Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Teszteszmodell-definíció TTCN-3-ban

Szerző:
KRIVÁN BÁLINT

Konzulensek:
DR. KOVÁCS GÁBOR
ERŐS LEVENTE

2012. OKTÓBER 26.

Tartalomjegyzék

1. Bevezetés **2**

2. Elméleti háttér **4**
2.1. Tesztelés 4
2.1.1. A tesztelés evolúciója és a modell alapú tesztelés 5
2.2. Modellezésre használt absztrakciók 6
2.2.1. Véges állapotgépek 6
2.2.2. Kiterjesztett véges állapotgépek 7
2.3. TTCN-3 8
2.3.1. TTCN-3 kód általános struktúrája 9
2.3.2. Típusok, sablonok 11
2.3.3. `alt` és `altstep` 13

3. TTCN-3 tesztmodell **15**
3.1. Forráskód tagolása 17
3.1.1. Komponensek, portok, adattípusok, sablonok definíciós modul 17
3.1.2. EFSM modellek viselkedését leíró modul 19
3.1.3. Környezet definíciója 20
3.1.4. Vezérlő rész 20

4. A LAPB protokoll tesztmodellje TTCN-3-ban **23**
4.1. Állapotgráf-leíró formátum 24
4.2. Gráf reprezentáció 25
4.3. LAPB protokoll gráfleírójának elkészítése 26
4.4. TTCN-3 kód generálása 29
4.5. Bejáró, kereső algoritmusok a gráfon 31
4.5.1. Megjelölt éleket tartalmazó séta 31
4.5.2. Hurkok keresése 33

5. Összefoglalás **34**

1. fejezet

Bevezetés

Napjainkban távközlő rendszerek teszteléséhez a tesztmérnökök többsége a TTCN-3 (Testing and Test Control Notation) nyelvet használja az iparban a tesztesetek leírására. A bevett gyakorlat, hogy a teszteseteket egy informális leírásból, specifikációból készítik el, melyeket aztán egy tesztvégrehajtó környezetben futtatnak. A tesztelés alatt álló rendszer (SUT – System Under Test) viselkedését vizsgálják, hogy az megfelel-e a specifikációban meghatározottaknak. A tesztesetek megtervezése, és azok manuális megírása sok időt emészt fel, és ha az implementáció változik, akkor újabb emberi munkaórák belefektetéssel lehet csak a teszteseteket átalakítani.

A modell alapú tesztelésnél azonban nem teszteseteket írnak a tesztmérnökök, hanem az adott tesztelendő rendszerből bizonyos szempontok szerint absztrakt modellt építenek, melyek bizonyos részleteket figyelmen kívül hagynak (pl. környezetfüggő kérdések: operációs rendszer, hardver). A felépült modellből pedig automatikusan, adott irányelvek és tesztcélok alapján generálnak teszteseteket. A modell megadáshoz és abból a tesztesetek generálásához sokféle eszköz létezik a piacon, melyek több különböző megközelítést használnak, a belső modellt eltérő nyelven (UML, Stateflow), eltérő absztrakcióval (véges állapotgépek, markov-láncok) reprezentálják, ebből adódóan bizonyos limitációkkal, problémákkal küzdenek. A Confirmiq Qtronic[7] eszköz például nem támogat nem-determinisztikus SUT-okat. MOTES[11] esetében a rendszer viselkedése csak egyetlen véges állapotgéppel adható meg, nem lehet több modell interakcióját leképezni.

A piacon lévő eszközök eltérő célokra vannak kihegyezve és így eltérő megközelítést kívánnak meg, emiatt az alkalmazkodás, vagy egyikről másikra való áttérés nehézségekbe ütközik. Tekintve, hogy a modell alapú eszközök TTCN-3 teszteseteket (is) generálnak és a tesztvégrehajtó eszközök általában ilyen nyelvű teszteseteket hajtanak végre, ezért jogosan vetődik fel az a gondolat, hogy a rendszerek modelljét is

e nyelven készítsük el. Ekkor egyetlen nyelv és egyetlen eszköz segítségével megvalósítható a modell alapú tesztgenerálás és a tesztvégrehajtás, mellyel anyagi és ember költségek spórolhatóak meg. A tesztmodellekben a korábban definiált adattípusok újrafelhasználhatóak, így az egész kód sokkal kompaktabb és átláthatóbb lesz. Mivel a tesztmérnököknek nincs szükségük a TTCN-3-on kívül egy másik nyelv megismerésére, megtanulására sem, így a tesztelési ciklus, a tesztmérnökök képzése is lerövidül. A végső cél konformancia tesztesetek hatékony és automatikus előállítása és futtatása.

A dolgozat alapjául a [6] cikk szolgál, mely – egy metamodell megadásával – az alapokat fekteti le egy tesztmodell TTCN-3-ban történő definiáláshoz. A cikk egy egyszerű protokollon keresztül mutatja be az ötletet, e dolgozat azt demonstrálja, hogy egy valós méretű protokollon is jól alkalmazható a módszer, a metamodell alkalmas valós méretű modellek leírására.

A dolgozat további része a következőképpen tagolódik. A második fejezet az elméleti hátteret mutatja be; szó esik a tesztelésről, a véges állapotgép alapú modellekről, és a TTCN-3 tesztleíró nyelvről általánosságban. A harmadik fejezet bemutatja a TTCN-3 alapú modellezést és kitér az EFSM-mel történő megfeleltetésre. A negyedik fejezet a módszert egy valós protokollon demonstrálja. Végül pedig az utolsó fejezet összefoglalja a TTCN-3 alapú tesztmodellezés tapasztalatait, lehetőségeit és kihívásait.

2. fejezet

Elméleti háttér

Ebben a fejezetben a dolgozat során használt fogalmak és az azokhoz kapcsolódó definíciók elméleti háttere kerül bemutatásra. A 2.1. szakaszban reaktív rendszerek tesztelésről és a modell alapú tesztelésről lesz szó. Ezt követően a 2.2. szakasz ismerteti a véges állapotgépek és a kiterjesztett véges állapotgépek definícióját, felépítését, továbbá azt, hogy egy rendszer miképpen modellezhető a segítségükkel. Az utolsó, a 2.3. szakasz a TTCN-3 tesztleíró nyelvet általánosságban mutatja be, kitérve a struktúrára és a fontosabb nyelvi elemekre, kulcsszavakra.

2.1. Tesztelés

A tesztelés jelentős szerepet tölt be a legtöbb szakterületen, így az IT világban is, olyannyira, hogy már a projektek tervezési szakaszában számításba veszik a tesztelésre szánt emberi és anyagi ráfordításokat, melyek egyes becslések szerint a teljes projektköltség legalább negyedét teszik ki.

Kifejezetten távközlési protokollok tesztelésénél domborodik ki a *konformancia tesztelés* szerepe, hiszen a kommunikációs rendszerek esetén jól definiált, pontos interfészek vannak, melyeket a szabványokban definiált módon kell megszólítani, kezelni. A távközlő rendszereknek két sajátosságuk van más szoftverekhez képest: az események nagy száma és a szigorú időzítési követelmények. Az elkészült eszközök bonyolult és drága mivolta miatt fontos, hogy ezek a lehető legnagyobb gondossággal legyenek tesztelve. A konformancia tesztelés [3] során egy rendszerről készült implementációt hasonlítunk össze egy, a specifikációból származtatott tesztmodellel tesztesetek segítségével. Ez a tesztmodell vagy szoftverkód formájában létezik, vagy nem áll össze kóddá és a tesztmérnök a tapasztalataira hagyatkozik a tesztesetek megírásakor. A konformancia tesztelés alatt álló rendszerünket (SUT – System Under Test) fekete

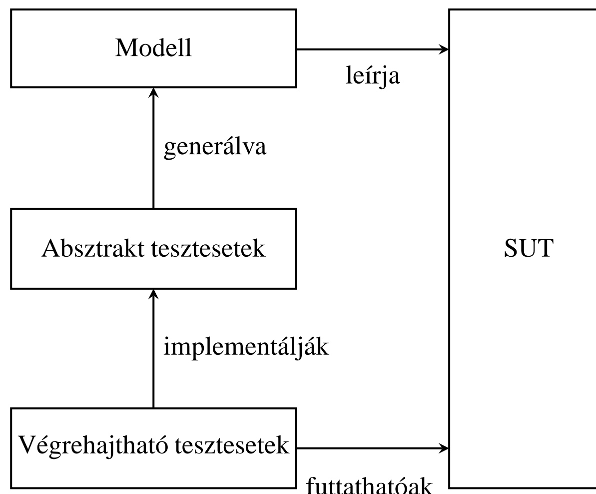
dobozként képzeljük el, azaz a belső felépítéséről nincsenek információink, pusztán a be- és kimeneteket tudjuk befolyásolni illetve megfigyelni a vezérlési és megfigyelési pontokon (PCO – Point of Control and Observation) és az így kapott eredményeket vethetjük össze az elvártakkal.

2.1.1. A tesztelés evolúciója és a modell alapú tesztelés

Ahogy [6]-ban is kifejtésre kerül, a tesztelés automatizáltsága több szinten is elképzelhető. Még napjainkban is igen gyakori a kézi, teszt forgatókönyv alapú tesztelés, azaz egy adott tesztelendő rendszerhez a felhasználó közvetlenül hozzáfér, és azon kézzel végzi el a forgatókönyvben megadott lépéseket, majd annak eredményét összeveti az előre meghatározottal. Ha ez a kettő egyezik, akkor a teszt sikeres, ellenkező esetben sikertelen. Ez egy egyszerű rendszernél viszonylag rövid tesztelést jelent, és talán egyszerű is, de ahogy a komplexitás növekszik a fejlesztés során, úgy egyre körülményesebbé válik, illetve rengeteg emberi munkaidőt felemészt.

A következő szint, amikor az előbbihez hasonlóan tesztforgatókönyvekből (melyeket még mindig kézzel állítanak össze a specifikációnak megfelelően) indulunk ki, de ezeket nem egy ember hajtja végre, hanem egy automatizáltan futtatható szkriptként írják le, majd egy erre kifejlesztett tesztrendszerben futtatják.

A harmadik, a legújszerűbb ezek közül, a modell alapú tesztelés [17] (lásd 2.1. ábra).



2.1. ábra. Modell alapú tesztelés

Az előzőektől eltérően, nem teszteseteket gyártunk a specifikációk és tesztforgatókönyvek alapján, hanem először egy modellt alkotunk a SUT-ról, amely elrejti előlünk

az implementációs részletek nagy részét. Ez alapján, már megfelelő eszközök és algoritmusok ismeretében, teszteseteket generálhatóak nagyrészt automatizáltan [1]. Ezek még nem a konkrét eszközre készülnek, hanem az általunk készített modellre, így a modell által használt absztrakt műveletek és értékek szerepelnek bennük. Annak érdekében, hogy a tényleges SUT megvalósításon futtatni tudjuk, konkretizálnunk kell azért, hogy ténylegesen végrehajtható teszteseteket kapjunk. Jól látható, hogy itt már a folyamat legnagyobb része automatizálható – vagy legalábbis nagyon afelé tartunk, hogy az lesz – és csak az első lépés, a modell előállítása igényel komoly emberi erőforrást, de ez is általában kisebb feladat, mint a tesztesetek kézzel való megírása.

2.2. Modellezésre használt absztrakciók

Ez a szakasz a reaktív rendszerek modellezésére használt legfontosabb absztrakciókat tekinti át, először a véges állapotgépeket majd a kiterjesztett véges állapotgépeket.

2.2.1. Véges állapotgépek

A véges állapotgépek (továbbiakban FSM - Finite State Machine) Mealy-modellnek megfelelő definíciója a következő ötössel [10] adható meg:

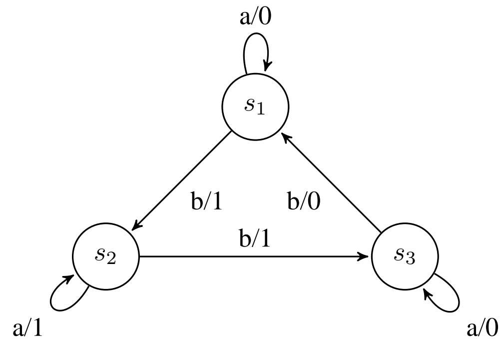
$$M = (I, O, S, \delta, \lambda)$$

Ahol I és O a lehetséges be- illetve kimeneti szimbólumok halmaza, S az állapotok halmaza, δ az állapotátmeneti függvény, λ pedig a kimenet függvény. A δ adja meg, hogy melyik állapotban melyik bemenet hatására melyik állapotba megyünk át ($\delta : S \times I \to S$), a λ pedig, hogy ekkor melyik szimbólum jelenik meg a kimeneten ($\lambda : S \times I \to O$).

Az állapotgépek ábrázolhatóak irányított gráffal, az úgynevezett állapotgráffal, ahol csomópontokkal jelöljük az állapotokat, és a köztük lévő élek az állapotátmenetek, melyeken feltüntetjük, hogy mely bemenetek hatására hajtódik végre az átmenet és ekkor mit adunk a kimenetre. Egy lehetséges példa a 2.2. ábrán látható.

Ekkor három állapotunk van ($S = \{s_1, s_2, s_3\}$), kettő bemeneti ($I = \{a, b\}$) és kettő kimeneti ($O = \{0, 1\}$) szimbólumunk. A gráfok közötti élekről pedig a δ és λ függvények értékeit olvashatjuk le, például: $\delta(s_1, a) = s_1$, $\delta(s_1, b) = s_2$ és $\lambda(s_1, a) = 0$ illetve $\lambda(s_1, b) = 1$.

Számos alkalmazási területen, például a kommunikációs protokollok esetében, a specifikációk egy FSM-mel leírhatók [2]. Az FSM alapján algoritmikusan tesztesetek is generálhatóak [10]. Egy valós méretű probléma esetében azonban, az úgyne-



2.2. ábra. Véges állapotgép állapot-átmeneti gráfja

vezett „állapottér-robbanással” kerülünk szembe. Valós méretű rendszerek esetén az adatstruktúrák értékkészletének növekedése miatt a rendszer állapotainak száma exponenciálisan nő, minden egyes új bit egy változó doménjében duplikálja az állapotteret, és ha bemeneti/kimeneti paraméterként is szerepel, az I/O ábécét is. Kezelhető méretű modellek létrehozása végett szükség volt egy másik absztrakcióra, a kiterjesztett véges állapotgépre (továbbiakban EFSM – Extended Finite State Machine).

2.2.2. Kiterjesztett véges állapotgépek

A modell vezérlő állapotainak terét a változók halmazából kiválasztott néhány változó értékkészlete adja. Az állapotváltások mellett a változók állapotváltozásait is fontos megfelelően kezelni, így az átmeneteken predikátumok és akciók bevezetése is szükségessé vált. Az EFSM egy lehetséges definíciója [10]:

$$EFSM = (I, O, S, \overrightarrow{x}, T)$$

ahol I és O a be- illetve kimeneti szimbólumok halmaza, melyek lehetnek paraméterezettek is, S az állapotok halmaza, $\overrightarrow{x}$ a változókat tartalmazó vektor, T pedig az átmenetek halmaza.

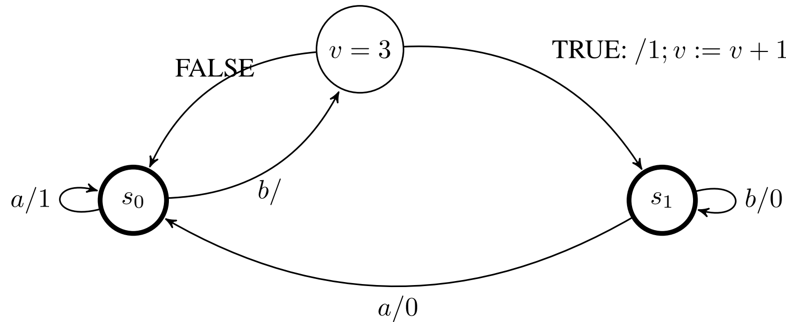
Egy átmenetet a következő hatos definiál [10]:

$$t = (s_t, q_t, a_t, o_t, P_t, A_t) \qquad \text{ha } t \in T$$

ahol
s_t: kezdő állapot
q_t: végállapot
a_t: bemenet
o_t: kimenet

$P_t(\overrightarrow{x})$: a predikátum függvény, amely igaz vagy hamis az adott változókra nézve
$A_t(\overrightarrow{x})$: az akció függvény, amely a változóknak ad új értéket: $\overrightarrow{x} := A_t(\overrightarrow{x})$

Az állapotgráfos megjelenítéshez szükség van egy gráfmodell megadására. Az állapotokhoz és az állapotokhoz tartozó átmenetek predikátumaihoz csomópontokat rendelünk. Az állapotcsomópontokból a többi állapotcsomópontba valamint a predikátumcsomópontokba élek vezetnek, melyek triggere az adott átmenet bemenete. A predikátumcsomópontokból pedig 2 él vezet ki, egy az átmenet kezdőállapotába (ha a predikátum nem teljesül), egy az átmenet végállapotába (ha a predikátum teljesül).



2.3. ábra. Kiterjesztett véges állapotgép

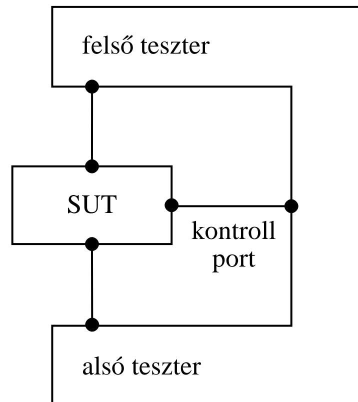
A fenti ábra a következő kiterjesztett véges állapotgépet mutatja be: $S = \{s_0, s_1\}$, $I = \{a, b\}$, $O = \{0, 1\}$ és $\overrightarrow{x} = (v)$. Három predikátum nélküli átmenet van, ahol akció sincsen: s_0 állapotban a bemenet hatására 1 lesz a kimenet és nem történik állapot váltás s_1-ből b hatására 0 lesz a kimenet és nem vált állapotot, míg a hatására szintén 0 a kimenet, de állapotot vált s_0-ba. s_0-hoz pedig tartozik egy predikátumos átmenet is: b bemenet hatására ha $v = 3$ predikátum teljesül, akkor 1-t ad ki a kimeneten, növeli v értékét és s_1 állapotba vált, ha nem teljesül, akkor nem történik semmi, maradunk s_0 állapotban.

A bemenetről jöhetnek paraméterek, melyek akár predikátumban, akár akcióban is felhasználhatóak, mint argumentumok, valamint a kimeneteket is paraméterezhetjük, tehát az input események függvényhívásoknak, az output események pedig ezen hívások visszatérési értékének felelnek meg. A dolgozat későbbi fejezetei bemutatják, hogy a modellezésnek ezen tulajdonsága a protokollok leírásánál jól kihasználható.

2.3. TTCN-3

A TTCN-3 egy tesztleíró és tesztvezérlő szkriptnyelv [3], amelyet főképp kommunikációs rendszerek konformancia-tesztelésénél alkalmaznak. Egyik legnagyobb előnye, hogy absztrakt tesztesetek írását teszi lehetővé, így nem kötődik szorosan a teszt-

végrehajtó környezethez, ahhoz csak a legutolsó fázisban lesz kötve, amikor a tesztet végrehajtjuk, így a kód rendkívül rugalmasan mozgatható különböző rendszerek között. Tervezésénél ügyeltek arra, hogy a megszokott szintaktikához hasonló nyelvi elemeket tartalmazzon, így egy C/C++-ban járatos embernek a kód megértése, és az ahhoz való alkalmazkodás nem jelent akadályt.



2.4. ábra. Egyszerű tesztkonfiguráció

Adott egy rendszer (SUT), amit tesztelésnek vetünk alá, hogy megvizsgáljuk, az elvárt módon viselkedik-e (lásd 2.4. ábra). Ekkor a tesztesetekben megadott módon gerjesztjük a SUT komponenst, amely erre üzenetekkel válaszol (elképzelhető természetesen az is, hogy az első üzenet nem a SUT felé, hanem a SUT felől érkezik). Amennyiben a dialógus az elvártakkal megegyezik, úgy pozitív, különben negatív vagy semleges ítélet (utóbbi esetben a teszter nem tud meggyőződni arról, hogy a SUT megfelelően működik-e, részletekért lásd [3, 4. oldal]) születik a teszt sikerességét tekintve.

A dolgozatnak nem célja a teljes TTCN-3 nyelv mélyreható bemutatása, csak egy átfogó kép nyújtása, kitérve a kód strukturálására és néhány olyan fontosabb nyelvi elemre, mely az eddigi nyelvektől eltérő megközelítést kíván meg és tesz lehetővé. Bővebb leírás [5]-ban illetve az ETSI szabványban található [4].

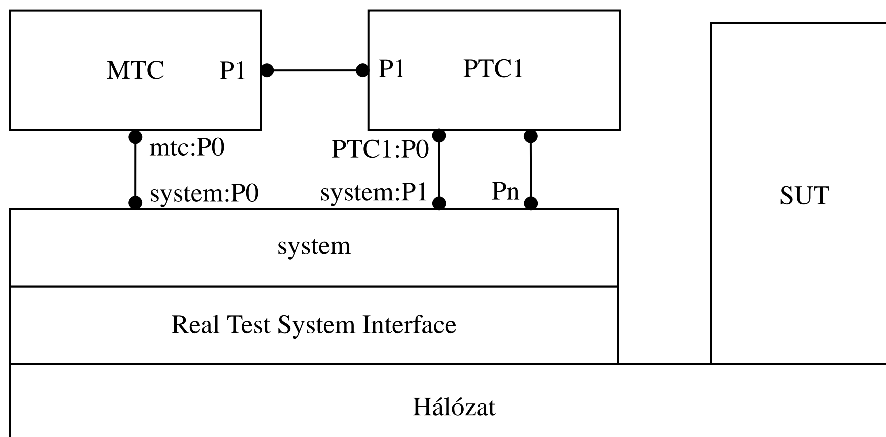
2.3.1. TTCN-3 kód általános struktúrája

A TTCN-3 kód a legmagasabb szinten modulokba tagozódik [3, 5, 18]. A modulok két részből állnak: egy definíciós és egy vezérlő részből. A definíciós részben megadottak máshol is felhasználhatóak a modulban, például a vezérlő részben is. A vezérlő rész írja le a tesztesetek végrehajtási sorrendjét. A tesztesetek a definíciós részben definiáltak, és a vezérlő részben hivatkozottak.

A definíciós részben az alábbi elemek szerepelhetnek:

- importálás más modulokból,
- modul paraméterek,
- adattípusok,
- szignatúrák,
- port típusok,
- komponens típusok (amelyekből a tesztkonfigurációnk állni fog),
- konstansok,
- `altstep`-ek (alternatív viselkedést leíró függvények),
- függvények, és természetesen
- tesztesetek.

Egy tesztesetnél meg kell adni, hogy azt milyen típusú főtesztkomponensen (MTC) futtatjuk és a SUT milyen típusú komponenst valósít meg. A törzs eleje általában az adott tesztesetben érvényes tesztkonfiguráció létrehozásával kezdődik, azaz megadjuk, hogy milyen komponenseink vannak, azok milyen elrendezésben, hogyan vannak összekötve. Ezt követően már többféle megközelítést alkalmazhatunk a teszttől függően, például: itt írjuk le a rendszer viselkedését vagy azt függvényekbe szervezzük és minden komponensen megadjuk, hogy rajta melyik futtatandó. Az eddigiek mellett még fontos szerepet tölt be a modulvezérlő rész: itt adjuk meg, hogy az előzőekben definiáltak közül, mely tesztesetek és milyen fussanak a tesztelő környezetben. A 2.5. ábrán egy tesztkonfiguráció sablonos rajza szerepel. Az elrendezés a [14]-ban leírt Remote Test Method-nak felel meg, hiszen a SUT felé nem támaszt extra elvárásokat (pl. speciális tesztelőportok), fekete dobozként kezeli azt, és a tesztben résztvevő komponenseink a tesztrendszerrel vannak kapcsolatban, nem a SUT-tal.



2.5. ábra. Tesztkonfiguráció [3]

Egy tesztesetben van egy főkomponens (MTC – Main Test Component), amin a teszteset fut, és emellé példányosíthatunk – nem kötelezően – párhuzamos komponenseket (PTC – Parallel Test Component). Ezeket egymással illetve a SUT-ot megtestesítő *system* portjaival kötjük össze. A TTCN-3 futtatókörnyezet felelős azért, hogy a *system* felé küldött, illetve az onnan jövő üzeneteket a SUT illetve a tesztkomponens számára kódolja, dekódolja.

2.3.2. Típusok, sablonok

A nyelvben megtalálható sok egyéb nyelvből is ismert adattípus, pl.: `integer`, `boolean`, `record` vagy a `charstring`, ezekre a fejezet mélyebben nem tér ki. Típusként használható továbbá az `anytype` [18, 157. oldal] is, amely – mint a neve is mutatja – bármilyen típust helyettesíteni tud. Ez a C programnyelvbeli `union`-nak felel meg, használata is nagyon hasonló.

A `port` típussal a komponensek által használt portokat tudjuk definiálni, hogy azon milyen adattípusok küldhetőek illetve fogadhatóak, vagy hogy milyen függvények hívhatóak rajtuk (lásd eljárás-alapú portok [18, 19-20. oldal]). Álljon itt egy példa szemléltetésként az üzenet alapú portra:

```
type port MyPort message
{
  in integer;
  out boolean;
  inout charstring;
}
```

A fentivel egy *MyPort* nevű porttípust definiáltunk, amelyről `integer` és `charstring` típusú adat fogadható, illetve `boolean` és `charstring` küldhető. A portok analógiába hozhatóak a C nyelvben használt függvényekkel: a bemenetként használt típusok érték szerint, a kimenetként is használtak pedig referencia szerint átadott függvényparamétereknek felelnek meg. Vegyünk egy pillantást az alábbi portra:

```
type port UniversalPort message
{
  inout anytype;
}
```

Az `UniversalPort` nevű porttípus fogadni és küldeni is tud `anytype` típust, azaz lényegében univerzális portként viselkedik. Ez egy olyan C-beli függvénynek feleltethető meg, amely egy univerzális mutatót (`void*`) vár paraméterül.

A `component` kulcsszóval komponenstípusokat tudunk definiálni. Megadhatjuk,

hogy milyen portokkal rendelkezzenek (típus és név), valamint milyen belső változói vannak. Például:

```
type component MyComponent
{
  port MyPort P0;
  var integer counter;
  var boolean hasMagic;
}
```

A fentivel egy *MyComponent* nevű komponenstípust definiáltunk, aminek egy *P0* nevű `MyPort` típusú portja, illetve két változója van, egy *counter* nevű `integer` és egy `boolean` *hasMagic* névvel.

A későbbiek szempontjából a `template` is egy fontos nyelvi elem. Ennek szerepe kettős; egyrészt a bejövő üzenetek illesztésénél használjuk, így el tudjuk dönteni, hogy az adott üzenet megfelel-e egy általunk megadott sablonnak, másrészt előre gyártott üzenetekként küldjük őket (akár paraméterezve is).

```
type record MyRecord {
  charstring msg,
  integer a optional,
  integer b optional,
  integer c optional
}

template MyRecord MyTemplate := {
  msg := "Hello",
  a := ?,
  b := *,
  c := omit
}
```

A fenti kódrészlet első felében egy *MyRecord* nevű `record` típust definiáltunk, a másodikban pedig ehhez készítettünk egy *MyTemplate* nevű sablont, amiben a *msg* értéke szigorúan `"Hello"`, *a* értéke lehet bármi, de legyen megadva, *b* értéke szintén bármilyen értéket felvehet, de akár hiányozhat is, *c* pedig kötelezően nem megadott (kihagyott). Paraméteres sablonra is álljon egy példa:

```
template MyRecord MyParTemplate (charstring msgP) := {
  msg := msgP,
  a := omit,
  b := omit,
```

```
    c := omit
  }
```

2.3.3. `alt` és `altstep`

Amikor TTCN-3-ban egy adott üzenet küldése után, arra több különböző választ várhatunk, akkor az `alt` vezérlési szerkezetet használjuk. Futás során, amikor a végrehajtó környezet egy `alt` blokkhoz ér egy pillanatképet – *snapshotot* – készít a rendszer jelenlegi állásáról és a blokkban szereplő alternatívákon végigmegy. Minden alternatívához tartozik őrfeltétel (ami lehet üres is) és egy adott esemény. Az alternatívákon sorban megy végig és az fut le amelyikre először igaz, hogy az őrfeltétel teljesül és az esemény bekövetkezik. Ha egyik sem teljesül, akkor új snapshotot készít és újrakezdi az alternatívák vizsgálatát. Vegyük például a következő kódot [3]:

```
P0.send("kerdes");
T0.start;
alt {
  [] P0.receive("jo valasz") { P0.send("koszonom!"); }
  [] P0.receive("rossz valasz") { P0.send("nem jo!"); }
  [] T0.timeout { P0.send("Lejart az ido!"); }
}
```

A fenti részlet egy üzenetet küld a `P0` porton és elindít egy számlálót. Ha `"jo valasz"`, vagy `"rossz valasz"` üzenet érkezik, akkor a megadott módon válaszol, de akkor is jelez, ha lejár az időzítő.

Az `altstep` nagyon hasonló a fentihez, lényegében kiszervezett alternatívákat tartalmaz, azzal a nagyon nagy különbséggel, hogy pillanatkép nem készül, hanem meglévőt használ fel. Ez magával vonzza azt, hogy csak `alt` vezérlési szerkezetből hívható.

```
altstep myAltstep() runs on MyComponent {
  [] P0.receive("jo valasz") { P0.send("koszonom!"); }
  [] P0.receive("rossz valasz") { P0.send("nem jo!"); }
  [] T0.timeout { P0.send("Lejart az ido!"); }
}

// ...

  alt {
    // ...
```

```
    [] myAltstep();
    // ...
  }
```

A fenti egy példát mutat, hogyan lehet kiszervezni alternatívákat `altstep`-be, és azt meghívni egy `alt` utasításból.

3. fejezet

TTCN-3 tesztmodell

A TTCN-3 alapok után ez a fejezet egy metamodellt ír le, vagyis azt, hogy hogyan adható meg egy EFSM modell ebben a nyelvben, a kód egyes részei hogyan rendezhetőek el. A metamodell alkalmazását egy konkrét példán keresztül be is mutatja.

Egy EFSM modellt egy TTCN-3 komponensként képzeljünk el [6]. Két részre tagoljuk, egyrészt egy definíciós részre, másfelől pedig egy, a modell viselkedését leíró részre. Az első részben definiáljuk a modell által használt portokat és az ezeken küldhető és fogadható üzeneteket, valamint, egy a modellt reprezentáló komponenstípust, ahol a portokat használjuk fel, és a belső változóinkat jelöljük. A modell viselkedését egy TTCN-3 függvényként írjuk le, melyet majd az adott komponensen futtatunk (ennek felépítését lásd később). Az EFSM és TTCN-3 elemek közötti megfeleltetés a 3.1. táblázatban látszik. A meglévő szintaktikán nem kell változtatnunk, mivel nem adunk a nyelvhez új elemet, csak új szemantikát alkalmazunk.

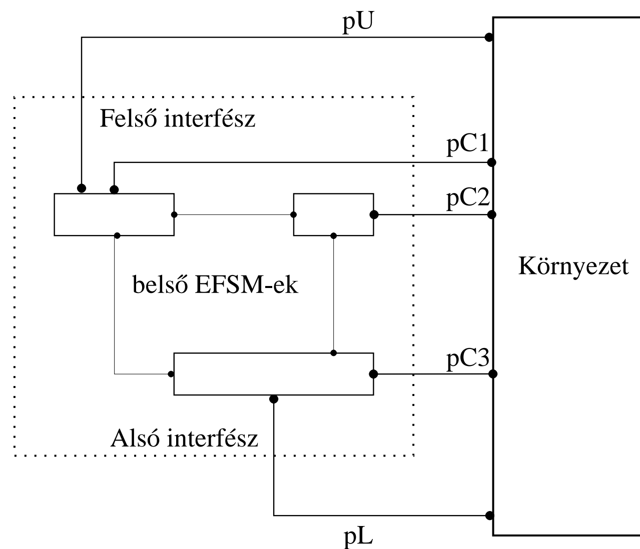
EFSM elem	**TTCN-3 elem**
EFSM	modul, komponens definíció
változók halmaza	adattípus definíció (komponensben)
be- és kimeneti üzenet paraméterei	adattípus definíció (rekordok)
be- és kimenet particionálása	port definíció
küldhető üzenet	paraméterezett sablon

3.1. táblázat. EFSM - TTCN-3 megfeleltetések [6]

Egy valós EFSM modell legyártása sok időt vesz igénybe, de a munka során elkészült portok, sablonok akár változatlan formában átemelhetőek a tesztesetekhez. Tekintve, hogy a modell TTCN-3-ban íródott, ezért ha rendelkezésre áll egy TTCN-3 végrehajtó rendszer, akkor a lehetőség biztosított az elkészült modell szimulálására és tesztelésére megadott szempontok szerint, ezzel ellenőrizve annak helyességét.

Egy EFSM alapú rendszernél 3 jól megkülönböztethető interfészt különítünk el:

felső réteg felé valamint az alsó réteg felé nyújtott interfész, valamint egy kontroll interfész, ahonnan a modellt szükség esetén vezérelhetjük. Egy EFSM-ben az időzítők az input ábécé részei, ezek lejártát input eseményként kezeljük, melyeket a rendszer saját magának küld. Ez lehetőséget teremt arra, hogy ezeket kívülről vezérelhetővé tegyük, hiszen mi is küldhetünk „timeout-üzeneteket” a kontroll interfészen. Ezen a három interfészen keresztül tudunk a modellel kommunikálni, hívjuk ezt a kommunikáló entitást *környezet*nek – ezt szintén egy komponensként valósítjuk meg TTCN-3-ban. Az elrendezést a 3.1. ábra szemlélteti.



3.1. ábra. Fekete doboz és a környezet

Az ábrán a bal oldali téglalap jelöli a modellünket, az általa reprezentált rendszer kívülről nézve egy fekete doboz. Nem tudjuk pontosan, hogy vannak belül összekötve, de ez nem is érdekel minket. A fontos, hogy egy felső (pU) és egy alsó interfészen (pL) gerjesztjük a rendszerünket, illetve az általa gyártott üzeneteket megfigyeljük. A modellhez még rendelkezésünkre állnak vezérlőportok (pCn) is, amiken keresztül a modell működését, annak vezérlését befolyásoljuk.

Egy nagyobb rendszer felbontható több kisebb egységre, és ezek modellezése különkülön EFSM-mel történhet (lásd a fenti ábra). Egy jól specifikált távközlési protokoll belül is moduláris felépítésű: vételi komponens, küldő komponens, kódoló-dekódoló komponens mindig van, és emellett lehet még pl. folyamvezérlési, torlódásvezérlési, linkhibakezelő, illetve menedzsment komponens is. A felső interfész az adott protokollréteg szolgáltatásait veszi igénybe, az alsó pedig az alatta lévő réteg szolgáltatásaira épít.

3.1. Forráskód tagolása

Az előzőekben ismertetett gondolatok alapján az alábbi részekre tagolódik a forráskód: *1.*) komponensek, portok, adattípusok, sablonok definíciós modul, *2.*) EFSM modellek viselkedését leíró modulok, *3.*) környezet definíciója és *4.*) vezérlő lész.

3.1.1. Komponensek, portok, adattípusok, sablonok definíciós modul

Ez az a modul, ahova a tesztmodellünkhöz tartozó EFSM-eket képező komponenstípusokat, az általuk használt port- valamint adattípusokat gyűjtjük.

```
1  type port MyInternalPort message {
2    inout anytype;
3  }
4
5  type component MyModel1 {
6    port MyInternalPort p1;
7    port MyInternalPort p2;
8    port MyInternalPort p3;
9    port MyInternalPort pC;
10   var anytype msg;
11   var integer v;
12 }
13
14 type component MyModel2 {
15   port MyInternalPort p3;
16   port MyInternalPort pC;
17   var anytype msg;
18   var integer w;
19 }
20
21 type component Environment {
22   port MyInternalPort pU;
23   port MyInternalPort pL;
24   port MyInternalPort pC;
25 }
```

A metamodellünk általánossága miatt a portokon küldhető adattípus minden esetben `anytype` típusú (3. sor). Tesztesetek leírásakor nem szokás `anytype` típust

használni, annak erőforrásigénye miatt. Azonban a modellt validációs és szimulációs célokra a felső és alsó interfészeken keresztül összekötjük a környezettel, így ha ezeken a portokon `anytype` típusú üzenetek küldésére és fogadására készülünk fel, akkor a környezet univerzálisan, minden ennek a metamodellnek megfelelő modellhez kapcsolódni tud. A 8-10. sorban megadjuk a `MyModel1` modellünk két portját, valamint a kontrollportját, mindhárom az 1. sorban definiált `MyInternalPort` típusú belső port, valamint a bejövő üzenetek változóba történő eltárolásához felveszünk egy `anytype` típusú változót, és a modell változóit is itt soroljuk fel (pl. 12. sorban egy egész számot tartalmazó v változó). A példa kedvéért definiálunk még egy komponenst `MyModel2` típussal, amely az előző modellel lesz összekötve a `p3` nevű portokon keresztül.

Itt kerül definiálásra a környezetünk komponense is (24. sor), aminek 3 portja van: a felső és az alsó interfészhez valamint a kontrollport, amire a komponensek kontrollportjait kötjük. Ebbe a modulba kerülnek továbbá az üzenetek küldésénél és fogadásánál felhasznált sablonok:

```
template anytype MyTemplate := {
  MyRecordType := {
    value1 := "magic",
    value2 := ?
  }
}

template anytype MyParTemplate (charstring v1,
   charstring v2) := {
  MyRecordType := {
    value1 := v1,
    value2 := v2
  }
}
```

Ahogy a fenti definíció mutatja, itt is a sablon típusa `anytype`, de ezen belül konkrét típushoz rendelhetjük a sablont, így a típusosságból nem vesztünk.

Vegyük észre, hogy egy szabványos rendszer tesztelése, modellezése esetén nagy lesz annak a valószínűsége, hogy az itt lévő entitásokból már jó pár elérhető más modulból, hiszen azokat már valaki valamikor használta és megírta.

3.1.2. EFSM modellek viselkedését leíró modul

A rendszerben szereplő minden EFSM-hez egy modult hozunk létre, benne a viselkedést leíró kóddal. Ez egy TTCN-3 függvény, melyet adott típusú komponensen (EFSM modellen) tudunk futtatni:

```
function f_model1() runs on MyModel1
{
  label STATE_1 {
    alt {
      [] p1.receive(MyTemplate) -> value msg {
        if (msg.MyRecordType.value2 == "OK") {
          v := v+1
          p1.send(MyParTemplate("a", "b"))
          p2.send(...)
          goto STATE_2
        }
        goto STATE_1
      }
      [] p2.receive(...) -> value msg {
        ...
        goto STATE_1
        ...
      }
    }
  }

  label STATE_2 {
    ...
  }
}
```

Az 1. sorban a függvény definíciója található, amelynél meg kell adni, hogy mely típusú komponensen fog futni, esetünkben ez a modellünket azonosító komponenstípus, a `MyModel1`. A függvény belseje az állapotok számával megegyező számú `label` szekciót tartalmaz. Egy címke egy vezérlő állapotot és az abból kiinduló állapotátmeneteket jelöli ki. Az állapotváltáshoz `goto` parancsot használunk (12. sor), mely az `alt` utasítás miatt `repeat` utasítással helyettesíthető, ha az állapotátmenet, nem jár valódi állapotváltással. Ez kevésbé olvasmányos, kifejezőbb, ha mindenhol `goto`-t használunk. Gondoljuk meg, hogy fontos akkor is kiírni a `goto` parancsot, ha állapotváltás nem szükséges, hiszen ha nem hívnánk meg, akkor a futás átesne a következő

címkeblokkba, ami nem várt működéshez vezetne.

Minden állapoton, azaz a kód szerint címkén belül egyetlen `alt` blokk található (4. sor), amely az adott állapothoz tartozó tranzíciók közül az éppen aktív kiválasztásáért felelős. Mindegyik alternatívához tartozó esemény egy bemeneti szimbólum megérkezése a porton. Ha nem jön egyik várt szimbólum se, akkor az elvártaknak megfelelően a rendszer várakozik. Ha valamelyik alternatíva aktiválódik, akkor a benne lévő kód lefut.

Az itt található részben (6-10. sor) a predikátumoknak megfelelő `if` blokkokat látjuk. A 6. sorban láthatunk példát arra, hogy az `anytype` porton érkező üzenet C-beli `union`-ként viselkedik, és így ha `MyRecordType` típust várunk benne, akkor azt hogy érhetjük el. Ha egyik predikátum sem teljesül, akkor a bejött eseményt eldobjuk és visszalépünk az `alt` utasításhoz a `goto` paranccsal (12. sor).

Ha egy predikátum teljesül, akkor az ahhoz tartozó `if` blokk kerül lefutásra. Itt végrehajtódnak az akciók (7. sor), kiküldésre kerülnek az outputok (8-9. sor), majd az állapotváltás (10. sor). Az outputok küldésénél paraméterezett sablonokat (8. sor) is használhatunk, melynek paraméterei akár belső változók, de akár bemenetként érkezett üzenet (`msg` változó) valamely eleme is.

3.1.3. Környezet definíciója

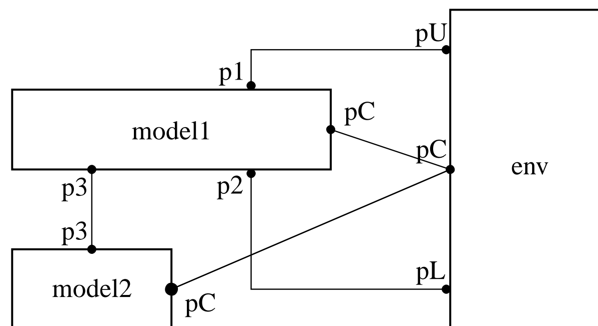
Ez az a modul, ami a környezet viselkedését írja le, hasonlóan az előzőhöz, ez is egy függvényt tartalmaz. Ezt a lehető legáltalánosabban kell megírnunk, hogy ne kelljen minden egyes rendszernél külön környezetet írni. A célunk itt, hogy a rendszertől érkező üzenetek, események hatására jelzést adjunk egy külső/környezeti komponensnek (legyen az valamilyen egyéb szoftver, vagy akár hardver), valamint az jelezni tudjon a modellnek: például az időzítők vezérlése és a terminálódás kezelése a kontrollportokon keresztül megvalósítandó üzenetek küldésével. Ez a függvény tartalmazhat továbbá a modell validációjára, szimulációjára is vonatkozó logikát.

3.1.4. Vezérlő rész

A modelleket jelentő komponenseket egymással és a környezettel a tesztesetek leírójában kötjük össze, majd azokon az előzőkben leírt viselkedést tartalmazó függvényeket indítjuk el. Nem használunk SUT-ot (nincs tesztelendő rendszer), csak a komponenseket és környezetet, mint párhuzamos tesztkomponenseket (PTC) használjuk. Ez a megfontolás a tesztvégrehajtó rendszerben való futathatóság miatt nagyon fontos.

```
testcase modelConfiguration() runs on MTCType {
  var MyModel1 model1 := MyModel1.create;
  var MyModel2 model2 := MyModel2.create;
  var Environment env := Environment.create;

  connect(model1:p1, env:pU);
  connect(model1:p2, env:pL);
  connect(model1:p3, model2:p3);

  connect(model1:pC, env:pC);
  connect(model2:pC, env:pC);

  model1.start( f_model1() );
  model2.start( f_model2() );
  env.start( f_env() );

  all component.done;
}
```

Az 1. sor a „teszteset” definíciója, meg kell adnunk az MTC típusát, aminek a felépítése nem fontos, mivel nem használjuk semmire, hiszen nem fut igazi értelemben vett teszteset, ugyanaz a szintaxis, de itt a modellkonfiguráció megadására használjuk. A 2-4. sorban példányosítjuk a komponenseket és a környezetet, majd `connect` parancsokkal a megfelelő portokat összekötjük (a környezet kontrollportjára minden komponens kontrollportját rákötjük – TTCN-3-ban ez megengedett[3] és így nem kell folyamatosan a környezetünk komponenstípusát a különböző modellekhez igazítani). Az összekötést a 3.2. ábra demonstrálja.



3.2. ábra. Komponens példányok és környezet összekötése

A 9-10. sorban elindítjuk a példányokon a modellek viselkedését leíró függvényeket. A teszteset végén pedig megvárjuk, hogy minden komponens befejezze a futását (12. sor).

4. fejezet

A LAPB protokoll tesztmodellje TTCN-3-ban

Ennek a fejezetnek a témája egy valós protokoll, a LAPB (Link Access Procedure, Balanced) egyszerűsített – néhány választott változó mentén létrehozott – EFSM modelljének megalkotása, valamint a modell alapján TTCN-3 kód generálása. A fejezet vége bemutatja, hogy az elkészült modellen egyszerűbb keresési algoritmusok, hogyan valósíthatóak meg, ezzel utat nyitva a jövőben a tesztmodellen összetettebb algoritmusok teszteléséhez.

A LAPB protokoll az X.25-ös protokoll készlet adatkapcsolati rétegét valósítja meg [8], HDLC (High-Level Data Link Control) keretszervezéssel. A LAPB-kereteknek 3 típusa van: Information (I-Frame), Supervisory (S-Frame) és Unnumbered (U-Frame). Az I-keretek szállítják a hálózat rétegbeli információt, az S-keretek adás kérésére és annak felfüggesztésére, státuszjelzésre valamint I-keretek visszaigazolására szolgálnak), míg az utolsó a kapcsolat beállításánál, bontásánál és hibajelzésnél használatos. A protokollról bővebb leírása [9]-ben olvasható.

Régi protokollról van szó, de demonstrációs célokra tökéletes alanyként szolgál. Általában a modelleket meglévő szöveges specifikációkból készítik el, én a „reverse engineering”-et választottam, azaz egy elkészült implementációból, forráskódból készítettem EFSM modellt. A motiváció ehhez az, hogy bármilyen forráskódból felírható egy EFSM, és mivel egy protokoll implementációjáról van szó, ezért maga a kód is egy állapotgép köré van strukturálva, ami jó iránymutatásként szolgál.

A forráskódot a Linux kernelben megtalálható implementációból veszi, ezek megtalálhatóak a legtöbb disztribúcióban, de elérhető online is számos helyen, például a kaliforniai Berkeley egyetem egyik szerverén [13, 12].

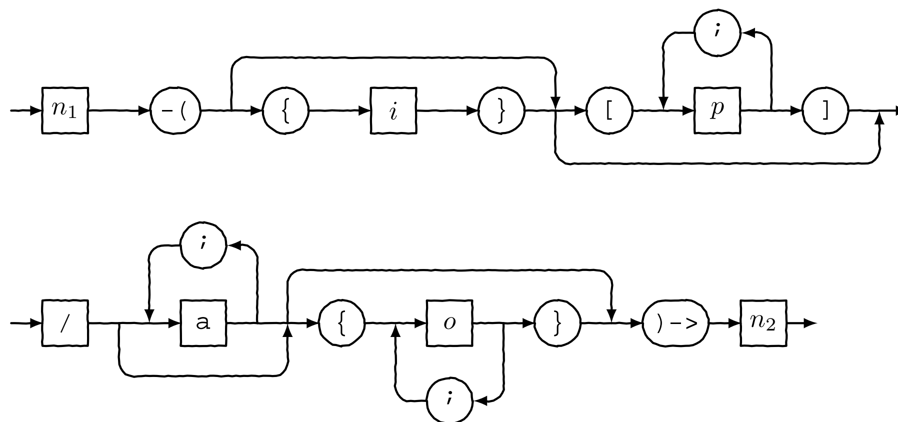
4.1. Állapotgráf-leíró formátum

Az implementáció alapján papíron felrajzolható az EFSM állapotgráfja, majd ebből az előző fejezetben leírtakat követve a TTCN-3 kód előállítható, de a további munka elősegítésére egy Java alkalmazás készült, ami egy jól strukturált szöveges fájlból felépít egy állapotgráfot, majd ebből TTCN-3 kódot generál a metamodellnek megfelelően. Ez azért is célravezető, mert a különböző algoritmusok vizsgálatához és futtatásához is szükség van az állapotgráfos leíráshoz. A következőkben a gráfot leíró fájl struktúrája van bemutatva.

A formátum csomópontok valamint élek és azok attribútumainak definiálását támogatja. Egy él a következő formában adható meg:

```
NODE1 -({I1} [P1] / a:=0; b:=0 {O1} )-> NODE2
```

A fenti a `NODE1` és `NODE2` csomópont közötti él, amely `I1` input és `P1` predikátum teljesülése esetén kerül érvényesítésre, az a és b változókat 0-ba állítja, valamint `O1` kimenetet ad ki. A fenti általános alak esetén nem szükséges külön predikátumcsomópont felvétele, így a gráf egy kicsit egyszerűbb alakú, mint a 2. fejezetben ismertetett gráfmodell. Ahol viszont egy bemeneti szimbólumhoz tartozóan több különböző predikátummal ellátott átmenet is rendelkezésre áll, ott a könnyebb áttekinthetőség érdekében fontos a köztes predikátumcsomópontok használata. A fájl egy sorának szintaxisát a 4.1. ábra szemlélteti.



4.1. ábra. Szintaxis gráf

Mivel a fenti formátum egyértelműen leírja a gráfban szereplő csomópontokat, ezért azok explicit definiálására nincs szükség. Amint a feldolgozó meglát egy még nem ismert karakterláncot csomópontként, létrehozza a csomópontot és felveszi a gráfba. Mivel a programhoz tartozik egy grafikus megjelenítő is (lásd később), ezért a csomópontok pozíciójának és típusának (állapotcsomópont vagy predikátumcsomópont)

megadása is támogatott a leíróban. Ezen soroknak a formátuma:

```
STATE1{{13,37}}
P1{800,85}
```

A fenti részlet két csomópont elhelyezését adja meg, az első egy állapotcsomópont (ezt dupla kapcsos zárójellel jelöljük), a második pedig predikátumcsomópont. A koordináták pozitív egész számok, a rajzolóvászon igény szerint nagyítható, kicsinyíthető.

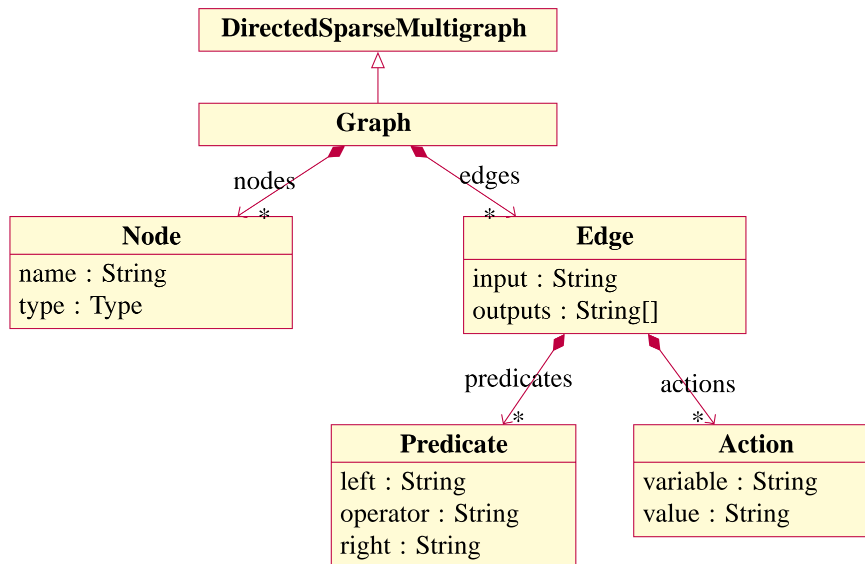
4.2. Gráf reprezentáció

Mielőtt rátérnék arra, hogyan tudjuk a fenti fájlt beolvasni és ebből egy gráfot készíteni, nézzük meg, hogy tudunk egy gráfot Java nyelvben hatékonyan és egyszerűen reprezentálni.

Szükségem volt a gráf vizualizációjára is, ezért a munkámhoz a JUNG[16] keretrendszert használtam fel. A rendszer legnagyobb előnye (amellett, hogy ábrázolhatjuk vele az elkészült gráfunkat), hogy a csomópontokat és az éleket jelentő osztályokat nem adja a kezünk alá, nem kell azokhoz igazodunk, azokból örökléssel saját osztályokat gyártani, hanem felhasználva a Java generikus típusait teljes szabadságot kapunk ehhez.

A dokumentációt végigolvasva a `DirectedSparseMultigraph` [15] osztályt választottam ősosztályul a gráfhoz, hiszen irányított élekre van szükségünk, valamint arra, hogy definiálhassunk párhuzamos éleket is. Mivel az élekkel és a csomópontokkal kapcsolatos relációkat (pl. két csomópont közötti élek definiálása vagy a csomópontokból kiinduló és oda érkező élek lekérdezése) lekezeli helyettünk a keretrendszer, így nekünk csak a modellel kapcsolatos egyéb tulajdonságokkal kell kibővítenünk a gráfmodellünket (bemenet, kimenet, akciók, predikátumok). Így a modell a 4.2. ábrán látható formát ölti.

Az osztálydiagram alapján látható, hogy a főbb komponensek, hogy kapcsolódnak egymáshoz. A `Graph` objektum tartalmazza az éleket (`Edge`) és a csomópontokat (`Node`). Az éleken szerepelhet bemeneti szimbólum, illetve több kimeneti szimbólum. Az adott átmenetekhez tartozó predikátumok és a végrehajtandó akciók az átmeneteknek megfelelő éleken eltároljuk. Az akciók (`Action`) esetén az értékadás bal és jobb oldalán lévő kifejezést tároljuk (értelmezés nélkül), a predikátumok (`Predicate`) esetén pedig a logikai kifejezés két oldalát és a köztük lévő logikai operátort (ha a kifejezés egy igaz/hamis érték, akkor belül ezt egy összehasonlítással reprezentáljuk). Az általam választott ilyen belső gráfmodell nem képes az operandusok és azokon értelmezett műveletek feldolgozására, kiértékelésére. A programomat használó algoritmusok és egyéb eszközök feladata lesz ezek értelmezése, az ezzel való munka.



4.2. ábra. A model osztálydiagramja

Nem közlöm a fájlbeolvasó algoritmus pszeudokódját, mert maga az implementáció nem rejt magában érdekességeket, de az elvet leírom. A fájlon sorról sorra haladva, reguláris kifejezések illesztésével a fentiekben ismertetett sor-formátumot értelmezzük és tokenizáljuk, így az objektumok felépítése nem jelent gondot. Az értelmezés során csak arra kell figyelnünk, hogy ha egy adott csomóponthoz már gyártottunk `Node` objektumot, akkor ha megint vele találkozunk, akkor a rá mutató referenciát használjuk, ne gyártsunk újabb példányt. Természetesen a pozíciókódok kiolvasása is hasonlóképpen történik, és itt tudjuk meg azt is, hogy mi lesz egy csomópont típusa (állapotcsomópont vagy predikátumcsomópont).

4.3. LAPB protokoll gráfleírójának elkészítése

A már említett LAPB C nyelvű implementációjából felépítettem a leírt formátumban az állapotgráfot. A `lapb_in.c` fájlból indultam ki, ahol lényegében az állapotgépet definiálták, azaz, hogy adott input hatására, mely állapotban mely feltételek mellett, minek kell történnie. A küldéssel kapcsolatos függvényeket a `lapb_out.c`-ben, a segédfüggvények definícióját a `lapb_subr.c`-ben, a felső interfészt jelentő függvényeket pedig a `lapb_iface.c`-ben találjuk. A protokollhoz kapcsolódó időzítőket a `lapb_timer.c`-ben leírt függvényekkel kezelhetjük.

A modell felépítéséhez így a gráfleíró elkészítéséhez is szükség volt bizonyos implementációs részek kitakarására, a következőkben ezekhez kapcsolódó tervezői döntéseket ismertetem.

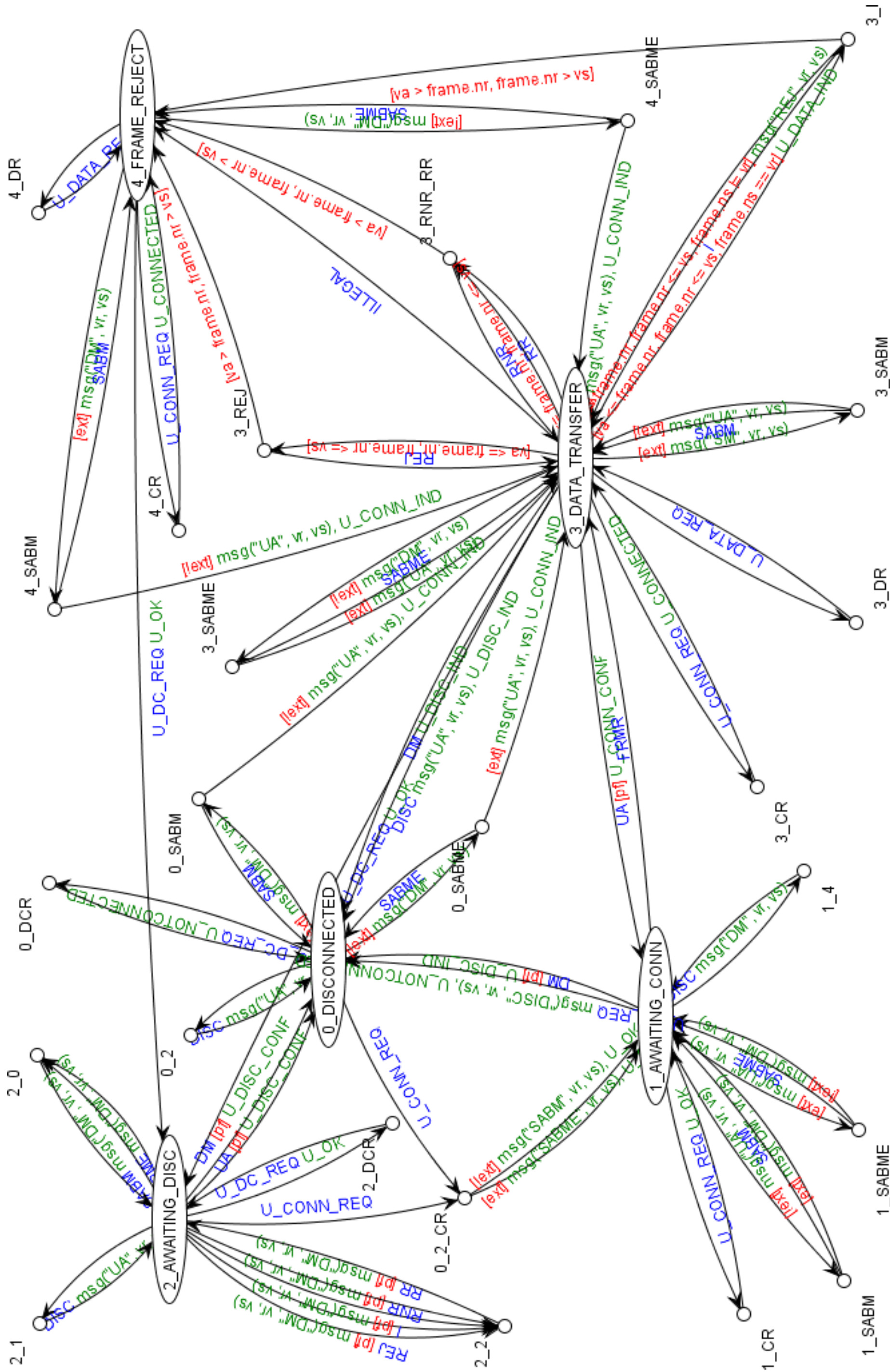
Egy LAPB kapcsolathoz rengeteg információ (lapb_cb nevű struktúra) tartozik, ilyen például az, hogy melyik hálózati eszköz tartja fent a kapcsolatot vagy az időzítők állapotai. Ezek közül csak a jelenlegi állapotot tároló változót (ennek értékkészlete lett a kijelölt állapottér), a kapcsolat módját jelző flaget (normál – 8 hosszú, vagy kiterjesztett – 128 vagy 32768 hosszú csúszóablakot használ), valamint az utoljára visszaigazolt, küldött és fogadott keret sorszámát tartalmazó változót hagytam meg, a többivel nem foglalkoztam, hiszen az így kiválasztott változók is kellő komplexitású modellt adnak ki, amely demonstrálja a metamodell leíróképességét. A rendszerben lévő időzítők is bekerültek a modellbe és az input ábécé részét képezik.

A LAPB keret mezői közül a két legfontosabb a vezérlőmező és az adatmező, amely a hasznos terhet hordozza. A protokoll számára a payload transzparens, ezért a modellünk is csak a vezérlőmezőre és annak elemeire szorítkozik. Az implementációban ez a lapb_frame struktúra elemeit jelenti, melyből a keret típusát, az $N(R)$ és $N(S)$ sorszámokat, valamint a P/F bitet (ami azt jelzi, hogy a küldő vár-e azonnali választ a fogadótól) használjuk. Ezeket azért választottam, hogy a modellben legyen példa bemenetként érkező értékek predikátumban, valamint akcióban történő felhasználására.

Egy apró részlet példaként a leíróból:

```
1 0_DISCONNECTED -({portP.DISC} / {portP.msg("UA", vr, vs)}
      )-> 0_2
2
3 ...
4
5 3_DATA_TRANSFER -({portP.RR} /)-> 3_RNR_RR
6 3_RNR_RR -([va <= frame.LAPBFrameType.nr; frame.
     LAPBFrameType.nr <= vs] / va:=frame.LAPBFrameType.nr)
     -> 3_DATA_TRANSFER
7 3_RNR_RR -([va > frame.LAPBFrameType.nr] /)-> 4
     _FRAME_REJECT
8 3_RNR_RR -([frame.LAPBFrameType.nr > vs] /)-> 4
     _FRAME_REJECT
```

Az 1. sorban arra láthatunk példát, hogy ha egy DISC bemeneti szimbólum érkezik, akkor egy konstansokkal és a modell belső változóival (v_r, v_s) paraméterezett kimeneti szimbólumot küldünk. Az 5. sorban egy állapotcsomópont és predikátumcsomópont közötti él van, mely az RR bemeneti szimbólum hatására aktiválódik. A 2-4. sorban a 3_RNR_RR nevű predikátumcsomóponthoz kapcsolódó élek szerepelnek. Ezeken feltüntetjük a predikátumokat, valamint az akciókat (2. sor). Jól látható arra is példa, hogy a predikátumban és az akcióban a bemenetről érkező keret (frame) változóit használ-



4.3. ábra. LAPB protokollhoz tartozó EFSM gráf vizualizációja

juk argumentumokként.

Szem előtt tartva, hogy ebből TTCN-3 kód generálódik, a bemeneteknél és a kimeneteknél – mivel a rendszerünk pusztán szövegként értelmezi ezeket – TTCN-3 sablonokat alkalmazunk (definícióikat lásd később). Az általános leíróképesség megtartásához a portokat is fel kell tüntetnünk, hiszen pusztán a sablonok alapján nem tudnánk, hogy azokat melyik porton kell várni, vagy éppen küldeni (megjegyzés: pusztán típus alapján történő illesztés sem jelent megoldást, hiszen elképzelhető egy olyan eset is, amikor egy komponens több különböző portján is küldhető ugyanolyan típusú üzenet). A `frame` változó `anytype` típusú, ezért a benne lévő tényleges adatot a C-beli `union` használatánál megszokott módon érhetjük el.

A teljes leíró alapján létrehozott vizualizáció megtekinthető a 4.3. ábrán a 28. oldalon. Az ábrán kék színnel jelöltem a bemeneti szimbólumokat, zölddel a kimeneteket és pirossal a predikátumokat. Nem olvasható minden élen tisztán a felirat (ráadásul az akciók még nem is szerepelnek), de szerencsére a program támogatja a tooltipek megjelenítését, így ha az alkalmazásban rámutatunk egy élre, akkor egy kis felugró ablakban megjelenik róla információ. Mivel a keretrendszer nem támogatja a hurokélek szép elhelyezését, így ahol erre volt szükség ott egy külön csomópont felvételével vált pozicionálhatóvá a „hurokél" (ezt csak a vizualizációnál használjuk, a generált TTCN-3 kódban már nem látszódik).

4.4. TTCN-3 kód generálása

A gráf alapján az előző fejezetben ismertetett modell viselkedését leíró függvények belseje a metamodellnél ismertetett szabályrendszerek mentén generálható. Viszont a komponens- és porttípusokat, valamint az üzenet sablonokat az implementáció alapján a modellünkhöz kézzel kell megírni.

A LAPB keretet leíró `record` az alábbi formát ölti:

```
1  type record LAPBFrameType {
2    charstring frameType,
3    integer nr,
4    integer ns,
5    boolean pf
6  }
```

Felépítését tekintve nagyon hasonlít az implementációban található `lapb_frame` struktúrára. A 2. sorban lévő `frameType` jelöli a keret típusát karakterláncként, az ezt követő 2 szám rendre a utoljára fogadott és küldött üzenet sorszámát, az utolsó pedig a már említett `P/F` bitet.

A porttípusok és a modellünket jelentő komponenstípus definíciója:

```
1  type port Control_PT message
2  {
3    inout anytype;
4  }
5
6  type port User_PT message
7  {
8    inout anytype;
9  }
10
11 type port Peer_PT message
12 {
13   inout anytype;
14 }
15
16 type component Model
17 {
18   port Control_PT pC;
19   port User_PT portU;
20   port Peer_PT portP;
21
22   var boolean ext;
23   var integer va;
24   var integer vs;
25   var integer vr;
26
27   var anytype frame;
28   var anytype msg;
29 }
```

Az 1-14. sorban definiáljuk rendre a kontroll, a felső és az alsó interfészt jelentő porttípusokat. A 16-28. sorban írjuk le a modelltípusunk definícióját: 18-20. sorban jelöljük a komponensünk portjait névvel, majd 22-25. sorban a modell belső változóit írjuk le: kiterjesztett bit, és a három sorszám (utoljára visszaigazolt, küldött és fogadott). A 27-28. sorban az alsó és felső interfészeken érkező üzeneteket tárolhatjuk el a feldolgozás idejére. Látható, hogy a metamodellt követve a portok `anytype` típusú üzenetek küldésére és fogadására alkalmasak.

Nem mutatom be az összes sablonhoz tartozó definíciót, mert csak a keret típusában

térnek el egymástól, csak egyet a példa kedvéért, továbbá leírom a küldéshez használt paraméteres sablont is:

```
template anytype I := {
  LAPBFrameType := {
    frameType := "I",
    nr := ?,
    ns := ?,
    pf := ?
  }
}

template anytype msg (charstring type_p,
             integer nr_p, integer ns_p) := {
  LAPBFrameType := {
    frameType := type_p,
    nr := nr_p,
    ns := ns_p,
    pf := 0
  }
}
```

Az 1-8. sorban definiált `anytype` típusú sablon egy `anytype` típusba ágyazott `LAPBFrameType` típusú üzenetre illeszkedik, mely I keretet tartalmaz (amelyben a LAPB adatkapcsolati protokoll hálózati rétegbeli információt szállít – ennek tartalma a protokoll szempontjából transzparens, ezért nem is szerepel a modellben). A 10-18. sorban pedig küldésnél használt paraméteres sablont ír le, amellyel megadható a küldendő keret típusa és a két sorszám ($N(R)$ és $N(S)$), a `P/F` bit pedig konstans 0 (ami a `LAPB_POLLOFF`[12] konstansnak felel az implementációban).

4.5. Bejáró, kereső algoritmusok a gráfon

Az elkészült gráfhoz különböző bejáró és kereső algoritmusokat írtam, melyeket TTCN-3-ban a *környezet* komponensben valósítottam meg.

4.5.1. Megjelölt éleket tartalmazó séta

Először egy olyan séta megadását kerestem, amely megadott éleken sorrendben átmegy. Az algoritmus nagy mértékben épít a szélességi bejárással történő útkeresésre

adott csomópontból adott csomópontba, ezért először ennek a pszeudokódját írom le:

bemenet: v_1 a kezdőcsomópont és v_2 a végcsomópont
kimenet : egy éllista, ami v_1-ből v_2-be vezető utat ad meg

```
S ← üres sor
v1 hozzáadása S végére
Látva(v1) ← TRUE
while S nem üres do
    v ← S első eleme
    if v = v2 then
        „apa” mutatók mentén az út visszakeresése
        visszatérés az úttal
    end
    for n ∈ v szomszédjai do
        if ¬Látva(n) then
            n hozzáadása S végére
            „apa” mutató mentése (v => n)
        end
    end
end
```

A sétakereső algoritmus pszeudokódja:

bemenet: v_1 a kezdőcsomópont és v_2 a végcsomópont, E megjelölt élek
kimenet : R egy éllista, ami v_1-ből v_2-be vezető sétát ad meg, az E-beli élek sorrendben történő érintésével

```
R ← üres lista
n1 ← v1
for e ∈ E do
    n2 ← e kezdőcsomópontja
    U ← Útkeresés(n1, n2)
    U hozzáadása R-hez
    e hozzáadása R-hez
    n1 ← e végcsomópontja
end
U ← Útkeresés(n1, v2)
U hozzáadása R-hez
```

A végén a sétánk az R listában lesz. Az útkeresés pedig a már fent említett szélességi bejárással implementálható.

4.5.2. Hurkok keresése

Változónként egyesével nézve az éleket 4-féle osztályba sorolhatjuk őket: 1. a változó értékadás bal oldalán, 2. az értékadás jobb oldalán, 3. predikátumban, vagy 4. bemenetként szerepel.

Felhasználva a fenti osztályozást olyan hurokkereső algoritmust implementáltam, amely adott vezérlőállapotból keres olyan hurkokat, amelyen egy megadott változót állítani tudunk (azaz az előbbiekben ismertetett 1-es osztályba sorolható).

Mivel a hurkok keresése során két csomópont közötti utat gyakran kell meghatároznunk, ezért az útkereséshez előre meghatározott feszítőfákat használunk. Ezzel két csomópont közötti útkeresés konstans időben megy. A hurokkereső algoritmus:

bemenet: E élek halmaza, ahol az adott változót állítani tudjuk, v a vezérlőállapot
kimenet : H egy olyan éllisták listája, amelyben egy éllista olyan hurkot ír le, amely v vezérlőállapotot érinti és az egyik éle E eleme

$H \leftarrow$ üres lista
for $e \in E$ **do**
  $n_1 \leftarrow e$ kezdőcsomópontja
  $R \leftarrow$ `ÚtkeresésFeszítőfával(`v, n_1`)`
  e hozzáadása R-hez
  $n_2 \leftarrow e$ végcsomópontja
  $U \leftarrow$ `ÚtkeresésFeszítőfával(`n_2, v`)`
  U hozzáfűzése R-hez
  R hozzáadása H-hoz
end

Az algoritmus végén, adott vezérlőállapot és változó mellett, a H lista fogja tartalmazni a keresett hurkokat.

5. fejezet

Összefoglalás

A dolgozat célja a távközlésben használt több különböző eltérő tesztmodell-megadási módszer helyett egy egységes, TTCN-3 alapú tesztmodellezésre használt metamodell megadása. Erre a konszolidációra azért van szükség, mert az eltérő megvalósítási korlátok miatt a cégek rengeteg időt és pénzt költenek különböző eszközökre. A költséghatékonyság mellett, a közös nyelv használata révén a tesztmodellek és a tesztesetek homogenitása is elérhető, amely a tesztmodellek és a belőlük generált tesztesetek kohézióját nagy mértékben növeli, a korábban definiált adattípusok újrafelhasználhatóságával.

A dolgozatban leírt módszer tesztmodellek alkotására nem csak azért hasznos, mert a tesztesetek nyelve – a TTCN-3 – használható a tesztmodell leírására is, és így a tesztmérnököknek nem kell kilépniük a komfortzónájukból, hanem mert felépítését tekintve egyszerű, szöveges formátumának köszönhetően bárki számára könnyen olvasható és kellően moduláris, ezért adoptálása rövid időn belül végrehajtható.

A megadott eljárás alkalmazása valós protokollon keresztül kerül demonstrálásra, valamint bemutatásra, hogy a kialakítása során fontos szempont volt, hogy a tesztmodell a környezetből programozható legyen. A dolgozat példákkal alátámasztva mutatja be, hogy algoritmusok is könnyen illeszthetőek az így megadott tesztmodellekhez.

A TTCN-3 futtató és végrehajtó környezetei a jelen pillanatban nem állnak készen a lehetőségek teljes mértékben történő kiaknázására (még nem megoldott a *környezet* komponens kényelmes, egyszerű implementálása és egy külső eszköz közti zökkenőmentes kommunikáció), jövőbeni tervek között szerepel ezen a téren a meglévő eszközökhöz beépülő, támogató modulok készítése, melyek áthidalják ezeket a problémákat.

Köszönetnyilvánítás

Végezetül szeretnék köszönetet mondani mestereimnek, tanáraimnak és mentoraimnak, akik folyamatosan támogatták felkészülésem és munkám, dolgozatom írása során szakmai tanácsaikkal, észrevételeikkel megfelelő útmutatást adtak. Hálás vagyok különösen Erős Levente bátorító szavaiért, valamint Dr. Kovács Gábor lankadatlan türelméért, alapos odafigyeléséért. Dr. Adamis Gusztávnak is szeretném megköszönni, hogy segített a TTCN-3 nyelvvel kapcsolatos kérdések tisztázásában. Mindhárman nagy mértékben hozzásegítettek ahhoz, hogy a kitűzött célokat sikerüljön elérnem.

Irodalomjegyzék

[1] Chourouk Bourhfir, Rachida Dssouli, and El Mostapha Aboulhamid. *Automatic Test Generation for EFSM-based Systems*. Tech. rep. Université de Montréal, Dept. d'Informatique et de Recherche Operationnelle, 1996.

[2] Rita Dorofeeva, Khaled El-Fakih, Stephane Maag, Ana R. Cavalli, and Nina Yevtushenko. "FSM-based conformance testing methods: A survey annotated with experimental evaluation". In: *Information and Software Technology* (2010).

[3] Levente Erős. *Mérési segédlet a TTCN-3 méréshez*. 2010. URL: `http://qosip.tmit.bme.hu/twiki/pub/Meres/TTCN/TTCN-3.pdf`.

[4] ETSI. *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*. 2012.

[5] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. "An introduction to the testing and test control notation (TTCN-3)". In: *Computer Networks* 42.3 (2003), pp. 375–403. ISSN: 1389-1286. DOI: `10.1016/S1389-1286(03)00249-4`. URL: `http://www.sciencedirect.com/science/article/pii/S1389128603002494`.

[6] Antal Wu-Hen-Chang, Gusztáv Adamis, Levente Erős, Gábor Kovács, and Tibor Csöndes. "A New Approach in Model-Based Testing: Designing Test Models in TTCN-3". In: *SDL Forum*. Ed. by Iulian Ober and Ileana Ober. Vol. 7083. Lecture Notes in Computer Science. Springer, 2011, pp. 90–105. ISBN: 978-3-642-25263-1.

[7] Conformiq Inc. *Confirmiq Qtronic*. 2012. URL: `http://www.conformiq.com/`.

[8] InetDaemon. *Link Access Procedure Balanced (LAPB)*. July 2012. URL: `http://www.inetdaemon.com/tutorials/telecom/x25/lapb.shtml`.

[9] ITU-T. *ITU-T Recommendation X.25: Interface between Data Terminal Equipment (DTE) and Data Circuit-terminating Equipment (DCE) for terminals operating in the packet mode and connected to public data networks by dedicated circuit*. Tech. rep.

[10] David Lee and Mihalis Yannakakis. "Principles and Methods of Testing Finite State Machines - A Survey". In: *PROCEEDINGS OF THE IEEE* (1996).

[11] Elvior LLC. *MOTES*. 2012. URL: http://www.elvior.com/motes/generator/.

[12] Jonathan Naylor and Henner Eisen. *LAPB header fájl*. 2000. URL: https://casper.berkeley.edu/svn/trunk/roach/sw/linux/include/net/lapb.h.

[13] Jonathan Naylor and Henner Eisen. *LAPB implementáció*. 2000. URL: https://casper.berkeley.edu/svn/trunk/roach/sw/linux/net/lapb/.

[14] Inc. Neda Communications. *Neda's Implementation of ESRO - Source*. 1999. URL: http://www.mailmeanywhere.org/sw.free/neda/leap/esro/ESROS-MulPub/doc/esroSrcDesc-tty/split/node36.html.

[15] JUNG Framework Development Team. *DirectedSparseMultigraph (jung2 2.0 API)*. Jan. 2010. URL: http://jung.sourceforge.net/doc/api/edu/uci/ics/jung/graph/DirectedSparseMultigraph.html.

[16] JUNG Framework Development Team. *JUNG - Java Universal Network/Graph Framework*. Jan. 2010. URL: http://jung.sourceforge.net/.

[17] Mark Utting and Bruno Legeard. *Practical Model-Based Testing*. Morgan Kaufmann Publishers, 2007. ISBN: 978-0-12-372501-1.

[18] Colin Willcock, Thomas Deiß, Stephan Tobies, Stefan Keil, Federico Engler, and Stephan Schulz. *An Introduction to TTCN-3*. John Wiley & Sons, 2005. ISBN: 978-0-470-01224-6.