interfésznek sablon paraméterben, ki a leszármazottja.
Cél szintaxis:

```
class DelegateConnection : public Connection<DelegateConnection>
```

Majd definiálnánk a speciális implementációkat, az interfész implementáció pedig statikus konverzióval a specializált *Connection* osztályra konvertálná önmagát, és meghívná a megfelelő műveletet. Ehhez fordítási időben tudnom kell, hogy egy adott porthoz milyen típusú kapcsolaton keresztül fog létrejönni a referencia. (Assembly vagy Delegáció)
A portokat akkor kapcsoljuk össze, mikor létrehozzuk 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 ismerünk, a potenciális összekapcsolások típusait igen, így fordítási időben el tudjuk dönteni egy adott kapcsolatról, milyen típusú. Az is lehetséges, hogy egy félkész, vagy hibás modell exportálása esetén egyáltalán nem adunk meg Konnektort két port között. Ilyenkor exportálási hibát nem szeretnénk adni, mivel lehetséges, hogy a felhasználót tudatosan érdekel egy ilyen modellnek a C++ kódja, helyette inkább bevezetünk egy *NoConnection* típusú leszármazást, melynek műveleteinek szemantikája megegyezik az üres utasítással. Akár sablon specializációval is dolgozhatunk, mely azért is lesz előnyös, mert a *Connection* ősosztályban általánosan tárolunk egy port referenciát, melyeket az ősök megörökölnek, így ezt az adattagok megspórolhatnánk.

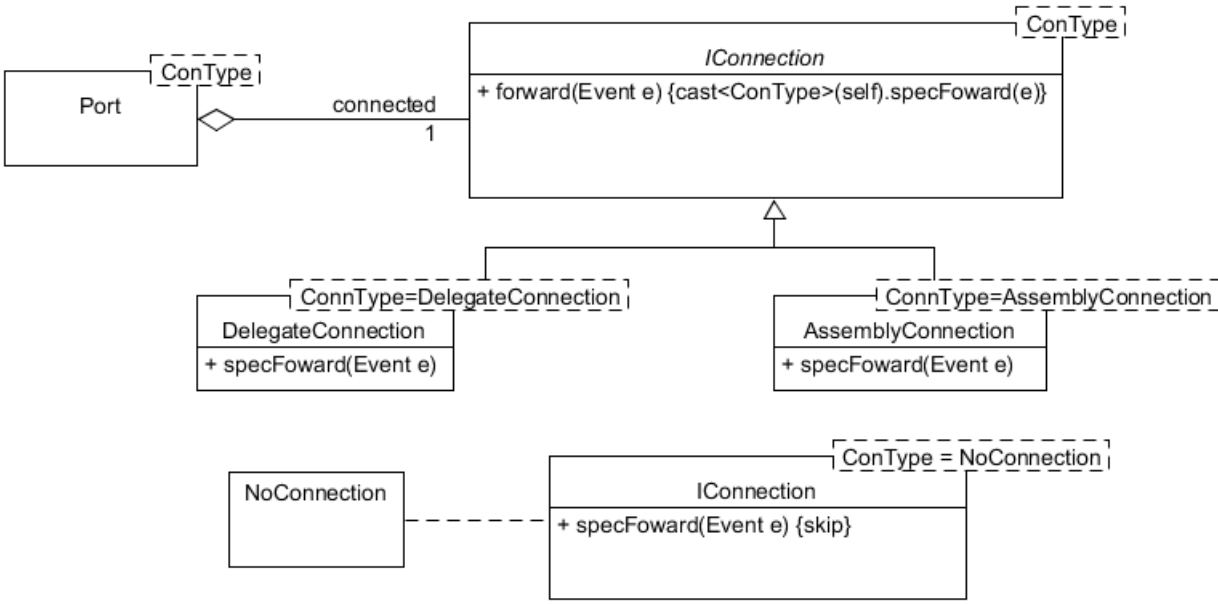
Megoldás szemléltetése

1. *NoConnection* leszármazással:



Ebben az esetben a *NoConnection* osztály leszármazik az *IConenction*-ből, így tartalmazza a *Port* típusú *connected* referenciát.

2. *NoConnection* specializációval:

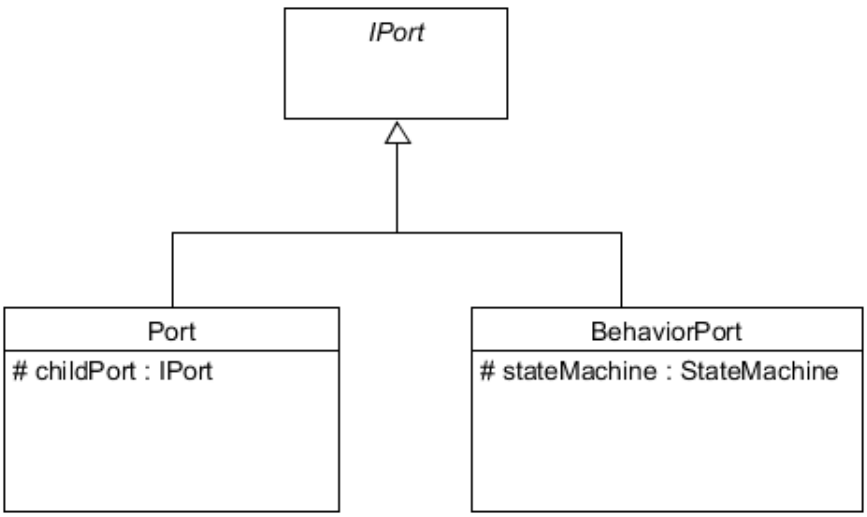


Ebben az esetben a *NoConnection* osztály csak egy szereposztály, mellyel specializáljuk az *IConnection* osztályunkat, így elhagyhatjuk belőle a *Port* típusú referenciát is.

Különböző port típusok megvalósítása

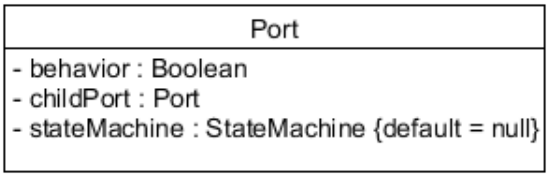
Az üzenetküldés megvalósítása után vizsgáljuk meg az üzenet fogadás (*receive*) szemantikáját is. A fogadás művelet végrehajtási szemantikája attól függ, hogy állapotgéppel összekapcsolt Portrol (Behavior Port) beszélünk-e vagy általános portról. Előbbi esetén egy objektum állapotgépéhez futnak be az 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, egyéb esetben pedig a gyerek komponens portjára. Ez különböző adattagokat igényel, az adat redundancia miatt leszármazás egy kézen fekvő választás. Unio típus segítségével azonban el lehetne kerülni a redundáns tárolást, ami egy hatékonyabb, de nehezebben olvasható kódot eredményez.

1. Leszármazással:



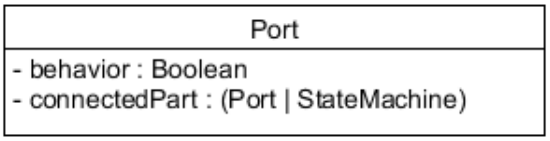
   Nem szorul magyarázatra, az ábrán a két speciális port látható különböző adattagokkal.

2. Behavior jelzéssel:



Az ábrán az a megoldás látható, hogy minden adatot egy osztályon belül eltárolunk, és a *behavior* adattag dönti el port típusát. A másik lehetőség lehetne, hogy az alapján döntünk a típusról, hogy melyik adattag van kitöltve, bevezethetnénk akár egy *Optional* típust is.

3. Unio típussal:



Itt mindenképpen kell a *behavior* jelző, hogy el tudjuk dönteni, pontosan melyik adattagra hivatkozhatunk az unio-ból.

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 egyerűség lesz, a generált kód mérete kisebb lesz. Hátránya pedig a validáció teljes hiánya, bármilyen két portot összekapcsolhatunk C++-ban. Az interfészeknél arra jutottunk, hogy szükséges a validáció, így ez nem a legjobb megoldás.

2. A másik lehetősünk 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 muszáj kigenerálni, függetlenül a konnektoroktól, így nem igényel további strukturális 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 egymásnak.

4. A legoptimálisabb megoldás most is a két megoldás előnyeinek az összekeveréséből lesz. A 3.5.1 fejezetben megállapítottuk, hogy fordítási időben képesek vagyunk eldönteni egy adott portról, hogy az milyen típusú kapcsolatban állhat a referenciában lévő porttal. Ez azért van, mert ismerjük a modellből a Konnektor struktúrákat, és egyértelműen el tudjuk dönteni, potenciálisan milyen kapcsolat jöhet létre a két port között. Ha nem tudnák eldönteni, egy nem egyértelmű, hibás modellt írtunk, mellyel most nem foglalkozunk. Ennél fogva, a konnektorból nem csak annak típusát, de azt a szerepnevet is ki tudjuk nyerni, mely meg fog egyezni a generált asszociáció szerepneveivel. A *Connection* referencia típusában pedig nem csak azt kódolhatjuk bele, milyen típusú kapcsolatban áll az adott porttal, hanem azt is, hogy az átellenes port milyen szereppel van jelen. A referenciák beállítása pedig csak egy olyan típusú *Connectiont* enged beállítani, ami megfelel a szerep típusúnak. A szerepeket a port sablonparaméterében adhatjuk meg. Ha egy hiányos modellben nincs kapcsolatban egy port egy adott *Connector*-on keresztül egyetlen másikkal sem, akkor megadhatnánk egy extremális *NoRole* szerepet, de itt már egyértelműen érdemes sablonspecializációval dolgozni, és a 3.5.1 fejezetben említett módon egyszerűen használni a *NoConnection*-re specializált verziót.
   Előnyök:

   – Továbbra is elég csak asszociációkat generálni a megfelelő szerepekkel.

   – A típus és szerep validációról is gondoskodunk, nem engedünk két olyan portot összekapcsolni, melyek között nincsen *Connector*

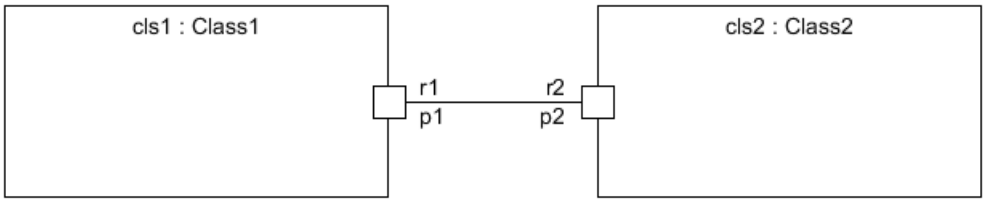
   Hátrányok:

– Azonban a portok kódja elbonyolódik sablonparaméterekkel, mely külső kód esetén nem optimális.

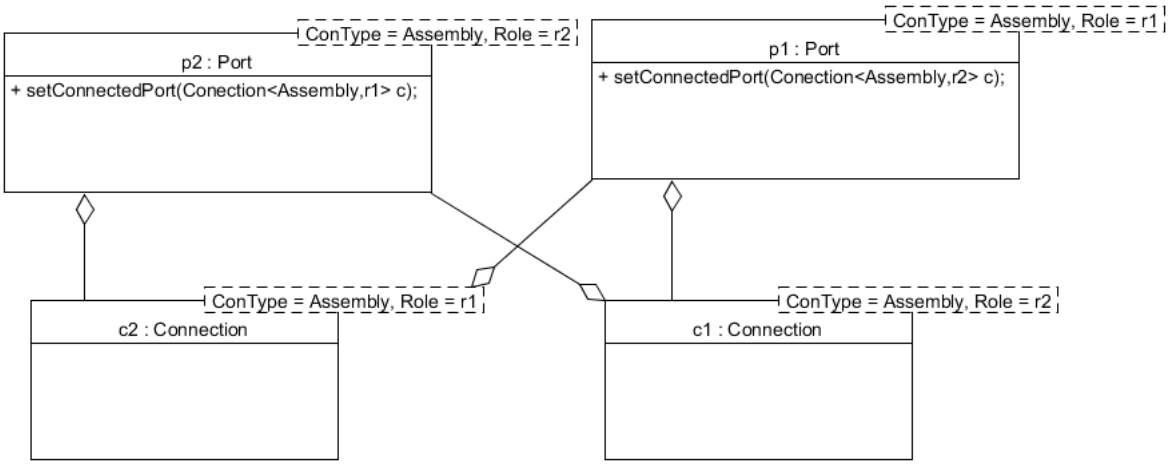
Példa: A példában van két portunk (textitp1 és p2), *r1* és *r2* szerepekkel, melyek Assembly kapcsolatban állnak egymással:

(p1) r1 <–> (p2) r2

Kompozit struktúra:



Megoldás szemléltetés objektumdiagrammal:



Ekkor az ábrán azt láthatjuk, hogy a *p1* port - mely a kapcsolatban *r1* szereppel áll - tartalmaz egy kapcsolt portot *r2* szereppel, és fordítva. A *Connection* objektumok pedig értelem szerűen a szerepüknek megfelelő portot tartalmazzák. Amikor pedig beállítjuk a megfelelő kapcsolatot, akkor csak a megfelelő típusú és megfelelő szerepű végpontot adhatjuk meg (a *p1* portnak *rl* és szerepet csak Assembly végpontot *r2* szereppel), egyébként pedig fordítási hibát kapnánk.

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 osztályon belülről. Ekkor az üzentet kifele továbbítjuk, egy másik port fogja megkapni. Ebben a fejezetben csak az lesz az érdekes, hogy az üzenet továbbítások végén, amikor eljut az üzenet egy állapotgéphez, hogyan fogjuk feldolgozni. Így csak az állapotgéppel összekapcsolt portok fogadó (*receive*) műveletei lesznek érdekesek számunkra ebben a fejezetben. A port tartalmaz egy referenciát az állapotgépre, így tudja neki továbbítani az üzenetet. 3.2.2 fejezetben láttuk, hogy a portokkal való kommunikációt hasonlóan fogjuk kezelni, mint ahogy eddig az üzenetküldést kezeltük, tehát továbbítjuk az üzenetet az állapotgép felé. 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 üzenet lehetséges forrásportjait. *txtUML*-ben csak egy portot adhatunk meg, vagy jelezhetjük, hogy az adott üzenet bárhonnan érkezhet, porton kívülről is, vagy bármely portról. (*AnyPort* extremális érték) Ennek megfelelően a meglévő állapot-átmenet táblát érdemes kibővíteni egy új dimenzióval: Megadni, hogy az átmenetben szereplő üzenetnek mely portok lehetnek a forrásai. Eddig *(esemény,állapot)* kulcsú táblánk volt, melynek értékei ez alapján megadták, melyik átmenetnek kell végrehajtódnia. [4] Most vizsgáljuk meg, hogyan tudjuk kiterjeszteni ezt a táblát.

1. A kulcsait kibővítjük egy port típusinformációval, így *(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, ki a forrás portja. Ez a *txtUML* esetében jól fog működni, mivel csak egyféle portról jöhet az üzenet. Azonban, ha azt adjuk meg, hogy bármelyik portról érkezhet az üzenet, már problémákba ütközünk, mivel ilyenkor nincs megadva port dimenzió a táblában, az üzenetben viszont igen. Ezt speciálisan le tudjuk kezelni az üzenetfeldolgozás során, azonban ez nem egy elég általános megoldás.

Átmenet tábla			
Esemény	Állapot	Port	Átmenet-Őrfeltétel
Event1	State1	Port1	Ttransition1-Guard1
...	...	...	...
EventN	StateN	PortN	TtransitionN-GuardN

2. Általánosabb megoldás, ha a port helyett megadhatnánk egy maszkban azokat a portokat, melyek lehetséges forrásai lehetnek egy átmenetnek. Mivel egy üzenet továbbra is csak egy helyről érkezhet, így elég lenne csak azt vizsgálunk, hogy az *(állapot,esemény)* kulcs alapján megkeresett átmenet érték portjai között szerepel-e a az a port, melyről az üzenet érkezett. Itt a port dimenzió a táblázat értékei között 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 forrásportjai között szerepel az üzenet forrásportja. Azt az esetet, amikor egy üzenet forrása nem port, egy extremális *NoPort* speciális értékkel jelezhetjük. *AnyPort* esetén az osztály összes lehetséges portját belerakjuk az átmenetbe, valamint a *NoPort* típust is.

Átmenet tábla		
Esemény	Állapot	Átmenet-Őrfeltétel-Portok
Event1	State1	Ttransition1-Guard1-Ports1
...	...	...
EventN	StateN	Ttransition1-Guard1-PortsN

4. fejezet

Portok felhasználása a modellezésben

4.1. FMU modellek generálása

A portok alapvetően a környezettől való szeparáció igénye miatt jöttek létre, mely elősegíti a komponens-alapú fejlesztést. Ebben a világban egy komponens bármilyen komplexitású lehet, akár az egész modellt tekinthetjük egy komponensnek, melynek van egy belső működése, és portokon keresztül kommunikál a külvilággal, hozzáköthetünk más, hasonló elven felépített modelleket. Ennek az elképzelésnek egy szabványa az FMI(Functional Mockup Interface) [13], mely kommunikációs interfészt ír le funkcionális egységek, modellek között. Ezeket nevezzük FMU-nak (Functional Mockup Unit), mely alatt egy olyan egységet értünk, mely az FMI szabvány segítségével tud kommunikálni a külvilággal, illetve más egységekkel. Az FMU funkcionalitása leírható egy diszkrét modellként állapotgéppel, mely input adatok hatására megváltoztatja a belső állapotát. Az FMI szabvány rendelkezik a *DoStep* művelettel, melyben megadjuk az aktuális input adatokat, és végre hajtjuk a megfelelő állapotváltozást. Ezek olyan folytatonos modellekkel vannak összekötve, melyek folyamatosan mérik a külvilág állapotát (pl. egy fűtésrendsernél, hogy mennyi a hőmérséklet), ezeket adott időközönként továbbítják egy FMU-nak a *DoStep* műveleten keresztül, mely megfelelően reagál változásra. (pl. abbahagyja a radiátortest melegítését) A portok, interfészek az *OpenCPS*[15] nemzetközi projektben is hasznosnak bizonyolultak, melyben a *txtUML* segítségével írtunk meg

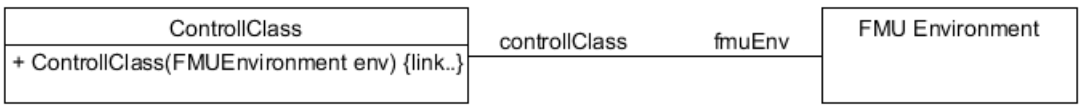
egy *FMU* diszkrét modellt, majd generáltunk belőle a szabványnak megfelelő *C++* kódot, így a modellt össze lehet kötni más modellekkel. Az FMI egy C-ben írt szabvány, így a C++ által generált modell összecsomagolható egy FMU-vá. [14]
Ezt az alábbi módon valósíthatjuk meg: Létrehozunk egy *FMUEnvironment* külső osztályt, mely implementálja az FMI-t. A környezetet egy asszociáció segítségével hozzáköthetjük a modellünk egy osztályához, melynek el tudja küldeni az input adatokat feldolgozásra, és megkapja tőle az output adatokat. Ehhez az *FMUEnvironment*-nek ismernie kell a modellben egy osztályt, mellyel összekapcsoljuk, és az osztálynak is ismernie kell az *FMUEnvironment*.
Az asszociáció helyett azonban portokkal is dolgozhatunk. Egy FMU-t felfoghatunk egy olyan strukturális rendszernek, melynek két eleme van:

1. Az *FMUEnvironment*, mely megvalósítja az FMI-t. (pl. a*DoStep* műveletet)
2. Az FMU működését leíró *txtUML* modell.

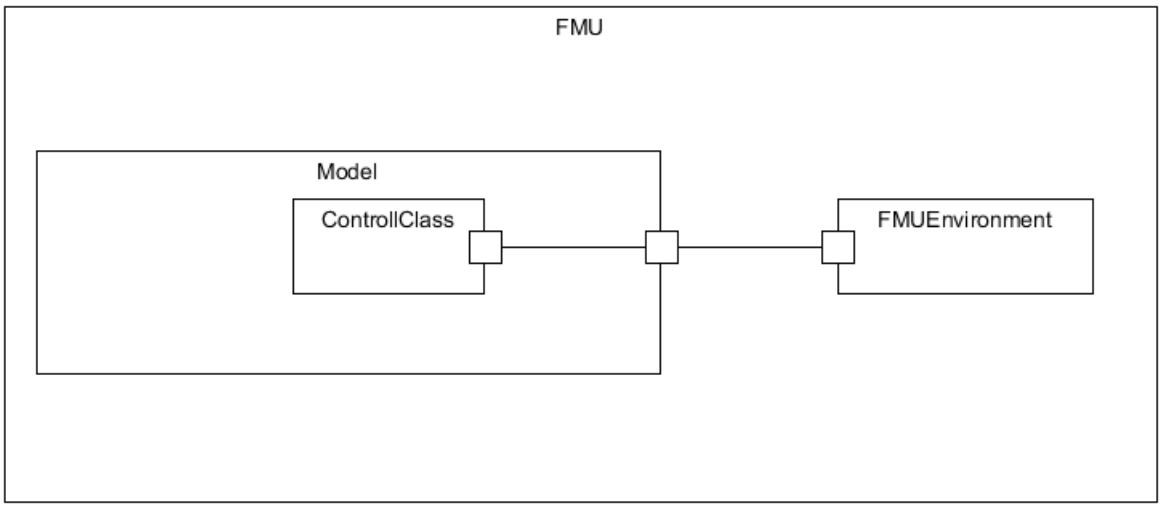
Ezek után a környezet és a modellt egy assembly kapcsolat segítségével összekötjük. Így a környezetnek nem kell ismernie a speciális kontroll osztályt, mellyel asszociációban van, de még a modell-re sem kell referenciát birtokolnia, csak a saját portját kell ismernie. A modell osztály pedig az input üzenetet tetszőleges gyerekének delegálhatja, az előzőekben említett kontroll osztálynak is. Az összekapcsolásokat pedig a legfelső osztály végzi el, mely összefogja az egész rendszert. Elég ennek az egy szereplőnek ismernie a modellt és a környezetet.

Szemléltetve az egyes megoldásokat

Egyszerű estet, amikor nincsenek portok:



Portos eset:



Előnyök:

- A modellt függetleníteni tudjuk a környezettől.
- Eddig, ha *FMU* kompatibilisen írtuk meg a modellünket, akkor fel kellett venni egy asszociációt, melynek az egyik osztálya az *FMUEnvironment*. Ha ennek ellenére nem szerettünk volna *FMU* generálását, akkor a generált kódban megjelent egy ismeretlen osztályként az *FMUEnvironment* referencia. Mivel az új megoldásban a modell egy olyan egység, aminek csak interfész függőségei vannak a portok révén, így egyszerűen generálhatunk egy forduló modellt akkor is, ha nem szeretnénk *FMU* csomagolást.
- Tisztább struktúra érhető el, könnyebbé válik bármilyen külső, modelltől független kódot hozzákötni a modellhez.

Hátrányok:

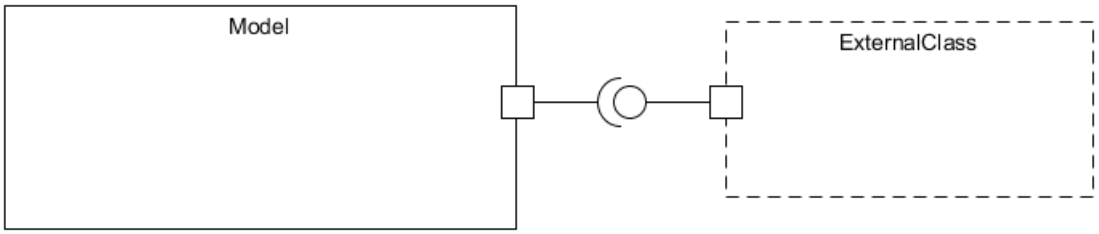
- Nehezebb leírni ezt a strukturális felépítést *txtUML*-ben.

Azonban a hátrányt kiküszöbölhető azzal, hogy ha a felhasználó beállítja, hogy ha szeretne FMU-t generálni a modellből, akkor minden szükséges strukturális kódot legenerálhatunk automatikusan, mivel a modell teljesen függetlenedett az *FMUEnvironment*-től, így azt *txtUML* modellben fel sem kell tüntetni, csak konfigurációnál kell megadnunk, mely portja szolgál majd a külvilággal történő kommunikációra.

4.2. Külső szereplők bekötése a modellbe

Általánosságban elmondható, hogy ha szeretnénk egy külső szereplőt bekötni a modellünkbe anélkül, hogy a generált kódhoz kézzel hozzá kéne nyúlnunk (pl. felvenni valamely osztályában referenciaként a külső osztályt), akkor a portok nagy szerepet játszanak ebben. Elég csak felvenni egy csatlakozási pontot a modellben, meghatározni, hogy milyen interfészen keresztül kommunikálhatunk a külvilággal, és kívülről hozzácsatlakozni az adott porthoz. Így, a generált kód teljesen függetlenedik a kézzel írt kódtól, újragenerálás után csak arra kell figyelnünk, hogy a külső kommunikációs port neve ne változzon, az interfésze ne szűküljön.

4.2.1. Szemléltetve



4.3. Párhuzamosítási lehetőségek kiterjesztése

A diplomamunkának nem célja a modellekből történő párhuzamos vagy elosztott alkalmazások kódjának generálása. A portok használata azonban egy fontos lépés ezen cél elérésében, mivel portok segítségével úgy tud egymással két osztály kommunikálni, hogy nem tárolnak egymásra referenciát, melynek köszönhetően nem tudnak szinkron hívást kezdeményezni egymás metódusaira, nem tudják elérni egymásnak még a publikus adattagjaikat sem, egyedül aszinkron üzenetküldéssel tudnak kommunikálni egymással. Ennek következtében egy metódus végrehajtásnál egy osztály csak a saját adattagjait olvashatja vagy írhatja felül, vagy a metódus törzsében létrehozott lokális változókat. Így nem lehet interferencia egy adattag elérésekor, ha a metódus végrehajtása közben nem ágazunk el, és hozunk létre újabb

szálakat. Ehhez persze arra is szükségünk van, hogy egy szignál ne tartalmazzon referencia értékéket, csak lemásolt, vagy primitív objektumokat. Továbbá statikus függvényhívásokra és változóelérésekre sem szabad lehetőséget adni az egyes osztályoknak, vagy ezen eléréseket szinkronizálni kell.

Ezen primitív szabályokat betartva elmondható, hogy két azonos szinten lévő, assembly kapcsolattal kommunikáló osztály porton keresztül kommunikálva nem fog interferenciába keveredni egymással. Szülő-gyerek kapcsolat esetén azonban nem ilyen szerencsés a helyzet, mivel a szülő kompozícióban áll a gyerekével, így referenciát tárol rá a szabvány szerint. Így ezen osztályok párhuzamosítása egy nehezebb feladat, és mivel a párhuzamosítás témája túlmutat a diplomamunka témakörén, így ezt nem vizsgájuk részletsebben.

4.4. Testre szabható protokoll

Egy portra való üzenetküldés megvalósítását tetszőlegesen kicserélhetjük, a *send* művelet megvalósítása akár egy, a modell mellé írt konfigurációtól is függhet. Ha két osztályt szeretnénk különböző szálakra helyezni, akkor valamilyen szálbiztos megvalósítást kell alkalmaznunk. Ha az egész modellünk egy szálon fut, akár valamilyen szinkron hívást is megvalósíthatunk a háttérben. Két osztály akár különböző processzben is futhat, ez esetben a portoknak valamilyen interprocessz kommunikáció kell implementálnia. Ezen lehetőségek részletes kielemzése szintén túlmutat a diplomamunka témakörén.

4.5. Az UML szabvány egyszerűbb szemléltetése

Amikor komponensekről, szereplők izolációjáról, elvárt és szolgáltatott interfészekről, konnektorokról beszélünk az UML kapcsán, akár egy egyetemi előadás keretében, akkor a *txtUML* portjai, annak szemantikájának demonstrálása egy modellen, és C++ megvalósítások elemzése hasznos anyag lehet, hogy a hallgatókhoz közelebb kerüljenek ezek a fogalmak. Így a későbbiekben a rendszerek tervezésnél nem csak egymással asszociációban lévő osztályokban fognak gondolkodni, hanem teljesen izolált szereplőkben is, mely kommunikációs pontokon, portokon keresztül

kommunikál a külvilággal.

5. fejezet

Megvalósítás

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.

5.1. Felhasználói interfész tervezés

A dolgozat a *txtUML* keretrendszerre épül, melynek részletes felhasználói dokumentációja megtalálható a annak hivatalosan honlapján [17].

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

5.1.1. 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..)

5.1.2. Generálás és futtatás

Kódot generálni legegyszerűbben Eclipse-ből tudunk egy txtUML modell megírása és egy kihelyezési konfiguráció alapján. A kódgenerálás eredményeként különféle kódok keletkeznek, különböző szerepekkel, különböző helyeken, melyek két nagy csoportra bonthatók. A *runtime* mappában az előre megírt fájlokat másoljuk, mely tartalmazza többek között a *Port* osztályt is, az összekacsoló műveleteket, és azok szemantikáját. Ezek a kódok egyfajta kertrendszerként funkcionálnak, minden modellnél megegyeznek. A konkrét port példányokat, konnektorokat, interfészeket pedig a *runtime* mappán kívül generáljuk az input modell alapján.

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

5.2. Rendszerarchitektúra

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

5.2.1. Szerkezeti felépítés

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

1. A *Java* modellt *UML* modellre leképző algoritmusok. Ezek nem tiszta Javaban, 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 **PortReferenceExporter.xtend** fájt, az üzenetküldéshez a **SendToPortActionExporter.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 kapcsolatokat, 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.

5.2.2. Tesztelés

A demonstrációs program néhány tesztesetét 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, ekkor fordítási hibát kapunk.

c) Nem-üres interfésszel, állapotgéppel összekapcsolt portot hozunk létre. Az osztály egy átmenete tartalmazza az adott portot. A *receive* művelet kipróbálása megfelelő eseménnyel.

3. Kétosztályos modell létrehozása, egy szülő osztály, melynek van egy belső osztálya. Mindkettőnek van egy-egy portja.

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

   b) Assembly kapcsolattal összekapcsoljuk őket, mely fordítási hibát jelez.

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

4. Komplex modell létrehozása, mely demonstrációs célokra is megfelelő: Egy ping-pong játék struktúra létrehozása, mely tartalmaz egy táblát, valamint egy bírót. Ezek tartalmaznak egy-egy portot, melyek Assembly kapcsolattal vannak összekötve. A tábla tartalmaz két játékost, melyek két portot tartalmaznak, egy ping-pong portot, valamint egy olyan portot, melyről a játék kezdése üzenetet várják. A ping-pong portok egymással vannak összekötve, ezen keresztül passzolgatnak egy szignált oda-vissza a játékosok egymás közt, az első játékos pedig delegációval hozzá van kötve a táblához. A játék indulásakor a bíró lead egy játék kezdése szignált a táblának, melyet tovább delegálunk a kezdő játékosnak, és ő elindítja az első szerválást.

A generált kódok megfelelőek, a modellek pedig az elvárt működésnek megfelelően működnek.

6. fejezet

A diplomamunka eredményei

Végül foglaljuk össze a diplomamunka főbb eredményeit!

6.1. UML kompozíciós szabvány áttekintése, értelmezése

A kompozíciós elemek az *UML*-nek kevésbé ismert és letisztult részei közé tartoznak, melyet már strukturális szinten is bonyolult értelmezni. A diplomamunka egy alap eszköztárat nyújt a bevezetésben az *UML* kapcsolódó fogalmaira. A portokkal kapcsolatos fogalmakat, az elvárt és szolgáltatott interfészeket, az üzenetküldés és fogadás szemantikáját, portok összekapcsolásáért felelős konnektorokat és konnektor műveletek helyes *UML* reprezentációját és annak szemantikáját mind elemzi a diplomamunka. A helyes leképezés elemzésére a szabványos modell előállításához van szükségünk, míg annak szemantikájára a helyes C++ kód generálásához.

6.2. Modellező keretrendszerek, UML szabványok áttekintése

Az elemzések előtt a diplomamunka megvizsgálja, milyen szabványos segítenek nekünk megtalálni a megfelelő *UML* reprezentációkat az egyes elemekhez, és mi ezeknek a pontos jelentésük. Majd áttekint különböző modellező keretrendszereket, melyeket tartalmaznak valamilyen kódgenerálást, és megvizsgálja a kompozíciós ele-

mek exportálását. Itt arra jutunk, hogy a legtöbb kertrendszer a kompozíciós elemeket (portokat, konnektorokat) csak nagyon kezdetleges módon támogatja, vagy nem foglalkozik a precíz *UML* szabvánnyal.

6.3. C++ fordítási lehetőségek elemzése

A diplomamunka egyik legfontosabb eredménye a C++ fordítási lehetőségek elemzése, melyben az *UML* elemek precíz szemantikájának tudatában megvizsgálja, milyen lehetőségek vannak megvalósítani *C++* nyelven a kompozíciós elemeket. Itt a hatékonyságot és a minél tisztább és tömörebb kódot tartjuk szem előtt. A kód tisztaságára azért helyezünk hangsúlyt, mert feltételezzük, hogy a generált kódhoz a felhasználó *C++*-ban ad kiegészítést, integrálva egy meglévő projektbe, felhasználva azt az API-t, melyet a generálás során is használhatunk. (Pl. egy üzenetküldésnél) Az elemzések végén megvizsgáljuk azt is, hogy a kompozíciós modellezés milyen esetben válhat a hasznunkra.

6.4. Prototípus készítése a meglévő elemezések alapján

A diplomamunkához a leírtak alapján implementációt is készítettünk, mely az eredmények demonstrálására, az *UML* szabvány konkrét példákon való megértésére is alkalmas lehet. Továbbá az implementáció segítségével ellenőrizhetjük az elemzéseink helyességét, és hatékonyságát. Az implementációt felhasználva lehetőségünk nyílik átalakítani az *fmu* exportálását is az *OpenCPS* projekt keretein belül.

Irodalomjegyzék

[1] OMG, *Unified Modeling Language (OMG UML)*, standard, 796, December 2017 https://www.omg.org/spec/UML/About-UML/ November 2018

[2] Bran Selic, ObjecTime Limited
Jim Rumbaugh, Rational Software Corporation *Using UML for Modeling Complex Real-Time Systems*, article
ftp://ftp.omg.org/pub/umlrtf/UML_rt_ext_F56dist.pdf November 2018

[3] Gábor Ferenc Kovács, Ádám Ancsin, Gergely Dévai,
Textual, executable, translatable UML, article 14th International Workshop on OCL and Textual Modeling, 2014.

[4] Hack János
C++ kódgenerálás futtatható UML modellekből, article, 2015

[5] Nagy András
Konfigurálható szálkezelés modellfordítóhoz, article, 2016

[6] Object Management Group.
Action Language for Foundational UML (ALF) standard , version 1.1, June 2017
https://www.omg.org/spec/ALF/About-ALF/ November 2018

[7] Object Management Group.
Semantics of a Foundational Subset for Executable UML Models (fUML) standard, version 1.4 beta , November 2018
https://www.omg.org/spec/FUML/About-FUML/ November 2018

[8] Object Management Group.
About the Precise Semantics of UML Composite Structures Specification standard, Version 1.1, 158, March 2018
https://www.omg.org/spec/PSCS/About-PSCS/ November 2018

[9] Andrew Forward, Omar Badreddin, and Timothy C Lethbridge
Umple: Towards combining model driven with prototype driven system development, Article
In Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on, pages 1–7. IEEE, 2010.

[10] OneFact,*BridgePoint xtUML tool.*, Software
http://onefact.net, 2018. 11. 28.

[11] MKLab, StartUML Documenation, software
https://docs.staruml.io/developing-extensions/accessing-elements, November 2018

[12] Codeproject,
TypeLists and a TypeList Toolbox via Variadic Templates, atricle
The Code Project Open License (CPOL), 16 Feb 2016
https://www.codeproject.com/Articles/1077852/TypeLists-and-a-TypeList-Toolbox-via-Variadic-Temp, November 2018

[13] Modelica Association, Functional Mock-up Interface for Model Exchange and Co-Simulation, standrad, 126
version 2.0, July 25, 2014
https://fmi-standard.org/docs/2.0.1-develop/, November 2018

[14] Jérémie TATIBOUET,Task Leader, Ákos HORVATH, Zoltan GERA, Boldizsár NEMETH, Dániel SEGESDI, Gergely Dévai, Ákos HORVATH, Sébastien REVOL
Interoperability of the standards Modelica-UML-FMI, report, 92 ,Nov 17, 2017

[15] OpenCPS projekt hivatalos oldala:
https://www.opencps.eu/ November 2018

[16] Apperly, H. *Service- and Component-based Development Using Select Perspective and UML*, book, ISBN: 978032115985, Addison-Wesley, 2013
https://books.google.com.bn/books?id=k69QAAAAMAAJ November 2018

[17] txtUML project hivatalos oldala:
http://txtuml.inf.elte.hu/wiki/doku.php November 2018