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 szerint. 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 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 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 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 abstrakt 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 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.

1.1.4. UML Connector

Egy asszociációt realizál két port között. Leírja, hogy milyen portokat köt össze, assembly vagyd 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. Szabványok és keretrendszerek áttekintése

2.2.1. fUML - ALF

A Foundational UML (fUML) [5] egy Object Managment Group (OMG) által specifikált végrehajtató UML szabvány, melynek szöveges akciónyelvi szintaxisát az ALF (Action Language for Foundational UML) [4] 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 [6] 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 érték pedig végig csorog a kapcsolatban álló portokon, melyre szintén egy aktivátorok 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

2.2.3. 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.

2.2.4. 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 konnektorrá. Hasonlóan a StartUML-hez, az akciókat aktivitás nodeok révén kézzel kell összerakni, így nehéz szabványos port műveleteket létrehozni. Összességében a BridgePoint-ról [7] 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. 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:

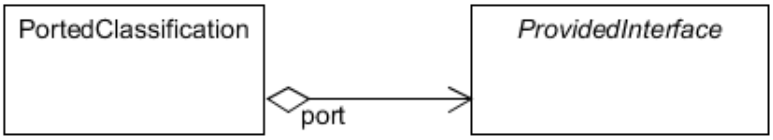
- Interfészek
- Portok és üzenet küldés/fogadás művelet
- Konnektorok és connect művelet

3.2. Portok reprezentálása

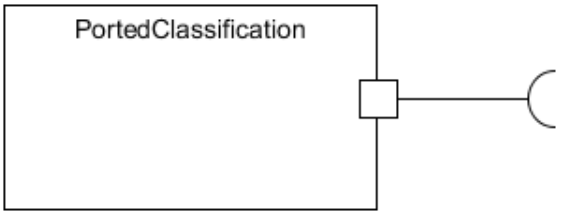
3.2.1. UML-ben

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



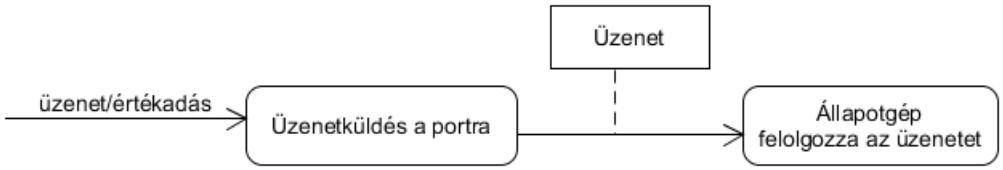
Szemléltetés komponens diagrammal



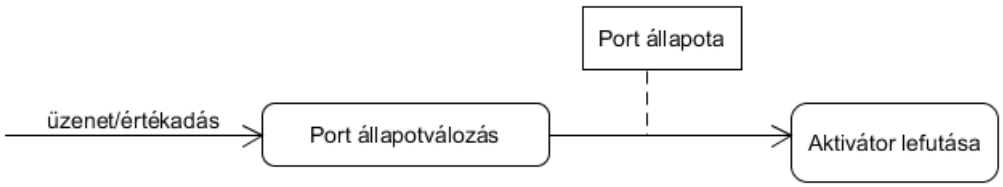
3.2.2. C++-ban

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 történű 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.

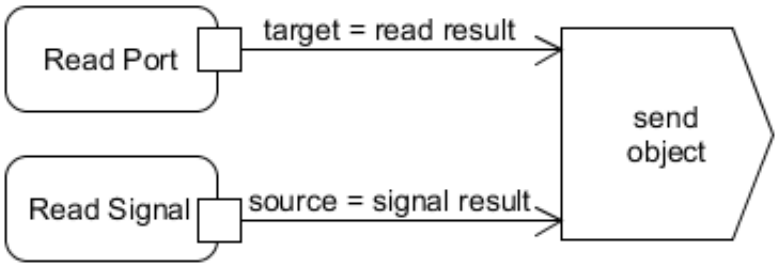
3.3. Portokra való üzenetküldés/fogadás reprezentálása

3.3.1. UML-ben

Üzenet küldés

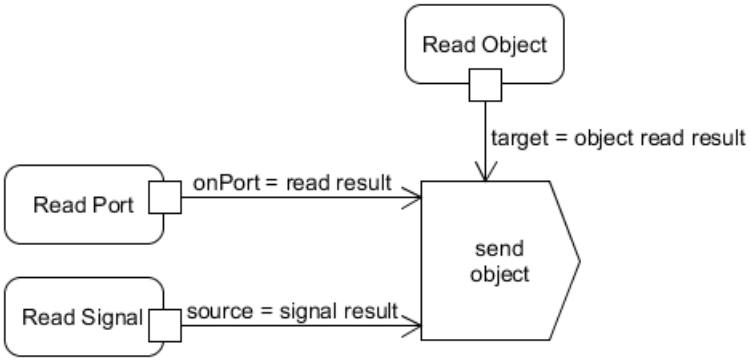
A *SendObjectAction* két paraméterből áll, egy célobjektumból és egy üzenetből. A célobjektum valamilyen *InputPin*, vagyis valamilyen akciónak az eredménye. Mivel a port egy strukturális elem, így a referenciáját egy *ReadStructuralFeatureAction*-nal megszerezhetjük.

- 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.



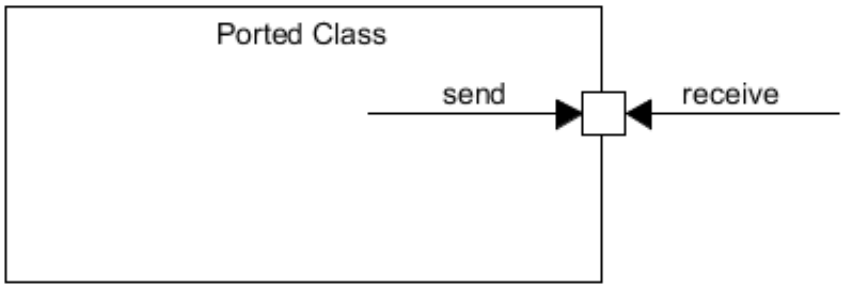
3.3.2. C++-ban

Üzenet küldés

Ez esetben már a *send* művelet végrehajtási szemantikája érdemes számunkra. A *send* művelet azt jelenti, hogy a Portra teszünk egy üzenetet, melyet továbbítunk egy másik komponens 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. 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. receptionöket.

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 üzentet 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.

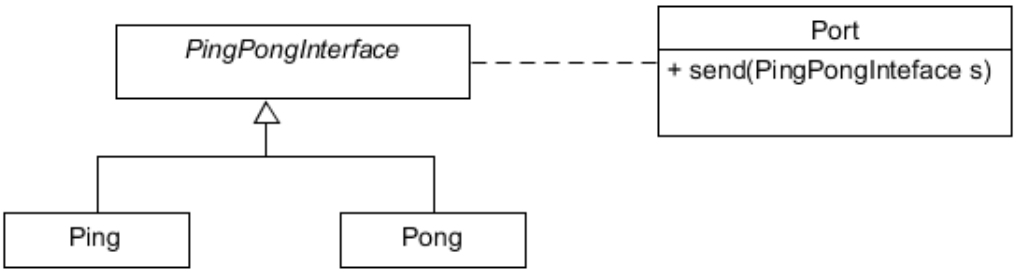
Van interfész

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:

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 receptionjai 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:

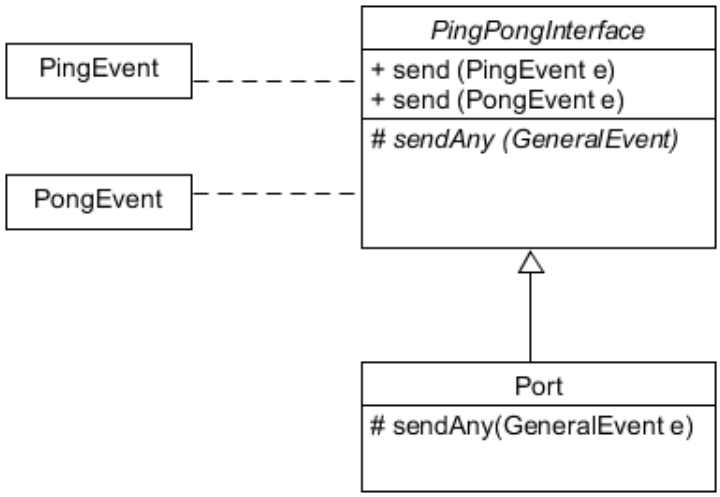
- Egyszerű megvalósítás, kevés extra kód generálásra van szükség hozzá, az interfészek lényegében csak üres, jelző osztályok
- Egyetlen *send* függvénnyel meg lehet oldani a validációt, így a kód a lehető legkisebb lesz, nem lesz technikai redundancia

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 in-

terfészen 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 fo-

gadni. 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.

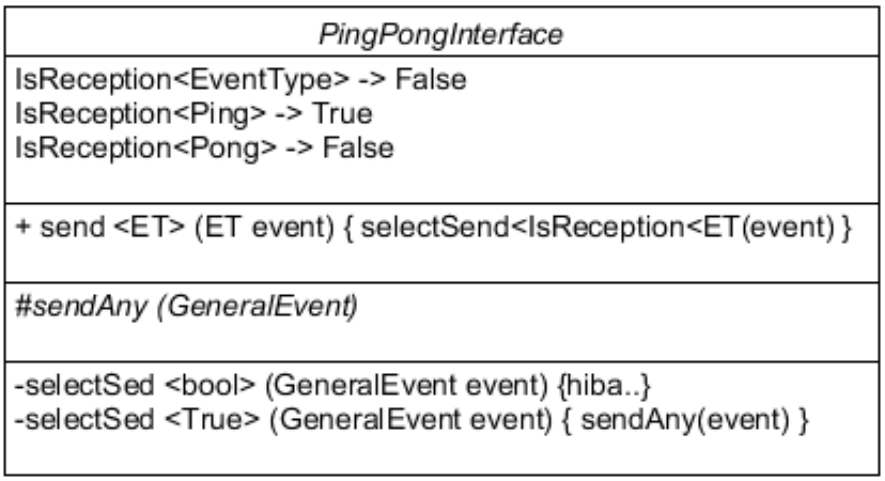
Cél szintaxis: *using PingPongInf = GeneralInterface¡Ping,Pong¿*

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* [9] 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

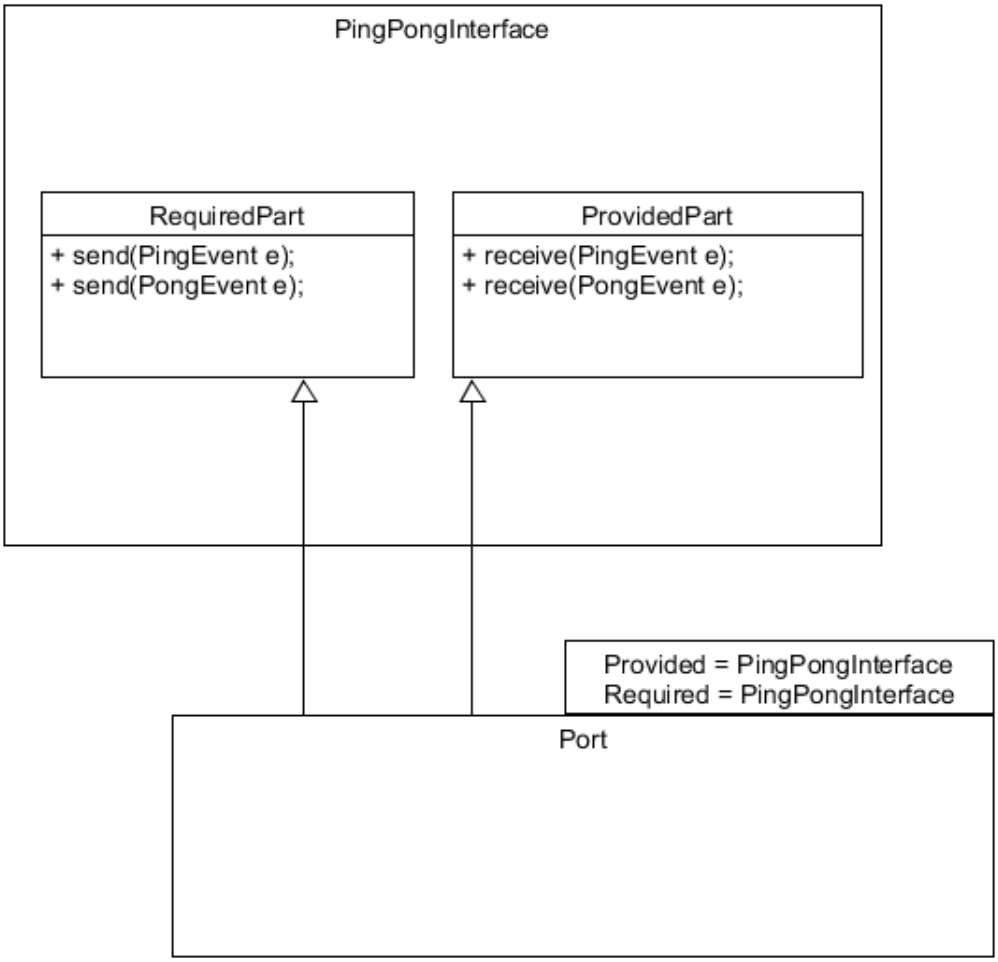
Az UML szabvány szerint egy *konnektor* csak olyan portok között húzódhat, melynek a szolgáltatott és elvárt interfészei kompatibilisek egymással. A port egy kommunikációs csomópontot reprezentál, a típusa azonban adattag révén tetszőleges lehet. A modellosztály számára a port szolgáltatásokat nyújt 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 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 receptiön függvényeknek 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



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 lére, 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.

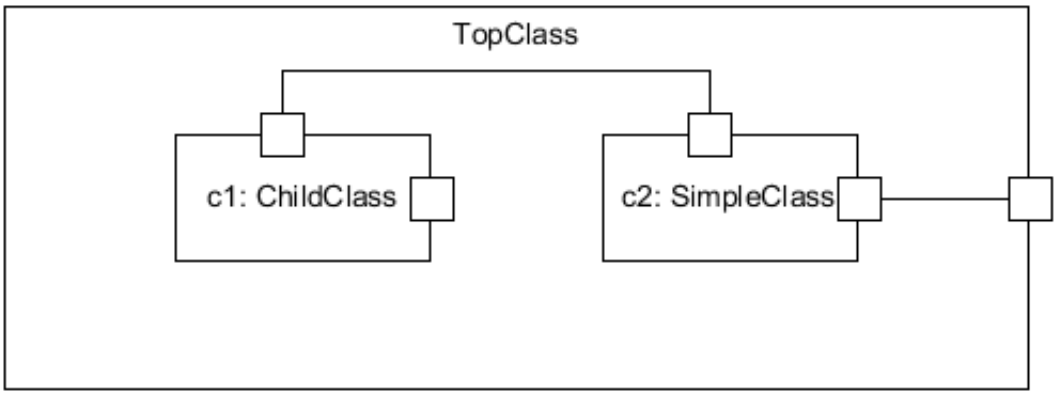
Összekapcsolás művelet reprezentációja UML-ben

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.

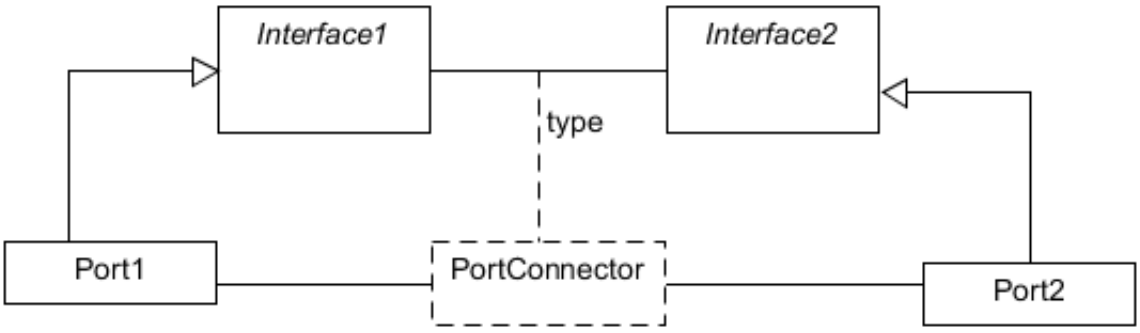
1. Mivel a Portok adattagok az UML-ben (Property-k) így két portot kölcsönösen értékül adhatunk egymásnak. Ez egy lehetőség lehet kifejezni az összekapcsolást, szabványos modell keletkezik, de mégsem ez a helyes szabványa Portok összekapcsolásának. A probléma ugyanis az lesz, hogy bármilyen két portot 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, melyet a szabvány definíciója is sugall. 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 rendelkeznek 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 továbbítania kell ezt a vele kapcsolatban álló port felé. Ehhez mindenképpen valami referenciát kell tárolni a kapcsolat Portra. Ezt alapvetően kétféle képen érhetük el:

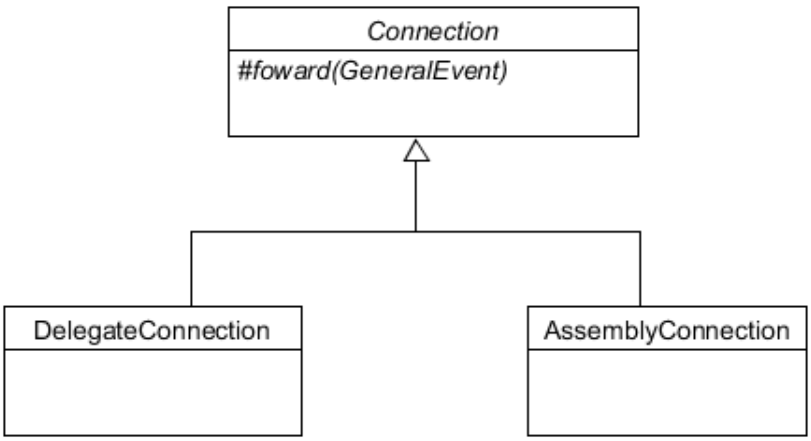
1. Egy globális adatszerkezetben, példásul egy asszociatív tömbben megmondjuk, melyik porthoz ki kapcsolódik. Ennek előnye, hogy nem a Port kódja semleges marad, nem kell belekódolni a kapcsolatokhoz. Hátránya az, hogy egy párhuzamos környezetben egy globális adatszerkezetet kell manipulálni, így sok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



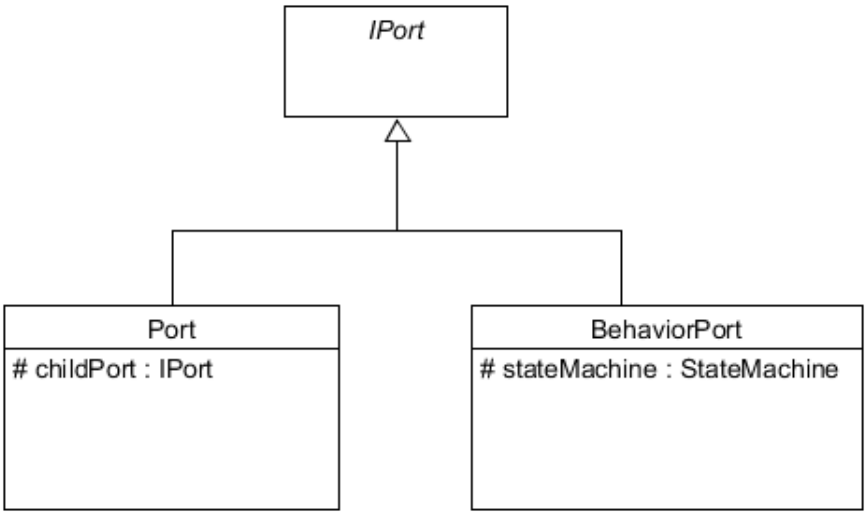
2. Delegáció jelző használata: Marad a Port referencia, maga a Port dönti el, melyik művelet hívására van szükség a jelző alapján. Ez a megoldás kevésbé letisztult, de hatékonyabb.

Vizsgáljuk meg az üzenet fogadás szemantikáját is. A fogadás művelet végrehajtási szemantikája attól függ, hogy állapotgéppel összekapcsolt Portrol (Behavior Port) beszélünk-e vagy 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

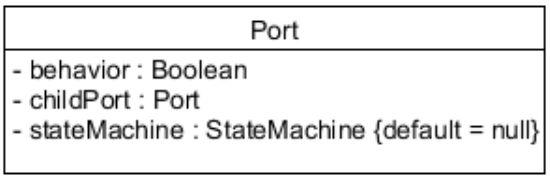
szintén a fent említett két lehetőségünk van, a virtualizáció, illetve a behavior jelző. Egy *Behavior Port* esetén nem csak a *receive* művelet törzse különbözik, de más-más adattagokkal dolgozik a két port. Állapotgép összekapcsolás esetén referenciát kell szereznünk az állapotgépet birtokló objektumra, 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.

Különböző port típusok megvalósítása szemléltetve

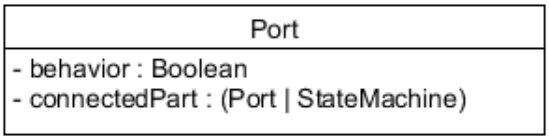
1. Leszármazással:



2. Behavior jelzéssel:



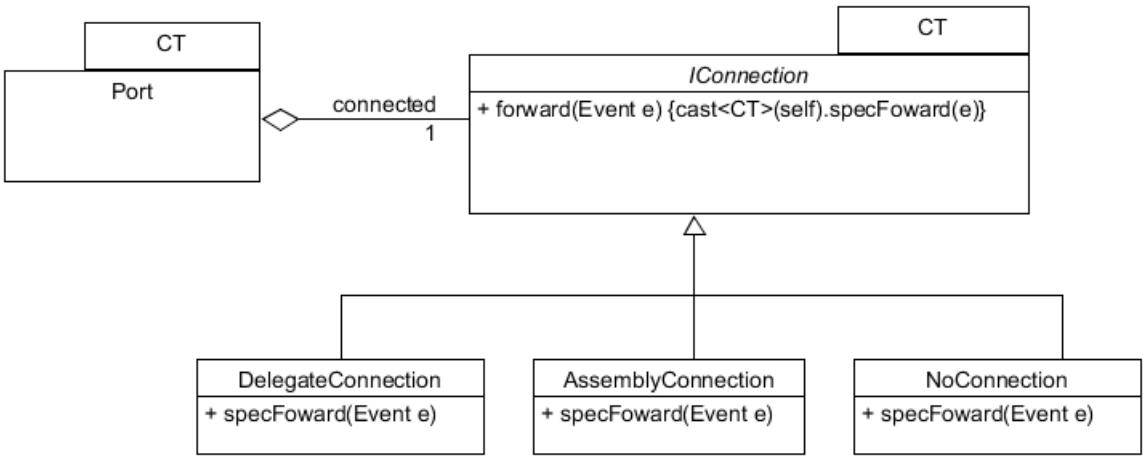
3. Unio típussal:



Hasonlóan az interfészekhez, a kapcsolatok esetén is a kevert megoldás lenne a leghatékonyabb. Ha dolgozhatnánk leszármazással, ugyanakkor nem kéne virtualizációt használni. Erre egy kézenfekvő megoldás az lenne, ha a leszármazáskor megadnánk az interfésznek, ki a leszármazottja.
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.
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. 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 megspórolhatjuk a benne eltárolt port referenciát.

Megoldás szemléltetése

1. *NoConnection* leszármazással:



2. *NoConnection* specliazizációval:

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. 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 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. 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 redundáns, hibás modellt írtunk, mellyel most nem foglalkozunk. Ennél fogva, a konnektorból nem csak annak típusát, de azt a szerep nevet 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 sablon specializációval 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.

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 kifele 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 érkezhet, 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. [10] Most vizsgáljuk meg, hogyan tudjuk kiterjeszteni ezt a táblát.

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 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 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 maskban azokat a portokat, melyek üzeneteit megfigyeli az adott átmenet. Mivel egy üzenet továbbra is csak egy helyről érkezhet, így elég lenne csak azt vizsgálunk, 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.

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

Egy FMU (Functional Mockup Unit) alatt egy olyan funkcionális egységet értünk, mely az FMI (Functional Mockup Interface) [11] szabvány segítségével tud kommunikálni a külvilággal. Olyan folytonos modellekkel való kommunikációra használjuk őket, melyek adott időközönként mintát vesznek a külvilágból (pl. megmérik a levegő hőmérsékletét), majd ezeket az input adatokat elküldik egy FMU-nak, mely eszerint változtatja meg a belső állapotát. Az FMU logikája tehát lehet egy diszkrét modell, mely leírható állapotgépekkel és átmenetekkel. Az FMI egy C-ben írt szabvány, így a C++ által generált modell összecsomagolható egy FMU-vá. [12]
Ezt az alábbi módon valósíthatjuk meg: Létre hozunk 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 feldolgozása, 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 *Environmentet*.
Az asszociáció mellett azonban van egy tisztább lehetőségünk is, ha portokkal dolgozunk. Egy FMU-t felfoghatunk egy olyan strukturális rendszernek, melynek két eleme van:

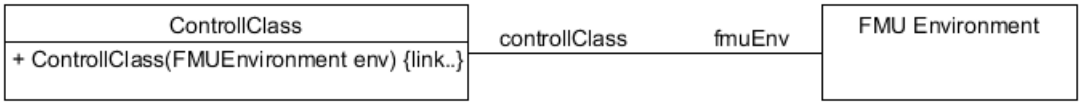
1. Az *FMUEnvironment*, mely megvalósítja az FMI-t. (Init FMU, DoStep, stb..)

2. Az FMU működését leíró *txtUML* modell.

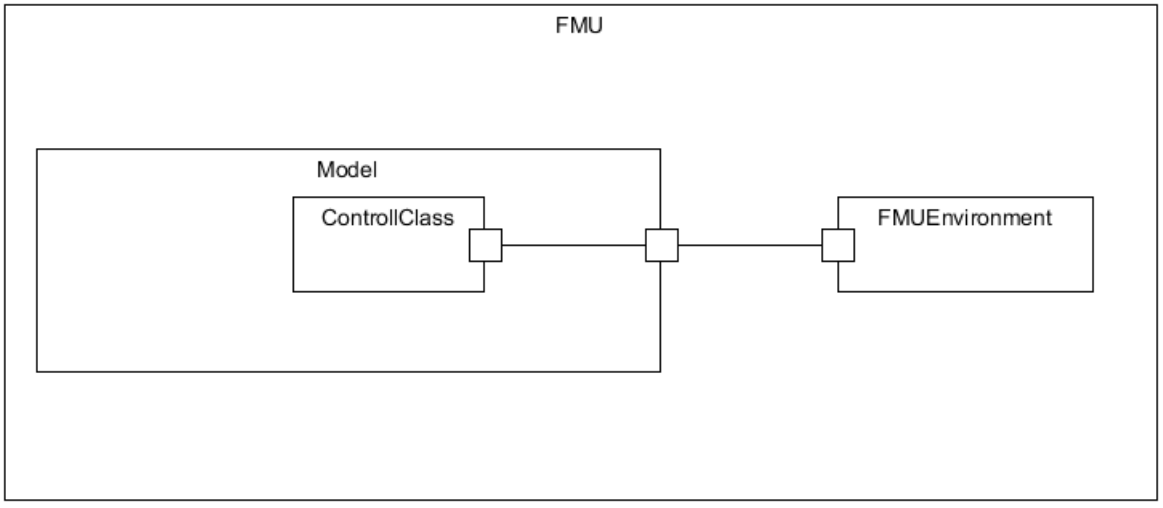
Ezek után a környezet és a modell egy Assembly kapcsolat segítségével lesznek össze kötve. Így a környezetnek nem kell ismernie egy speciális kontroll osztályt, mellyel asszociációban van, de még a modell-re sem kell referenciát birtokolnia, csak a saját portját kell ismernie. A modell osztály pedig 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.

– Ha nem szeretnék FMU generálást, akár arra is jobba lehetőségünk van, hogy figyelmen kívül hagyjuk a modellen kívüli szereplőket, mivel annak semmi külső függősége nem lesz így.

– Tisztább struktúra, külső kódból is könnyen felépíteni ezt a függőséget, mert a modell egy portjára kell rákötni a külvilágot.

Hátrányok:

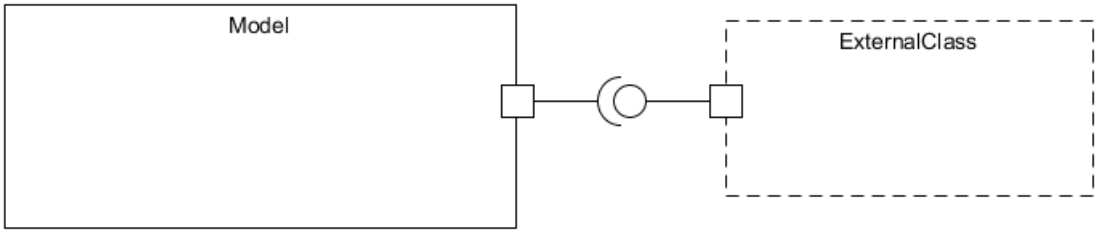
– Nehezebb leírni ezt a strukturális felépítést *txtUML*-ben.

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 portjára várjuk az üzeneteket.

4.2. Külső komponensek bekötése a modellbe

Általánosságban elmondható, hogy ha szeretnénk egy külső komponenst bekötni a modellünkbe anélkül, hogy a generált kódhoz kézzel hozzá kéne nyúlnunk, akkor a portok nagy szerepet játszanak. 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

Anélkül, hogy precízen megvizsgálnánk, pontosan hogyan tudunk párhuzamosítani portok segítségével általánosan elmondható, ami már az eddigiekből is kiderült: a portok sokkal könnyebben tudnak függetleníteni két különböző logikai egységet. 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 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, mely túlmutat a diplomamunka témakörén.

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 processzoron is futhat, ez esetben a portoknak valamilyen interprocessz kommunikációt 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 tanóra 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, ne pedig csak a levegőbe lógjanak. 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

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.

5.1. 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.

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 *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 környzet 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. 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.

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 **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 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.

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

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 komponensen. 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.

Irodalomjegyzék

[1] OMG, Superstructure

Unified Modeling Language (OMG UML)

[2] Bran Selic, ObjecTime Limited

Jim Rumbaugh, Rational Software Corporation *Using UML for Modeling Complex Real-Time Systems*

ftp://ftp.omg.org/pub/umlrtf/UML_rt_ext_F56dist.pdf

[3] Gábor Ferenc Kovács, Ádám Ancsin, Gergely Dévai,

Textual, executable, translatable UML, 14th International Workshop on OCL and Textual Modeling, 2014.

[4] Object Management Group.

Action Language for Foundational UML (ALF), standard, version 1.0.1.

http://www.omg.org/spec/ALF/ 2013

[5] Object Management Group.

Semantics of a Foundational Subset for Executable UML Models (fUML), standard, version 1.1

http://www.omg.org/spec/FUML/1.1/ 2013

[6] Andrew Forward, Omar Badreddin, and Timothy C Lethbridge

Umple: Towards combining model driven with prototype driven system development

In Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on, pages 1–7. IEEE, 2010.

[7] OneFact.

BridgePoint xtUML tool.

http://onefact.net

[8] StartUML Tool

http://staruml.io/

[9] TypeLists and a TypeList Toolbox via Variadic Templates

https://www.codeproject.com/Articles/1077852/TypeLists-and-a-TypeList-Toolbox-via-Variadic-Temp

[10] Hack János *C++ kódgenerálás futtatható UML modellekből* 2015 Ezt kibővíthetjük az alábbi módokon:

[11] Functional Mock-up Interface for Model Exchange and Co-Simulation

version 2.0, July 25, 2014

https://fmi-standard.org/docs/2.0.1-develop/

[12] Interoperability of the standards Modelica-UML-FMI

Nov 17, 2017