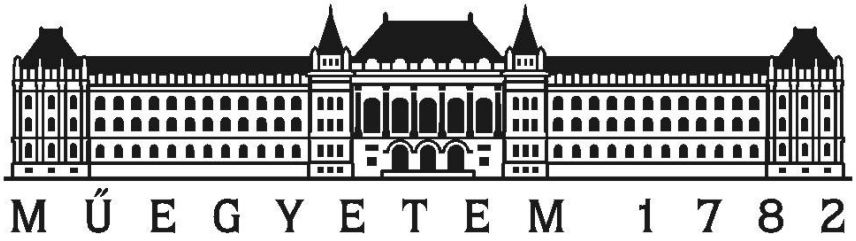


Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatikus kódgenerálás helyességének ellenőrzése

Készítette:

Jeszenszky Balázs, V. Inf., jeszyb@gmail.com

Konzulens:

Dr. Majzik István, Méréstechnika és Információs Rendszerek Tanszék,

majzik@mit.bme.hu

Tudományos Diákköri Konferencia

2012. november 14.

Tartalomjegyzék

1 Bevezetés 4
2 Motiváció 6
2.1 Beágyazott rendszerek modellalapú fejlesztése 6
2.2 Az elkészült kódgenerátor 7
2.2.1 A kiindulási modell 7
2.2.2 Kódgenerálási módszer 9
2.2.3 Platformszolgáltatások és kiterjesztések integrálása 10
2.3 Verifikáció kritikus rendszerekben 12
3 Az ellenőrzési koncepció 14
3.1 Meglévő megoldások 14
3.2 A lehetőségek áttekintése 15
3.3 Az ellenőrzés módszere 16
3.4 Szükséges technológiák 16
3.4.1 Automatikus teszteset-generálás 16
3.4.2 Tesztek futtatása 19
3.4.3 Teszt eredmények kiértékelése 19
3.5 Jellegzetességek 20
3.5.1 A módszer korlátai 20
3.5.2 Komponensek tesztelése elosztott rendszerekben 20
4 Keretrendszer a helyességellenőrzéshez 23
4.1 Áttekintés 23
4.2 Az alkalmazott algoritmusok 24
4.2.1 A modell felműszerezése 24
4.2.2 Temporális logikai kifejezéskészlet előállítása 25
4.2.3 Absztrakt tesztesetek leképezése 31
4.2.4 Teszt végrehajtás 34
4.2.5 Tesztek értékelése 34
4.3 Automatikus eszközök 37
5 Megvalósítás és mintapéldák 39
5.1 Az eszközök ismertetése 39
5.2 Mintapéldák 39
5.2.1 Demonstráció szimulátorral 39

5.2.2 Mitmót konfiguráció ismertetése 42
6 Összefoglalás és értékelés 45
6.1 Elért eredmények 45
6.2 Problémák 45
6.3 Továbbfejlesztési lehetőségek 45
Hivatkozások 47

1 Bevezetés

Beágyazott rendszerek fejlesztése során egyre elterjedtebbé válik a formális modelleken alapuló módszerek használata. A modelleken végrehajtott formális verifikáció segítségével sok tervezői döntés, algoritmus helyessége igazolható, így az implementáció már ellenőrzött tervek alapján indítható. Korábbi munkánk eredményeként elkészült egy olyan automatikus kódgenerátor, ami adott platformok szolgáltatásaihoz illeszkedve, könnyen paraméterezhetően képes közvetlenül fordítható C forráskód modell alapú szintézisére. Kritikus alkalmazások esetén a generált kód felhasználásához azonban be kell látni, hogy maga a kód, valamint a kód és a futtató platform együttműködése is pontosan megfelel a modell által meghatározott viselkedésnek.

Dolgozatomban bemutatok egy olyan módszert, amivel lehetővé válik a tervezés során használt modell (időzített automaták hálózata) és a modell alapján generált alkalmazás viselkedésének összevetése, és meghatározott eltérések kimutatása. Mivel az alkalmazás komplex platformszolgáltatásokat (pl. függvénykönyvtárakat, firmware komponenseket) használhat, ezért az ellenőrzés dinamikusan történik, a modell alapján generált tesztek automatikus végrehajtásával és a teszt eredmények ellenőrzésével. A módszer és az ezt támogató keretrendszer főbb elemei a következők:

- A modell által meghatározott viselkedés ellenőrzéséhez szükséges tesztkészlet generálása automatikusan történik, kihasználva a tervezői környezet modellellenőrző komponensét. Egy olyan temporális logikai kifejezéskészletet állítok elő, aminek ellenőrzéséhez a modellellenőrzőnek el kell végeznie a modell szisztematikus bejárását (adott fedettségi kritériumok szerint), eközben absztrakt tesztesetként rögzítve a bemeneteket és az elvárt kimeneteket.
- A keretrendszer szintén automatikusan végzi az absztrakt tesztesetek leképezését olyan konkrét tesztesetekké, amelyek a vizsgált alkalmazáson (a platform interfészeihez illesztve) végrehajthatók. A megvalósítás során ugyanakkor külön ügyeltem arra, hogy fenntartsam a teszt végrehajtó komponens platformfüggetlenségét.
- A teszt végrehajtás során rögzített viselkedés (napló) és a tesztesetek által elvárt viselkedés összevetése precízen definiált reláció alapján történik. Ennek módosításával meghatározható a megengedett eltérések köre (pl. extra kimenetek elfogadása). Amennyiben a tényleges lefutás eltér a reláció szerint elvárttól, az eltérés tényén túl számos diagnosztikai kiegészítés is biztosítható.

A tesztelési folyamat során a kódgenerátort „fekete dobozként” kezeljük, tehát a megfelelő kapcsolódási pontok betartása mellett a keretrendszer más kódgenerátorral, vagy akár kézzel kódolt alkalmazás viselkedésének ellenőrzésére is alkalmazható, így a keretrendszer általánosan is támogatni tudja egy modell és egy konkrét alkalmazás viselkedése közötti eltérések tesztelését. Ehhez kiemelhető, hogy mind a modell fedettségi kritériumok, mind a teszt kiértékelési relációk módosíthatók.

A 2. fejezetben részletesen írok a motivációkról, vázlatosan ismertetem a kódgenerátor működését, és a kritikus rendszerekben használatos fejlesztési megkötéseket. A 3. fejezetben bemutatom a kidolgozott koncepciót, majd a 4. fejezetben részletesen írok

a megvalósított keretrendszerről. Az 5. fejezetben példákon keresztül demonstrálom a rendszer működését. Végül a 6. fejezetben összefoglalom és értékelem az elért eredményeket, kitérek a bővítési lehetőségekre.

2 Motiváció

Dolgozatom elsődleges motivációja a modellvezérelt szoftverfejlesztés támogatása. Különös hangsúlyt fektettem beágyazott rendszerek fejlesztésének támogatására.

A modellalapú fejlesztést mutatom be a 2.1. részben. A kiinduló módszer egy korábbi TDK munka [1], erről szól a 2.2. fejezet.

2.1 Beágyazott rendszerek modellalapú fejlesztése

A hibakritikus alkalmazások legnagyobb részét beágyazott rendszereken használják. Ide lehet sorolni például a repülőgépek vezérlőit, a vasúti biztosítóberendezéseket, autók elektronikáját, stb. Ezen rendszerek esetében egy meghibásodás katasztrofális következményekkel (akár emberéletek elvesztésével) is járhat, így nagyon szigorú szabályozások vannak az ilyen programok helyességének igazolására vonatkozóan.

Egy lehetséges, az előírásoknak megfelelő fejlesztési módszer a formálisan ellenőrzött modelleken alapuló tervezés, majd a modell alapján történő forráskód generálás.

A felmerülő problémák felvezetéseként először röviden ismertetem a kódgenerátort is alkalmazó modell alapú fejlesztés tipikus lépéseit.

Az első két lépés a modell elkészítése és verifikálása. Ennek során az adott problémának elkészítjük a formális reprezentációját, majd megfogalmazzuk azokat a kritériumokat, amik teljesülése esetén a viselkedést helyesnek ítéljük. A verifikáció során matematikai ellenőrző eszközök segítségével belátjuk, hogy a modellünk megfelel az elvárt viselkedésnek, és a kívánt algoritmust valósítja meg.

A következő lépés a generált kód kívánt paramétereinek kiválasztása. A kódgenerátor képes akár arra is, hogy a kódba bizonyos kiegészítéseket írjon, melyekkel az alapvető funkcionalitás bővíthető – ilyen például a naplózó funkció illesztése vagy a platformnak megfelelő függvénykönyvtár kiválasztása.

A modell és a megfelelő paraméterek kiválasztása után már a konkrét kódgenerálás következik: itt az eszköz a megadott modellt beolvassa, és annak nyelvi elemeit a forráskód szintjére képezi le. A modell alapján sok esetben csak kódvázat (pl. vezérlési struktúrát) generálnak, amibe a konkrét műveletek (pl. vezérlés kiadása, kommunikáció) kódját kézzel írja be a programozó, de a megfelelő függvénykönyvtárak használatával az is lehetséges, hogy az így kapott kód emberi beavatkozás nélkül működőképes és fordításra készen áll.

Az utolsó lépés a kész alkalmazás telepítése az adott platformra.

A fenti (nagyon rövid) leírásnak megfelelő kódgenerátort készítettük el egy korábbi dolgozat részeként.

Ilyen módon a formális modell ellenőrzésével biztosított a tervezett algoritmus helyessége, ugyanakkor felvetődik a generált kód és a modell viszonyának kérdése: mennyire feleltethető meg a futó kód viselkedése a (bizonyítottan helyes) formális modellnek?

Az implementáció során ugyanis számos probléma léphet föl:

- A manuális kódolás vagy az automatikus kódgenerálás algoritmusának hibája,

- A platform (pl. egy beágyazott mikrokontroller és a hozzá tartozó kommunikációs könyvtárak) szolgáltatásainak nem megfelelő figyelembevétele a generált kódban,
- A generált kód nem megfelelő fordítása és telepítése.

Felmerül a kérdés, hogy a lehetséges hibákat detektálni képes ellenőrzéseket nem lehet-e automatizálni, vagyis a formális modellből generált (és telepített) kódot emberi beavatkozás nélkül, szisztematikus módszerekkel verifikálni?

A kérdés megválaszolása előtt a fejezet további részében bemutatom az elkészült kódgenerátort olyan mélységben, ami szükséges a későbbi ellenőrzések megértéséhez.

2.2 Az elkészült kódgenerátor

2.2.1 A kiindulási modell

A kódgenerátor bemenetéül szolgáló formális modell az *időzített automaták*, melyhez szerkesztő és ellenőrző eszközül az UPPAAL [2] modellező és modellellenőrző rendszert választottuk.

Az időzített automaták formális definíciója[3][4] a következő:

A definíció során az órák halmazát jelöljük C-vel, B(C)-vel pedig az órák közötti, alábbi formulának megfelelő kapcsolatokat: $x \bowtie c$, vagy $(x-y) \bowtie c$, ahol $x,y \in C$, $c \in \mathbb{N}$, és $\bowtie \in \{<, \leq, =, \geq, >\}$. Az időzített automata ezek után egy olyan (L, l0, C, A, E, I) hatos, ahol L az állapotok (vezérlési pontok) halmaza, $l0 \in L$ a kezdőállapot, C az órák halmaza, A az akciók halmaza, $I: L \rightarrow B(C)$ pedig az állapotokhoz tartozó invariáns-hozzárendelés. $E \subseteq L \times A \times B(C) \times 2C \times L$ jelöli a két állapot közötti, akcióval, őrfeltétellel ellátott, és az órák esetleges nullázását tartalmazó éleket.

Kevésbé formálisan az időzített automatákat (jó közelítéssel) olyan állapotgépekként képzelhetjük el, ahol:

- az egyes állapotokban tartózkodásnak létezhet feltétele (invariáns),
- az egyes átmenetek tüzelhetőségének létezhet feltétele (őrfeltétel),
- a használt változók értékét az átmenetekhez tartozó akciók segítségével módosíthatjuk,
- az órákat az egyes őrfeltételekben a felsorolt relációkkal kontextusban alkalmazhatjuk,
- az óráknak akciók keretében csak 0 értéket adhatunk.

Ezek után az időzített automaták formális szemantikája is definiálható:

Egy óra kiértékelése egy $u: C \rightarrow \mathbb{R}_{\geq 0}$ leképezés. Legyen R^C az összes kiértékelések halmaza. Legyen $u_0(x) = 0$ minden $x \in C$-re. Az $u \in I(l)$ jelölést használjuk, ha u kielégíti I(l)-t. Legyen (L, l0, C, A, E, I) egy időzített automata. A szemantika az átmenetek egy $(S, s_0, \rightarrow)$ rendszere, ahol $S \subseteq L \times \mathbb{R}^C$ az állapotok halmaza, $s_0 = (l_0, u_0)$ a kezdőállapot, és $\rightarrow \subseteq S \times \{R_{\geq 0} \cup A\} \times S$ az átmenet, úgy, hogy:

$(l, u) \rightarrow (l, u+d)$ ha $\forall d': 0 \leq d' \leq d \Rightarrow u+d' \in I(l)$, és

$(l, u) \rightarrow (l', u')$, ha $\exists e = (l, a, g, r, l') \in E$, ahol $u \in g$, $u' = [r \rightarrow 0]u$, és $u' \in I(l)$,

ahol d $\in \mathbb{R}_{\geq 0}$, u + d minden C-beli x óra értékét az u(x) + d-re állítja, és [r → 0]u jelöli azt az órakiértékelést, amely minden r-beli órát 0-ra állít, és megegyezik u-val C \ r fölött.

A modell működését egyszerűsítve az egyes állapotváltásokkal írhatjuk le. Engedélyezettnek tekintjük azt az állapotváltást, ahol:

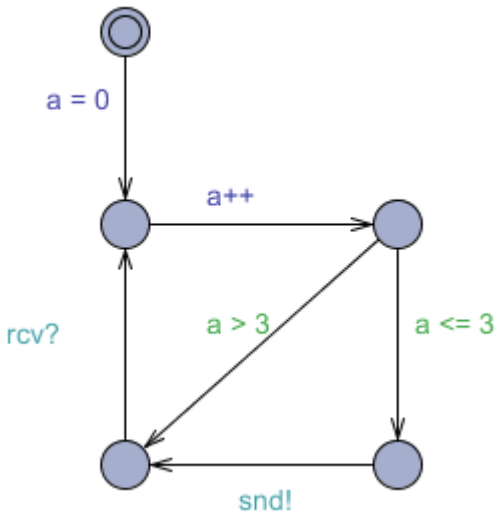
- A vezérlési pont azonos marad (tehát tranzíció nem hajtódik végre), és az állapotra vonatkozó invariáns az új állapotban is teljesül. Ilyen állapotváltást az órák értékének változása idézhet elő.
- Két vezérlési pont közötti tranzíció tüzelhető, vagyis a tranzícióra vonatkozó őrfeltétel teljesül, és a tranzíció végrehajtása után az új állapotban sem sérül az invariáns kritérium.

Az imént ismertetett időzített automaták hálózatokba szervezhetőek szinkronizáló akciók segítségével, így definiálható egy időzített automata hálózat. A két komponens közti szinkronizáció megvalósítása randevú jellegű, tehát a szinkronizáció a résztvevők között egyszerre zajlik le (vagyis több átmenet tüzel egyszerre). Ennek megvalósítása a modellben definiált csatornákon keresztül történik: az együtt lépő átmenetek a csatorna fölötti szinkron kommunikációt írják le. Lehetőség van üzenetszórás jellegű szinkronizációra is, ilyenkor az egyszerre végrehajtódó átmenetek száma 1..n között lehet. A kezdeményező átmenet mindenképp tüzel (tehát az előző esettel ellentétben nem blokkoló üzenetküldésről beszélhetünk), a fogadó oldalon pedig tetszőleges számú komponens állhat.

A fent leírt hálózatok tetszőleges számú (n) automatából állhatnak. Az automaták állapotai együttesen határozzák meg a rendszer egészének állapotát.

Ilyen időzített automata hálózatok kezelésére szolgáló eszköz az UPPAAL. Konkrét szintaxist, és hozzá tartozó grafikus reprezentációt ad meg, melyekkel megvalósítja a formalizmust, és segíti annak használatát. Lehetőséget ad a követelmények formális megfogalmazására, és ezek automatikus ellenőrzésére. A követelmények ellenőrzését, megfogalmazását bővebben a 3.4.1. fejezetben mutatom be.

Az 1. ábra egy egyszerű UPPAAL modellt mutat be.



1. ábra: Egyszerű UPPAAL modell

Az átmenetekhez tartozó zöld feliratok jelölik az őrfeltételeket, a kékek pedig az akciókat. A kékeszöld színű („?” vagy „!” végű) feliratok a szinkronizációs akciók. A kezdőállapotot a kettős kör jelöli.

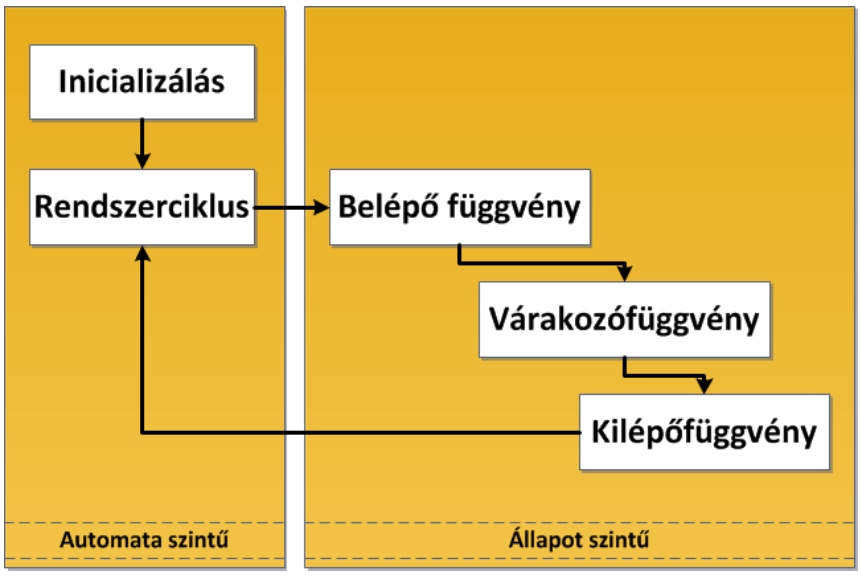
2.2.2 Kódgenerálási módszer

A fentiek alapján elmondható, hogy az időzített automaták tulajdonképpen állapotokból és állapotátmenetekből álló formális modellek, ahol a működés elemi lépéseit az egyes átmenetek tüzelése, ezen belül a hozzájuk tartozó őrfeltételek és szinkronizációs lépések kiértékelése, a forrásállapotból való kilépés, az akciók végrehajtása, valamint a célállapotba való belépés adják. Ennek megfelelően a kódgenerátorban a szisztematikus leképezés alapját az adta, hogy ezeket a nyelvi elemeket forráskód szinten megfeleltettük egy-egy függvénynek. A teljes lista bemutatása nem szükséges az aktuális téma tárgyalásához, viszont az alapvető függvények megismerése fontos. Minden állapothoz létrehoztunk egy állapotba való belépést, egy abból való kilépést, és egy állapotban tartózkodást kezelő függvényt.

Egy-egy automatához tartozó generált kódban helyet kap egy inicializáló függvény, ez a megszokott módon a változók, mutex-ek, kommunikációs alrendszerek inicializálását végzi. Ennek lefutása után a vezérlést megkapja a fő rendszerciklus, amelynek a feladata a folyamatos működés biztosítása – ez a függvény fut, ha semmilyen egyéb teendője nincs a programnak. Ezek a függvények együttesen felelősek azért, hogy az automata szintű működés biztosított legyen.

Ha egy állapotátmenet végrehajtható (a fent leírt szemantika szerint), akkor meghívódik a következő állapotra generált belépőfüggvény. Ennek feladata az állapothoz tartozó változók inicializálása, valamint (ha a kimeneti átmenetek tartalmaznak szinkronizációt) a szinkronizációs csatornák inicializálása. Lefutása után átkerül a vezérlés a konkrét állapothoz tartozó várakozófüggvényhez, amely azt vizsgálja, hogy a kimeneti állapotátmenetek közül van-e végrehajtható. Ha talál egyet, meghívja a saját állapotához tartozó kilépőfüggvényt. Ez állítja le a kommunikációs csatornák figyelését (szinkronizáció esetén), valamint kezeli az állapotátmenetet – ennek hatására hívódik meg a következő állapothoz tartozó belépőfüggvény, és záródik le az átmenet.

A fent leírtakat szemlélteti a 2. ábra.



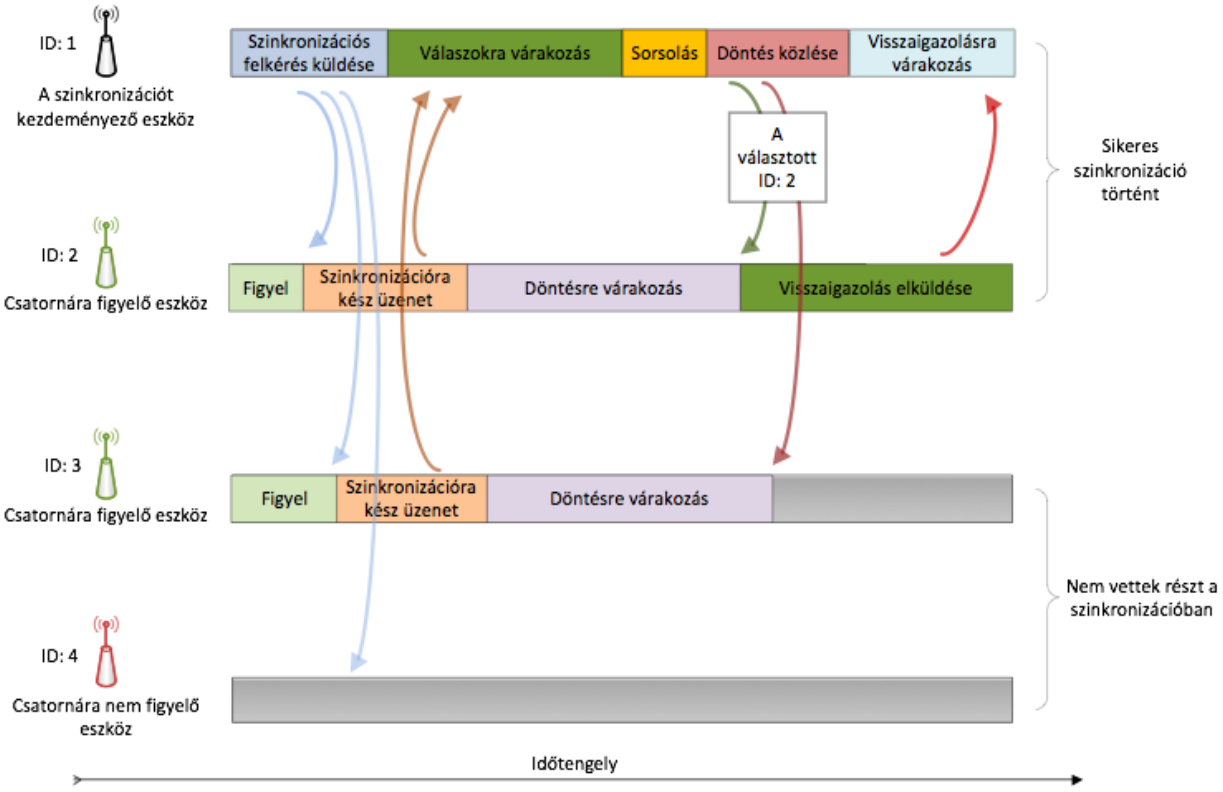
2. ábra: A generált kód működésének vázlata

Fontos tulajdonsága a kódgenerátornak, hogy az elkészült C nyelvű kód minden módosítás nélkül fordítható és futtatható, így a leképezés helyességének belátása jelentős lépés az alkalmazás egészének verifikációjában.

2.2.3 Platformszolgáltatások és kiterjesztések integrálása

A kódgenerátor megalkotásakor nagy figyelmet fordítottunk arra, hogy a generált kód platformfüggetlen legyen. A felhasznált hardverközeli szolgáltatások miatt ez nem megvalósítható egyetlen kellően általános kóddal, így az általunk választott módszer az ilyen szolgáltatások generált kódból történő kiemelése (platformfüggő könyvtári függvényként való megvalósítása) volt.

A formalizmus szemantikájának leképezéséhez a platform szintjén meg kellett valósítanunk az időkezelést, valamint – az időzített automaták esetén használatos szinkronizációs formák implementálásához - egy komplett kommunikációs alrendszert is. Ennek szemléltetésére szolgál a 3. ábra, mely egy (két résztvevős) szinkron csatornához szükséges, a platformszolgáltatásban megvalósított kommunikációs lépéseket mutatja be. Az üzenet elküldése után a küldő fél egy adott ideig várakozik a válaszra. Ez idő alatt a végrehajtásra készen álló komponensek visszajelzést küldenek, és a küldő fél kiválasztja (véletlenszerűen, a modell szemantikáját követve) azt a résztvevőt, akivel a szinkronizációt végrehajtja. Egy újabb sikeres üzenetváltást követően lezajlik a tényleges állapotváltás.



3. ábra: A két résztvevős szinkronizáció megvalósítása

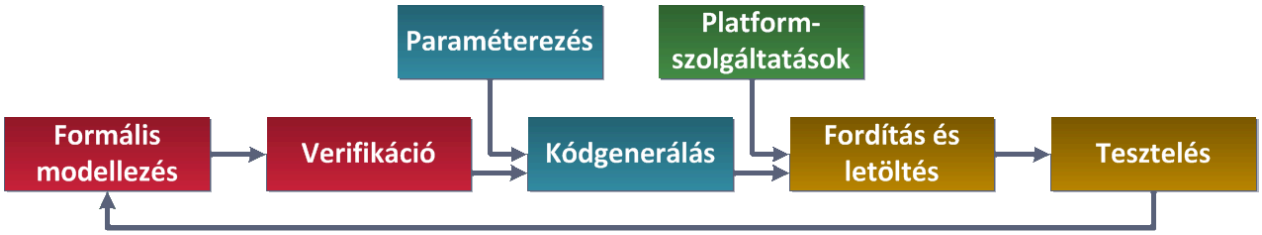
A külvilággal való interakció támogatásához szükséges volt az erre irányuló funkciók kiemelése. A felhasználói bemenet minden alkalmazás esetén mást jelenthet (pl. egy gombnyomás, többállású kapcsoló átállítása, stb.), ezért ezek általános kezelését úgy oldottuk meg, hogy minden bemenetet egy szinkronizációnak tekintünk. A bemenet változása egy, az automaták szinkronizálásával azonos alapon működő folyamatot indít el, melynek végeredménye a szinkronizációs csatornához tartozó változó értékadása.

A kimenet megvalósítása is hasonló: ilyenkor a program először beállítja a belső változó értékét, majd szinkronizációt kezdeményez egy előre megadott csatornán.

Ezen szolgáltatások konkrét függvényekben való megvalósításának a generált kódhoz, illetve modellhez való kötése igen egyszerű. A platformfüggő komponensek implementációjához segítséget nyújt a kódgenerátor által hivatkozott függvények precíz illesztési felülete: adott a függvény definíciója (bemenetek, elvárt kimenet), valamint a hozzá tartozó leírás az elvárt működésről. Ezek alapján az implementáció bármilyen platformon lehetséges. A kommunikációs csatornák, és a hozzájuk tartozó változók felismerése pedig elnevezési konvenciók alapján történik, így a modell megalkotásakor elegendő erre figyelni.

A platformfüggő szolgáltatások módszerét használva a kódgenerátor támogatja az egyes felhasználói kiterjesztések integrálását is. Ezek olyan kódrészletek, melyek a fő funkcionalitást nem befolyásoló kiegészítéseket adnak a kódhoz, erre egy jó példa a naplózó funkció (melyet a lehetőség demonstrálása céljából a kódgenerátorral együtt megvalósítottunk). Ennek segítségével az egyes állapotváltásokat, változók értékadásait lehet elmenteni az egyes automatákhoz tartozó kódrészletekből.

A kódgenerátor felépítését, a kódgenerátorral történő fejlesztés folyamatát szemlélteti a 4. ábra.



4. ábra: Fejlesztés kódgenerátorral

2.3 Verifikáció kritikus rendszerekben

A verifikációs követelmények szemléltetésére az EN 50128:2011-es szabványt [5] választottam. Ez a szabvány a vasúti jelzőrendszerekhez és biztosítóberendezésekhez tartozó szoftverfejlesztési követelményeket írja le.

A szabvány három szintre helyezi a fejlesztés során felhasznált eszközöket. A kódgenerátor a legszigorúbb szabályozás alá eső, T3 kategóriába esik: olyan kimenetet állít elő, ami közvetlenül kapcsolódik a kritikus rendszeren futó kódhoz.

A T3 szintű eszközöknél követelmény, hogy a kimenet helyessége bizonyított legyen, vagy létezzen biztos módszer a kimenet hibájának detektálására. Erre az egyik lehetőség a felhasznált eszköz validációja (további lehetséges módszerek: redundáns kód készítése, az eszköz megfelelése a szükséges SIL (Safety Integrity Level, biztonságintegritási szint) előírásnak, stb.). A hibák kezelését minden esetben meg kell oldani. Az eszköz validációja során rögzíteni kell a végrehajtott teszteseteket, és a hozzájuk kötődő naplót. Szintén követelmény, hogy a tesztkészletnek megfelelő (a pontos definíció eszközönként eltérhet) fedettséget kell biztosítania.

Ha a helyességbizonyítás illetve SIL előírásoknak megfelelő tanúsítás nem áll rendelkezésre, akkor az eszköz használatának eredményeként kapott kimenet vizsgálata, a hibák detektálása és kezelése szükséges. Példának a szabvány egy fordító (compiler) ellenőrzését említi. Egy fordító használata akkor tekinthető megfelelőnek, ha teszteléssel sikerült minden fordított kódrészletet elérni, és azok helyesen működnek. Ha a forráskódban a tesztelés során elérhetetlen kódrészlet kap helyet, annak működését külön kell igazolni. Olyan tesztek elvégzése is szükséges, amelyek az alkalmazás reakcióját vizsgálják a fordító esetleges hibáira. A tesztelést minden alkalommal el kell végezni, amint változás következik be a fordítóban vagy a forráskódban.

A szabvány Translatornak nevezi azokat az eszközöket, melyek a fejlesztési folyamat során egy absztrakciós szintről egy másikra való leképezést végzik. Ide tartoznak a különböző tervezéstámogató eszközök, linkerek, illetve a kódgenerátorok is (hiszen a modellezési szintről a forráskód szintjére végeznek szintézist). A dokumentum külön kitér az ilyen eszközök kezelésére: a tesztelést ezen esetekben egy adott alkalmazás egészére, vagy az alkalmazások egy adott osztályára kell elvégezni. Leírja, hogy a Translatorok teszteléséhez a dinamikus tesztelés lehet a fő eszköz, kiemelve, hogy lehetőség szerint automatikusan előállított tesztkészletet kell használni.

A szabvány követelményei alapján elmondható, hogy a tesztkészlet automatikus előállítása és a tesztek automatikus végrehajtása, kiértékelése gyakorlatilag elengedhetetlen a szigorú követelményeknek való megfeleléshez.

3 Az ellenőrzési koncepció

Az eddigiekben bevezettem az időzített automaták formalizmusát, majd bemutattam az alapul használt kódgenerátor működését nagy vonalakban. Ebben a fejezetben azt vizsgálom, hogy a kész alkalmazás és a formális modell közötti relációt milyen módszerrel lehet a legjobban megállapítani.

3.1 Meglévő megoldások

A lehetőségek megismeréséhez szükséges volt a már meglévő megoldások áttekintése. Ezek közül legelterjedtebbek a teszteléssel történő ellenőrzést megvalósító módszerek illetve eszközök.

Mivel az UPPAAL nagy népszerűségre tett szert egyszerű, felhasználóbarát használata és potens verifikációs képességei miatt, ezért már sokan foglalkoztak az UPPAAL-alapú teszteléssel, teszteset-generálással.

Az első említésre méltó eszköz a CoVer [6][15], amely segítségével automatizálhatjuk az automatákhoz köthető tesztesetek előállítását. Ehhez az eszköz úgynevezett megfigyelő (observer) automatákat használ, melyeknek célja, hogy pontosan megadják egy modellen belül a vizsgált komponenst és tesztelési célt. Itt a fedettségi kritériumok változó alapúak (pl. olyan tesztkészlet előállítása, mely lefedi az X változó értékadásait illetve annak felhasználásait). A tanszéki próbák során az eszköz használata nehézkesnek és instabilnak bizonyult, a megfigyelő automaták összeállítása nem elég intuitív. A változó alapú teszteset generálás miatt a specifikusabb fedettségi kritériumok definiálása manuális observer automata írással jár.

Egy másik, tesztelést segítő eszköz a TRON [7][16]. Ez egy valós idejű, bemeneti és kimeneti konformancia (input/output conformance – IOCO) reláció vizsgálatára alkalmas eszköz. A TRON egy teszt primitívek (tehát nem összetett tesztkészletek) generálására és futtatására alkalmas eszköz, amely véletlenszerű időközönként véletlenszerű bemenetet ad a vizsgált komponens bemenetére. Ez után képes a kapott kimenetet összevetni a modell alapján elvárttal, és így az adott esetre az IOCO reláció fennállását igazolni vagy cáfolni. A futtató eszköz platformfüggő, a vizsgálandó alkalmazáshoz adaptert kell készíteni, ami az elvárt interfészhez köti a ki- és bemeneteket. Az összehasonlítás alapjául szolgáló modellt tekintve igen engedékeny a TRON, gyakorlatilag bármilyen UPPAAL modellel képes dolgozni (nem merül fel megkötésként a determinizmus, az órák korlátozása, stb.). Az eszköz látványos eredményeket tud produkálni, és széles körben elterjedt. Azonban korlátozza a platformfüggő megvalósítás, és a szisztematikus teszteset-generálás hiánya. Bár a véletlenszerű teszteléssel hosszú vizsgálat esetén kaphatunk teljes fedést, a kritikus rendszerek esetén ez a „tapasztalati szabály” nem elég bizonyíték. Mivel az ellenőrzés valós idejű, a TRON nem alkalmas az IOCO-tól eltérő relációk vizsgálatára (a relációkról bővebben a 6.2-es fejezetben lesz szó). Továbbá az eszköz megvalósítása (az eszközhöz kapcsolódó tesztfuttató konzol) nehézkessé teszi az elosztott rendszerekben történő alkalmazást.

A következő releváns eszköz a JTorX [8]. A TRON-hoz hasonlóan ez az eszköz is IOCO vizsgálatot végez, ám sokkal szélesebb körben. A támogatott eszközök skálája igen sokrétű: az UPPAAL mellett számos elterjedt modellező eszköz formátumát támogatja,

melyek közül a legtöbb címkézett tranzíciós rendszer (Labelled Transition System, LTS) reprezentációt jelent. A JTorX is egy futó alkalmazáshoz kapcsolódva végzi a tesztelést, a módszer is hasonló a TRON-nál ismertetetthez. Adaptereket kell készíteni, amivel kapcsolódni tudunk a célplatform ki- és bemenetéhez. Nagy könnyebbség, hogy a standard be- és kimenetek támogatása, valamint a TCP kommunikáció illesztése beépítve érkezik az eszközzel, így számos alkalmazás készen („out of the box") tesztelhető. Az ellenőrizendő modell vizsgálata valós időben történik, így lehetővé téve a végtelen állapotterű modellek ellenőrzését is. A három említett eszköz közül egyértelműen a JTorX a legfejlettebb, mind a felhasznált technológiát, mind pedig a megvalósítást illetően. A számos funkció (melyek közül csak néhányat emeltem ki) mellett azonban vannak hiányosságai is a programnak. Nem képes automatikus teszteset-generálásra, a teszteseteket a felhasználónak kell megadnia (LTS formátumban, vagy a vizsgált modell kiegészítésével). Ez az eszköz is IOCO relációk megállapítására alkalmas. A TRON és a JTorX esetében is probléma, hogy beágyazott rendszerek esetében a fejlesztés során külön figyelmet kell fordítani az adapter megvalósíthatóságára. Egy valós, ipari (elosztott) alkalmazás fejlesztésénél nem mindig van igény arra, hogy az egyes komponensek belső működése a külvilág számára megfigyelhető legyen, így az adapterek megalkotása nehézkes volna. A fenti eszközök adott platformhoz kötött megvalósítása pedig nem teszi őket alkalmassá beágyazott rendszerekre (más platformokra) történő telepítésre. Így a platformszolgáltatások helyességének ellenőrzése nagyon nehézkessé válik – például az időzítés esetében a teszt futtatónak nincs közvetlen hozzáférése a konkrét megvalósítással nem támogatott futtató platform által használt órához, stb.

3.2 A lehetőségek áttekintése

A 3.1. fejezet alapján látható, hogy a rendelkezésre álló eszközök nem felelnek meg teljes körűen az elvárásainknak, ezért egy teljesen új ellenőrzési módszert dolgoztam ki. Az alapvető megközelítést illetően két út állt előttem.

Az első lehetőség, hogy a program alapján újra elkészítem a rendszer modelljét, vagyis a forráskód alapján egy új, az eredetitől potenciálisan eltérő modellt kapunk. Ekkor a feladat a két modell (az eredeti, verifikált formális modell, és a programból előállított tényleges modell) összehasonlításának elvégzése, és az azok közötti reláció megállapítása. Ennek a megközelítésnek az előnye, hogy bármilyen kód és modell között elvégezhető az ellenőrzés, nem szükséges a kódot teszteléssel történő vizsgálathoz felműszerezni, és a visszafejtett modellen (amennyiben ez eltér az eredetitől) is el lehet végezni a verifikációs lépéseket. A módszer hátránya viszont, hogy ilyen módon csak a létrejött kódot tudjuk vizsgálni – tehát nincs lehetőségünk kitérni a kód és a futtató platform, a felhasznált könyvtárak közti együttműködés helyességére.

A másik lehetőség, hogy a modell alapján vizsgáljuk a kész alkalmazást, a modell és az alkalmazás viselkedése közötti kapcsolatot. Az alkalmazás viselkedését annak végrehajtásával figyeljük meg. Lényeges különbség az előző ponthoz képest, hogy a vizsgálatot már a konkrét platformszolgáltatásokkal, könyvtárakkal végezzük el, így lehetőségünk van a teljes, későbbiekben ténylegesen telepített kód viselkedésének ellenőrzésére. Ehhez dinamikus vizsgálat, tesztelés szükséges, így a kihívás is ebből fakad: a kimerítő teszteléshez szükséges tesztkészletet kell kidolgozni[13], végrehajtani és értékelni. A módszer előnye továbbá, hogy a folyamat konstruktív, azaz az

ellenőrzés lépései sorrendileg megegyeznek a program elkészítésének lépéseivel, nem kell „visszafele” haladni (pl. kódból újra modellt alkotni). Emiatt a tesztelés előkészítésével (tesztesetek megalkotása, optimalizálás, stb.) nem kell megvárni a fejlesztés végét, amivel időt takaríthatunk meg.

Mivel a célunk a modellalapú alkalmazásfejlesztés teljes támogatása volt, ezért a második módszerre esett a választásunk. Ehhez a módszerhez rendelkezésre áll a modell (a tervezés során kézzel készítendő, az egész folyamat alapja), valamint az implementáció (a kódgenerátor segítségével készülhet), így ténylegesen a tesztelésen és annak automatizálásán volt a hangsúly.

A 3.3-as fejezetben a választott ellenőrzési folyamat lépéseit mutatom be.

3.3 Az ellenőrzés módszere

Az első feladat tehát az, hogy egy adott időzített automata modell alapján a szisztematikus teszteléshez szükséges teszteseteket megalkossuk. A kívánt tesztkészlet meghatározása (modellfüggő) fedettségi kritériumok segítségével történhet. A létrejött tesztkészlet alapján a teszteket végre kell hajtanunk, majd az eredményt kiértékelnünk.

A diagnosztikai információk utólagos kinyeréséhez biztosítanunk kell a vizsgálni kívánt alkalmazás megfigyelhetőségét is. Ezt is el tudjuk végezni még a modell szintjén: olyan plusz (eddig nem használt, tehát a logikai viselkedésre hatással nem lévő) változókat veszünk fel a kész modellbe, melyek segítségével később a program által bejárt út (végrehajtott utasítássorozat) egyértelműen visszakövethető, azonosítható.

A tesztkészlet összeállítása a modell alapján automatikusan elvégezhető, ennek részleteit a 4.2.2. fejezetben ismertetjük.

A kód ellenőrzése nem valós időben történik. A kód futása közben (a kódgenerátor naplózó funkcióját kihasználva) naplót készítünk a bejárt állapotokról, ez adja meg a vizsgálandó kimenetet. Az értékelés ennek a kimenetnek és az adott tesztek generálása során bejárt állapotok (referencia kimenet) összehasonlításával történik. Az összehasonlítás eredménye tetszőlegesen részletes lehet: az alapvető „GO/NO GO” információn felül megkaphatjuk az eltérés helyét és irányát is, amivel pontosabb diagnosztikát tudunk adni. Szintén ilyen módon tudjuk szűrni a nem specifikált viselkedést, vagyis a futó kódnak a modelltől való olyan irányú eltérését, amelyben a modellnek való megfelelésen túl (nemkívánatos) plusz működést is produkál. A szűrések elvégzéséhez felhasznált módszereket a 4.2.5. fejezetben mutatom be.

3.4 Szükséges technológiák

3.4.1 Automatikus teszteset-generálás

A teszteset-generáláshoz az UPPAAL modellellenőrzőjét használtam. Ez a modellellenőrző adott logikai kifejezések állításait igazolja (vagy cáfolja) a modellen. Az állítások megfogalmazásához egy redukált temporális logikai kifejezéskészlet (Computational Tree Logic, CTL) áll rendelkezésre [9], melyet a következőekben röviden ismertetek.

Legyen S (s_1, s_2, …) a rendszer állapotainak halmaza. Legyenek továbbá p és q az állapotokra megfogalmazott mellékhatásoktól mentes állítások. Ezek szintaktikája hasonló a C-ben, Java-ban megismertekhez, tehát használhatunk zárójeleket,

egymással vagy konstanssal való összehasonlítást, Bool-logikai kifejezéseket, stb. Ezeken felül az állítások vonatkozhatnak egy állapot bekövetkezésére is: ezek vizsgálata *<rendszerbeli komponens neve>.<állapot neve>* formában történhet, ahol a rendszer az időzített automaták hálózatát jelenti, egy komponens pedig egy időzített automatát. Ez a szintakszis használható többkomponensű rendszerek esetén a változók elérésére is. Egy speciális állítás a deadlock (holtpont), ami egy olyan állapotot szimbolizál, melyből az adott lefutás során nem tudunk továbblépni. Az állapotok között „ha p akkor q is" jellegű állításokat is megfogalmazhatunk, erre szolgál az imply kulcsszó (p imply q).

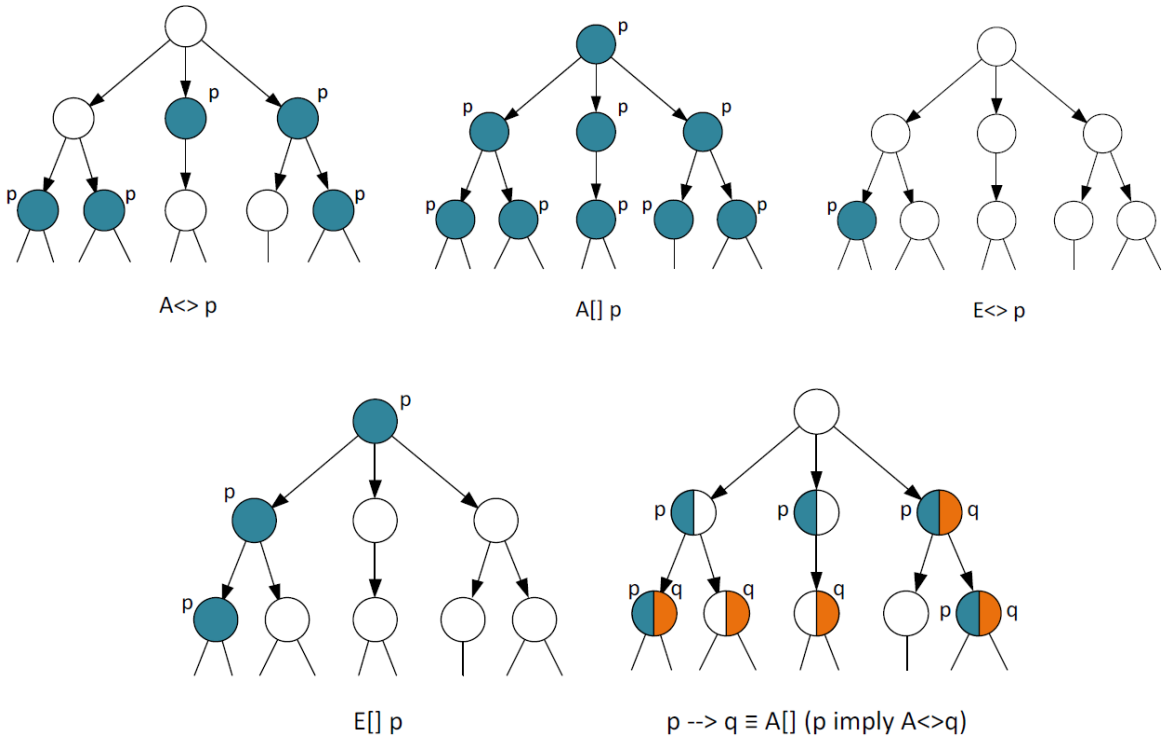
Használhatunk továbbá temporális logikai operátorokat a következők megfogalmazására:

- Az egy állapotból induló végrehajtási útvonalak számának jellemzésére: az E (∃) kvantor egy adott tulajdonságú végrehajtási útvonal létezését írja elő, az A (∀) kvantor pedig azt fogalmazza meg, hogy egy adott állapotból induló minden útvonal adott tulajdonságú legyen.
- Egy-egy végrehajtási útvonal mentén található állapotok jellemzésére: a <> (future) operátor egy adott tulajdonság jövőbeli előfordulását írja elő, a [] (global) operátor pedig azt fogalmazza meg, hogy egy adott útvonal mentén minden állapot adott tulajdonságú legyen.

A kvantorok és operátorok sorrendje kötött, kvantort operátornak kell követnie. Ezért a gyakorlatban összetett operátorok (pl. A[]) alakulnak ki, ezek segítségével formalizálhatók a modellel szembeni követelmények. A fenti bevezetés után p-re és q-ra definiálhatjuk a következő tulajdonságokat:

- Létezik: E<> p igaz akkor és csak akkor, ha létezik olyan $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$ átmenet-sorozat, hogy s_0 a kezdőállapot, és s_n kielégíti p-t.
- Invariáns: A[] p igaz akkor és csak akkor, ha minden elérhető állapot kielégíti p-t. Másképp felírva: not E<> not p.
- Lehetséges útvonal invariáns: E[] p igaz akkor és csak akkor, ha létezik olyan végrehajtási útvonal, melynek mentén p igaz minden állapotra, és az útvonal vagy (1) végtelen, vagy (2) véges, és az utolsó állapotból nincs kimenő átmenet, vagy (3) az utolsó állapotban tetszőleges idejű várakozás mellett p és az állapot invariánsa is igaz.
- Eshetőség (elérhető állapot) minden útvonalon: A<> p igaz akkor és csak akkor, ha minden lehetséges végrehajtási útvonalon a rendszer végül eljut egy olyan állapotba, melyre teljesül p.
- Feltételes bekövetkezés minden útvonalon: p --> q azt jelenti, hogy ha egy állapotban p igaz, akkor az abból kiinduló minden végrehajtási útvonalon valahol q is igaz lesz. Ez a tulajdonság kifejezhető az A[] (p imply A<> q) kifejezéssel is.

Az állítások jelentését szemlélteti az 5. ábra.



5. ábra: CTL kifejezések jelentése

Néhány egyszerű példa CTL kifejezés:

- A[] !deadlock: a rendszer holtpontmentes.
- A[] (P1.state1 imply P1.var1==2): P1 komponens state1 állapotában a var1 változó értéke mindig 2.
- E<> P1.state1: P1 komponens el tud jutni state1 állapotba.

A modellellenőrző tehát a fenti formátumban vár követelményspecifikációt. Ezek teljesülése esetén az eszköz tájékoztat a sikerről. Azonban ha valamelyik állításunk nem állja meg a helyét, az eszköz képes arra, hogy ellenpéldát mutasson.

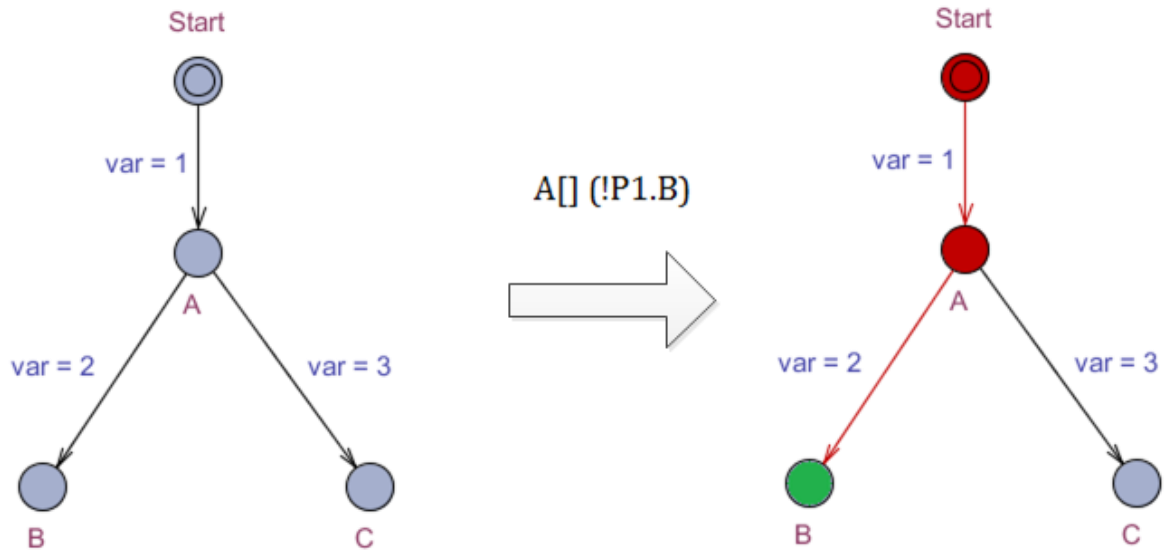
Az ellenpélda azt mutatja meg, hogyan talált az ellenőrző olyan állapotot, amire a követelményspecifikáció nem teljesül. Ezért az ellenőrző folyamatosan naplózza a bejárt állapotokat, a végrehajtott lépéseket. Amikor ellenpéldát kapunk, valójában ezt a nyomvonalat (simulation trace) kapjuk meg: a bejárt állapotok, a végrehajtott átmenetek, a belső változók értékei, a komponensek közötti kommunikáció naplóját.

Ilyen módon a meghatározott temporális logikai kifejezés verifikálása során olyan bejárási útvonalat kapunk, amely a későbbiekben számunkra nagyon hasznosnak bizonyul. Ha ugyanis *negatív* követelményspecifikációt adunk meg, az adott CTL kifejezés nem állja meg a helyét, tehát az ellenőrző által visszaadott ellenpélda (napló) a modell *helyes* működésének egy esete lesz. Vagyis gyakorlatilag megadja a modell által elvárt bemeneteket, és az erre válaszként megkövetelt viselkedést, ezért az ilyen módon generált nyomvonal a kész alkalmazás szempontjából absztrakt tesztesetként értelmezhető.

Ezt szemlélteti egy egyszerű példán a 6. ábra. Itt a modellen egy állapot (egészen pontosan a P1 automata „B” állapota) bekövetkezését szeretnénk vizsgálni, ezért megfogalmazunk egy CTL kifejezést amely szerint az állapot nem érhető el:

A[](!P1.B)

A modellellenőrző pedig ad egy lefutást (ellenpéldát), amely végén éppen a kívánt állapotba kerül a rendszer – ez jelenleg a Start – A – B állapotsorozat.



6. ábra: Állapot elérésére vonatkozó negatív kifejezés

3.4.2 Tesztek futtatása

A tesztesetek futtatásának alapgondolata egyszerű: az absztrakt tesztesetek alapján el kell készítenünk a teszteset végrehajtásához szükséges bemeneti információk listáját, majd ezeket a megfelelő időben meg kell adnunk az alkalmazás számára.

A megvalósítás nehézsége a platformfüggetlenség fenntartásából fakad. A bemeneteket egy olyan csatornán kell biztosítanunk (kommunikálnunk a tesztelendő alkalmazás felé), amiről előzetesen semmilyen információnk nincsen. A konkrét megvalósítást a 4.2.4. fejezetben mutatom be.

3.4.3 Teszt eredmények kiértékelése

A tesztek értékelése az absztrakt tesztesetek és az alkalmazás által előállított kimenetek alapján történik. Az eredmények kiértékelésénél már feltételezzük, hogy a bemeneti adatok a helyes sorrendben (és időben) lettek szolgáltatva, tehát a lefutás a tesztesetnek megfelel – ennek biztosítása a teszt végrehajtó feladata. A legalapvetőbb kiértékelési funkciók (GO/NO GO) ellátásához elegendő lenne (az alkalmazás naplózásának formátumát megkötve) a kimeneti napló „egy az egyben” történő összevetése az absztrakt tesztesetben kapott nyomvonallal. Ez azonban implementációs szempontból nehézkes, valamint nem támogatja a bonyolultabb kiértékelések elvégzését.

Ezért a kiértékelő modul valójában egy értelmező eszköz, amely fel tudja dolgozni a modellellenőrző kimenetét és az alkalmazás naplóját is. A feldolgozás eredménye egy belső reprezentáció, amin a megalkotás után sokkal könnyebben végezhetünk további műveleteket.

Az összevetés jól meghatározott relációk alapján kell, hogy történjen. Érdemes többféle relációt is támogatni. A leggyakrabban használt a k-ekvivalencia reláció, amely azt mondja ki, hogy a specifikációban és az implementációban azonos bemeneti sorozat mellett azonos kimeneti sorozatot kapunk az első k lépésre. Ez egy egyszerű reláció, amely szigorú követelményeket fogalmaz meg (nem engedélyezett sem szűkítés, sem bővítés az eredeti kimenethez képest).

A modul megvalósítását részleteiben tárgyalja a 4.2.5. fejezet.

3.5 Jellegzetességek

3.5.1 A módszer korlátai

A módszer korlátai jelenleg az UPPAAL formalizmusának korlátaiból fakadnak, amely megengedi a nemdeterminisztikus működést. Ez azt jelenti, hogy ha egy adott állapotból kivezető átmenetek közül több is tüzelhető, akkor nincs közöttük prioritási sorrend – a végrehajtandó tranzíció kiválasztása véletlenszerű.

A fentiekben nem tértünk ki az ebből fakadó irányíthatósági kérdésekre. Hogyan érjük el, hogy az alkalmazás (amely megvalósításában teljesen követi az UPPAAL formalizmust) véletlenszerű választását a tesztesetek végrehajtása során befolyásolni tudjuk?

Ennek megoldásához a modellt a megfigyelhetőségen túl az irányíthatóság szempontjából is fel kellene műszerezni. Ahhoz azonban, hogy a külvilágból származó bemeneteket kezelni tudja a program, a kódgenerátor megvalósítása miatt szinkronizációt kellene használnunk (ahogy azt a 2.2.3. fejezetben írtam). Az UPPAAL azonban egy átmeneten egyszerre csak egy szinkronizációt enged meg. Mivel elképzelhető, hogy ezt az egy lehetőséget már kihasználtuk a modellezés során, ezért erre a kézenfekvő megoldásra nem lehet építeni.

A kódgenerátor működése lehetővé tenné, hogy – a modelltől eltérően – több szinkronizációt végezzen egy átmenet során. Azonban fontos szempont, hogy a kódgenerálás ellenőrizhető modell alapján történjen, és mivel az így irányíthatóvá tett modell szintaktiailag hibás, ezért ezt a kitételt nem teljesíti. Emiatt a jelenlegi megvalósítás nem támogatja ezt a lehetőséget. Mivel az alapvető cél az ellenőrzési módszer helyességének igazolása, ezért (a nemdeterminizmus támogatásának jövőbeni lehetősége mellett) a továbbiakban olyan esetekre koncentrálunk, ahol a modell determinisztikus. Ez a tényleges felhasználást illetően megkötést jelent, de hangsúlyozni kell, hogy a kritikus alkalmazásokban – éppen a verifikációs nehézségek miatt – ellenjavallt a nemdeterminisztikus viselkedés, ezt meg kell szüntetni a modellben még a kód szintézise előtt.

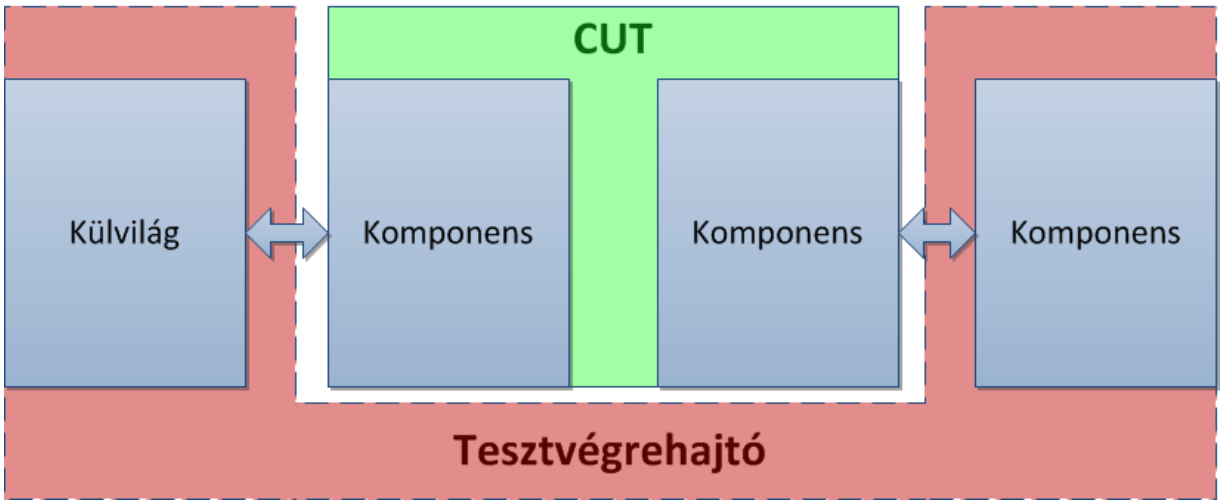
3.5.2 Komponensek tesztelése elosztott rendszerekben

A módszer kidolgozása során végig lazán csatolt elosztott rendszereket feltételeztünk, erre van felkészítve a kódgenerátor is. Ez a tesztelés szempontjából azért érdekes, mert egy ilyen rendszer esetében több komponens együttes, egy alkalmazásként való vizsgálata eredményezhet a teszt végrehajtó számára nem érzékelhető belső kommunikációt, állapotváltást. Mivel a rendszer állapotának egésze a komponensek állapotaiból tevődik össze, ezért ezek az átmenetek is a teszt kimenet részét kell, hogy képezzék. Ezért a kimenet előállítása ilyen esetekben komoly kihívás.

Ez a probléma akkor lép föl, ha az egyes automatákból származó (példányosított) komponensek között nem teszünk különbséget. Ilyenkor (mivel a komponensek nem szükségszerűen „tudnak” egymásról, egymás állapotáról) nincs egyetlen komponens sem, amit megfigyelve a rendszer egészének állapotát megismernénk. Egyéb esetekben az irányító (fölérendelt) komponensnek célszerű tisztában lennie a teljes rendszer állapotával, így ennek naplója alapján visszakövethető a komplex működés.

Ezekben az esetekben megoldást jelenthet extra megfigyelők alkalmazása: olyan plusz tesztelő komponenseké, melyek a rendszer szereplői közti kommunikációs csatornákat figyelve együttesen képesek a rendszerről naplót írni. Ennek a megoldásnak a legnagyobb előnye, hogy általánosan alkalmazható – itt nem számít, hogy a komponensek milyen alá-fölérendeltségi viszonyban állnak egymással, mindenképpen jó megoldást ad. Az ilyen megfigyelő komponensek megalkotásának nehézségét az adja, hogy nagyban függnek a kommunikációs csatornától. Egy egyszerű példával élve, egy vezetékes kommunikáción alapuló hálózat teljes figyeléséhez számos megfigyelőre lenne szükség, míg a vezeték nélküli csatornán egy alkalmasan elhelyezett vevő képes lehet lefedni a teljes területet. Ezek miatt a különbségek miatt ez a módszer körülményes, megvalósítása nehezen automatizálható.

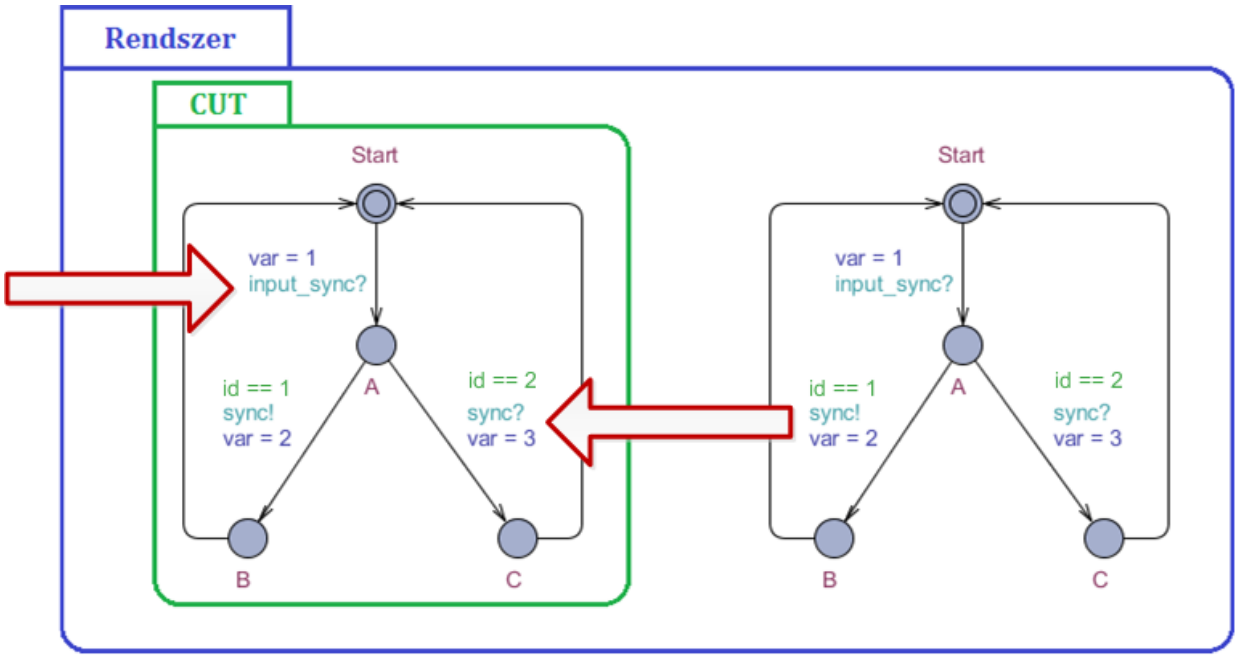
Az általunk választott megoldás olyan elosztott rendszereket biztosan támogat, ahol az egyes komponenseken futó kód megegyezik. Ez megfelel az UPPAAL leggyakoribb felhasználásának: azonos mintával (template) leírható automatákat lehet példányosítani paraméterek megadásával, és így azonos (vagy kevés különböző típusú) komponensek hálózatát kell kezelni (pl. szenzorhálózatok). A kód azonossága miatt elegendő a teljes hálózatból egyetlen komponens tesztelése, ennek reakcióiból tudunk következtetni a többi komponens viselkedésére is. A tesztek futtatása során ilyenkor ügyelni kell, hogy – bár a tesztelendő komponenst (Component Under Test, CUT) kiemeltük a környezetéből – a tesztelés során a teljes modellnek megfelelő viselkedést produkálja a CUT és a teszt végrehajtó kettőse. A CUT és a végrehajtó kapcsolatát mutatja be a 7. ábra.



7. ábra: CUT és tesztvégrehajtó kapcsolata.

A 8. ábrán látható eset egy konkrét példát mutat. Itt a komponensek külső beavatkozásra várnak, majd miután mindkét résztvevő „engedélyezve” lett, szinkron módon végrehajtják funkciójukat (ami jelen esetben egy változó értékének állítása). A

teszt futtatónak ekkor mind a külvilágtól érkező jelet (input_sync), mind pedig a kiemelt komponens által küldött üzenetet (sync) produkálnia kell.



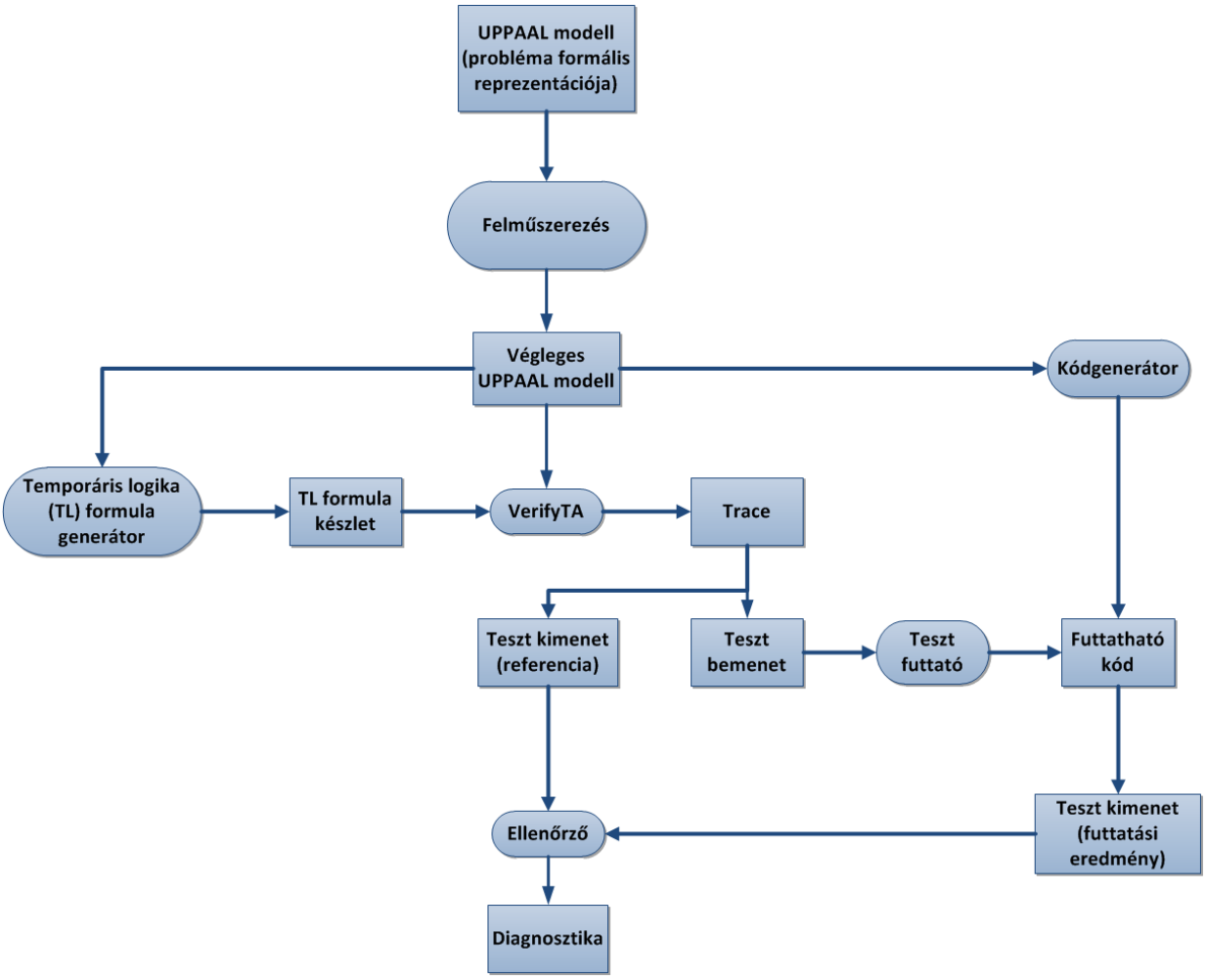
8. ábra: A teszt végrehajtó által megadandó bemenetek

A jelenlegi teszt végreható megvalósítás mellett ez az egyetlen módszer az azonos kódot futtató komponensek tesztelésének támogatására.

4 Keretrendszer a helyességellenőrzéshez

4.1 Áttekintés

A 3.3. fejezetben bemutatott módszert egy keretrendszerbe szerveztem, ennek vázlata látható a 9. ábrán.



9. ábra: UPPAAL alapú tesztelési folyamat

Az első lépés (még a tesztelési keretrendszeren kívül) a modell megalkotása, majd az alkalmazási követelmények verifikálása. A kész modellt felműszerezzük, így előkészítve a tesztgenerálásra. Fontos megjegyezni, hogy a felműszerezés során semmi olyan változtatást nem végzünk, ami a követelmények teljesülését befolyásolná, tehát a verifikáció ismét elvégezhető, és az ellenőrzéseket ezen a ponton elvégezve is ugyanazokat az eredményeket kapjuk.

A folyamat következő (párhuzamosítható) lépései a kódgenerálás valamint a tesztkészlet előállításához szükséges temporális logikai állításkészlet elkészítése. A CTL kifejezések alapján generáltatjuk az absztrakt teszteseteket. Az egyes tesztesetek generálása után van lehetőség egy tesztkészlet optimalizációs lépésre (ennek részleteit a 4.2.2.3. fejezetben ismertetem).

A kapott tesztkészletet elemenként feldolgozzuk: elkülönítjük a referencia kimenetet és a teszt futtatásához szükséges bemeneteket, majd a bemenetek sorrendjének ismeretében elő tudjuk állítani a teszt végrehajtót.

Az elkészült alkalmazást és a teszt végrehajtót használva megkapjuk az alkalmazás által generált naplót az adott tesztesetre, amit értelmezve össze tudunk vetni az absztrakt tesztesetből kinyert referencia kimenettel.

Az egyes lépésekhez tartozó implementációs kihívásokat mutatja be a 4.2. fejezet.

4.2 Az alkalmazott algoritmusok

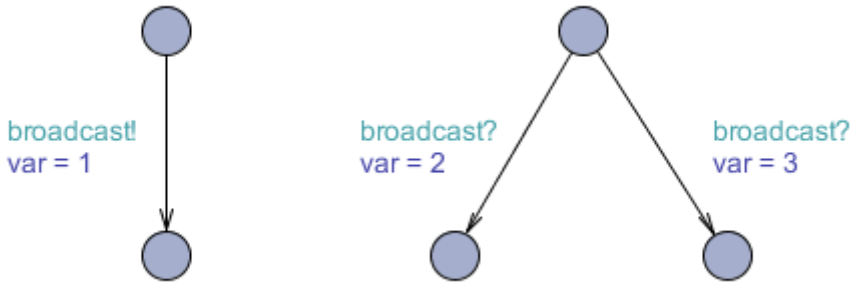
4.2.1 A modell felműszerezése

A keretrendszer első feladata az elkészült modell felműszerezése.

Az UPPAAL-ban két állapot között több átmenet is lehet, esetlegesen eltérő akciókkal, őrfeltételekkel, ezért a tesztelés során ügyelnünk kell, hogy pontosan ugyanazok az átmenetek hajtódjanak végre a verifikáció és a futtatás során. Ezért ezeket az átmeneteket meg kell különböztetnünk teszt fedettségi és végrehajtási szempontból is.

Ennek biztosítására egy rendszer szinten globális változót hozunk létre (így könnyebb a CTL kifejezéseket megfogalmazni), amelynek minden átmenet során egyedi értéket adunk. Ez a továbbiakban gyakorlatilag az átmenet azonosítójaként funkcionál. A változó értékkészletét a modellben felvett átmenetek száma határozza meg.

A felműszerezéssel kapcsolatban felmerül egy érdekes probléma. Mi történik akkor, ha egy változónak egy szinkronizáció mindkét (esetleg több) résztvevője is értéket ad (ld. 10. ábra)? Az UPPAAL ebben az esetben determinisztikus feloldást követ: a szinkronizációt kezdeményező átmenethez tartozó akciókat értékeli ki legelőször, ez után következik a vevő oldal. Üzenetszórás esetén a fogadó oldalon az átmenetek definiálásának sorrendje adja meg a kiértékelés sorrendjét is.



10. ábra: Üzenetszórás azonos változónak történő értékadással

Mit jelent mindez a gyakorlatban? Egy egyszerű szinkronizációt alapul véve látható, hogy megfigyelhetőség szempontjából elveszítjük (nem tudjuk követni) azt az átmenetet, amely a szinkronizációt kezdeményezi. Nem tudunk olyan CTL kifejezést megadni, amely esetén a változó a kívánt értéket veszi föl, hiszen a két értékadás egyetlen állapotváltás alatt történik, mégis kötött sorrendben. A probléma tovább bonyolódik üzenetszórás esetén, hiszen itt nem csak a küldő, de a fogadó oldalon is megfigyelhetetlenné válhatnak átmenetek.

A problémát úgy oldottam föl, hogy több (n darab) változót definiáltam megfigyelhetőségi célokra. Ezek postfixes elnevezési konvenciót követnek, a

szinkronizációk miatt létrejött változók az eredeti változó nevét kapják, mögötte egy sorszámmal. Mivel létrehozásuk szisztematikus és követhető (randevú típusú szinkronizációnál csatornánként egy darab plusz változó, valamint üzenetszórás esetén a fogadó élek számával megegyező plusz változó), ezért a CTL kifejezések előállításakor (ld. a következő alfejezetben) nem jelent akadályt a kezelésük.

4.2.2 Temporális logikai kifejezéskészlet előállítása

A modell felműszerezése után a következő lépés a CTL kifejezések összeállítása. A kifejezéskészletet a teszteléshez használt fedettségi kritérium határozza meg. A vezérlési struktúra szisztematikus teszteléséhez alapvető kritériumként a modell állapotainak vagy átmeneteinek tesztekkel történő lefedését szokták előírni.

4.2.2.1 Állapot alapú fedés

A választott módszer a 3.4.1. fejezetben leírtakból következik. Keresünk egy alkalmas logikai kifejezést, ami a fedettségi célt adja meg (pl. az x_i állapot elérése), majd ennek negáltját adjuk meg a modellellenőrzőnek. Ilyen módon biztosak lehetünk benne, hogy az eredmény (ha az állapot elérhető) az állítás ellenpéldával történő cáfolása lesz, vagyis éppen a fedettségi cél elérése.

Az állapot alapú fedettség vizsgálata során olyan logikai kifejezéseket fogalmazunk meg, melyek egy állapot elérését (elérhetőségét) vizsgálják. Egy ilyen logikai kifejezés lehet például az

$$\mathrm{E}\langle\rangle\ (\mathrm{P1.A})\ ,$$

ahol P1 a komponens neve, A pedig a vizsgálandó állapot neve (a 3.4.1. fejezetben leírtak szerint). Ekkor a modellellenőrzőnek megadandó negatív állítás (amire ellenpéldaként várunk egy olyan utat, ami az állapotot eléri) az A[] p = not(E<>(not p)) azonosság alapján:

$$\mathrm{A}[](!\mathrm{P1.A})$$

Ez alapján – A helyére az összes vizsgálni kívánt állapotot behelyettesítve – a teljes kifejezéskészlet tömören a következőképp írható le:

$$\{\ \mathrm{A}[](!\mathrm{P1.x_i})\ \},$$

ahol $X=\{x_0, x_1, \ldots x_n\}$ az összes vizsgálandó állapot halmaza. Az ilyen kifejezésekre kapott ellenpéldák eredményeként biztosak lehetünk benne, hogy a tesztelés során minden X-beli állapotot érinteni fogunk.

Az állapotalapú kritériumok egyik hasznos tulajdonsága, hogy az állapotfedéshez készült tesztek felhasználhatók olyan tesztesetek származtatásához, melyek segítségével a rendszer nem specifikáció szerinti (tehát a modellben nem szereplő) bemenetekre adott reakcióját vizsgálhatjuk. Nem megengedett bemenet lehet a vezérlési jel érkezése az átmenethez tartozó őrfeltétel megsértésével (pl. rossz időzítéssel), illetve olyan bemenet érkezése, melyre az adott állapotban nem szerepel reakció a modellben. Az elvárt viselkedés mindkét esetben a helyben maradás.

Az ilyen vizsgálatokhoz szükséges teszteseteket is ezen lehetőségek szerint osztályozzuk:

- Az egyszerűbb eset a modell állapotaiban nem megengedett bemenetek érkezése szerinti tesztelés. Először az adott állapotba kell vinni a tesztelt komponenst az annak fedését biztosító, fentiek szerint generált tesztesettel, majd ott a nem megengedett bemenetek megvizsgálhatók az

  $$\{\text{összes lehetséges bemenet}\} \setminus \{\text{engedélyezett bemenetek}\}$$

  halmaz szisztematikus végigpróbálásával.

- Komolyabb kihívást jelent olyan tesztek előállítása, amelyek a modell állapotaiban az őrfeltételeknek nem megfelelő esetben vizsgálják átmenetek végrehajtását. Amennyiben az őrfeltétel nem tartalmaz óraváltozót, a tesztgenerálás egy módszere lehet az állapotfedés mellett annak megkötése, hogy a vizsgált továbblépő átmenethez tartozó, modellben megadott őrfeltétel ne teljesüljön. A tesztelés során a generált teszt végrehajtása után a CUT-nak az átmenethez tartozó bemenetet megadva tudjuk vizsgálni a reakciót.

  Az óra alapú őrfeltételek vizsgálatakor egy helyes lefutás mutációjával kaphatjuk meg a kívánt tesztesetet: generálunk egy olyan tesztesetet, amelyben az adott állapot elérhető, a vizsgált továbblépő átmenet őrfeltételének nem óraváltozó alapú elemei teljesülnek, az óraváltozó pedig az engedélyezettséghez tartozó intervallum határán van. A teszt végrehajtása során az időzítést úgy változtatjuk meg, hogy éppen kikerüljön az engedélyezett intervallumból. Például ha az őrfeltétel

  $$T <= 10 \;\&\&\; T => 3$$

  alakú, akkor T==3 illetve T==10 beállításhoz generálunk egy-egy teszt esetet, majd az első esetén a várakozást csökkentve, a második esetén pedig növelve próbáljuk meg a bemenetet adni.

  Ezen tesztek elvégzéséhez további felműszerezés nem szükséges, de a használandó CTL formulák természetesen bonyolultabbak lesznek, hiszen pl. egy-egy őrfeltétel teljesülését is feltételként kell adni a vizsgált állapot elérése mellett. Az egyes állapotokból induló átmenetek tulajdonságainak (trigger bemeneteinek, őrfeltételeinek) megállapítását a modell fájl alapján végezhetjük.

4.2.2.2 Állapotátmenet alapú fedés

Az állapotok fedettségével még nem állíthatjuk, hogy minden lehetséges lefutást megvizsgáltunk, ugyanis – ahogy azt a 4.2.1. fejezetben írtam – az egyes állapotok között több átmenet is lehet, potenciálisan eltérő akciókkal, őrfeltételekkel. Ebből adódóan a szisztematikus teszteset-generálás egy alaposabb tesztelést biztosító módja az átmenet alapú fedettség vizsgálata.

Legyen a felműszerezéshez használt változó neve „__VER”, a vizsgálni kívánt átmenet azonosítója pedig 5. Ebben az esetben a keresett kifejezés:

$$A[]\ (\_\_VER!=5)$$

Általános esetben a kifejezések halmaza a következőképpen írható le:

$$\{ A[](\_\_VER!=y_i) \},$$

ahol $Y=\{y_0, y_1, \ldots y_n\}$ az átmenetek azonosítóinak halmaza. A forma tehát nagyban hasonlít az állapot alapú fedésnél használtra. Az ilyen módon létrehozott tesztkészlet szükségszerűen tartalmazza az állapot alapú fedés során kapottat.

4.2.2.3 Optimalizálás

Nagy állapotterű modell esetén a módszer szűk keresztmetszetét a modellellenőrző adja, pontosabban ennek memória- és processzoridő igénye. (A probléma annyira jellegzetes, hogy a munkaállomások korlátai miatt az UPPAAL beépítve támogatja a távoli ellenőrző szerverek használatát is.)

Emiatt felmerült az igény, hogy a tesztkészlet generáláshoz szükséges modellellenőrzési futások számát csökkentsük. A cél az, hogy a kritériumoknak megfelelő fedést az ellenőrző minél kevesebb futtatásából elérjük. A megoldáshoz kihasználjuk az átmenet alapú fedési kritériumokat, valamint az absztrakt teszteseteket értelmező modult is.

Az alapgondolat az, hogy először a modell lassabban (a kiindulási állapotból több átmeneten keresztül) elérhető átmenetein végezzük el az ellenőrzést, ekkor ugyanis a kapott nyomvonal hosszabb lesz. Ez azért előnyös, mert a kívánt átmenet fedésén túl számos másik átmenetet is lefedtünk egy ellenőrzéssel. A már lefedett átmenetek követése érdekében a keletkező absztrakt teszteseteket értelmezzük, és az azokban szereplő átmenetekre már nem futtatjuk le az ellenőrzőt.

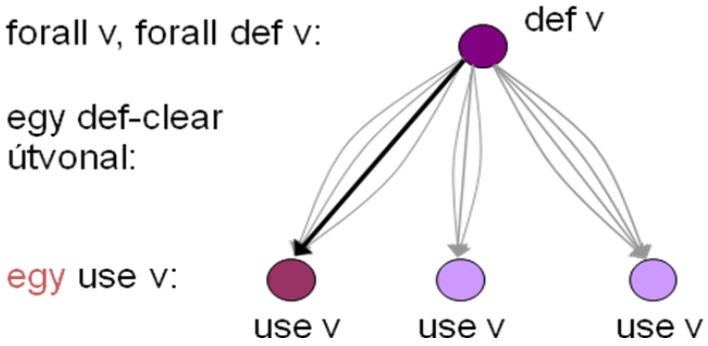
Ez a módszer valójában egyetlen módosítást igényel a fent vázolt megvalósításhoz képest. Ott a teljes tesztkészlet előállítása után következik az egyes absztrakt tesztesetek értelmezése, ez a sorrend azonban gond nélkül felcserélhető. Sajnos a módszer hatásfokára nincsen garancia: logikusan egy ilyen optimalizálásnál a nagyobb sorszámú átmenetek ellenőrzését kellene először elvégezni, ám (mivel az állapotátmeneteket a DOM fában szereplő sorrendben járom be) ez nem garantálja, hogy ez egy „távol” lévő átmenet lesz. A modellt reprezentáló XML fájlban az állapotok, átmenetek sorrendje a létrehozás sorrendjét követi, így ha a modell alkotása közben a tervező néhányszor változtat a modellen, máris felborul a sorrend. Mindazonáltal így sem lehet szükség több verifikációra, mint eredetileg volt, tehát ennek a lehetőségnek a kihasználása minden esetben ajánlott.

4.2.2.4 Adatfolyam alapú fedés

A korábban említett kétféle fedettségi kritérium tekinthető a vezérlési struktúra tesztelésének alapesetéül. Egy bonyolultabb, sokszor alkalmazott kritérium az adatfolyam alapú fedés [14]. A kritérium tárgyalásához bevezetek néhány egyszerű fogalmat.

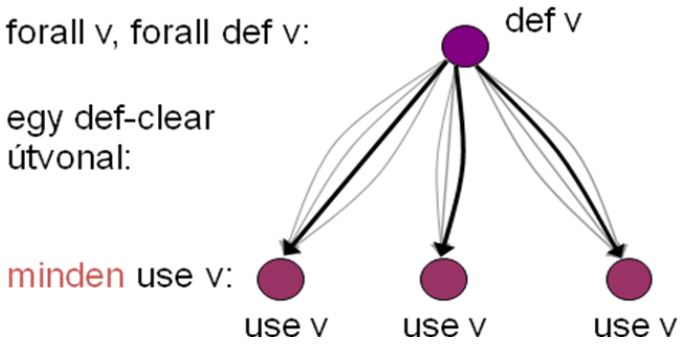
- Nevezzünk def(v)-nek minden olyan állapotátmenetet, amely (egy akció részeként) értéket ad a „v” változónak.
- Legyen use(v) az olyan állapotátmenet, amely mellékhatás-mentes kifejezésben használja föl a v változót.
- Legyen def-clear(v) egy olyan állapotátmenet-sorozat, ahol semelyik átmenet nem tartalmaz def(v)-t.

Ezen kifejezések ismeretében már definiálhatjuk a használatos adatfolyam alapú fedési kritériumokat. All-defs fedésnek nevezik az olyan fedettségi kritériumot, mely minden „v”-re, minden def(v)-hez teszttel vizsgál egy olyan def-clear(v) útvonalat, amely egy use(v)-hez vezet. Ezt szemlélteti az 11. ábra.



11. ábra: All-defs fedettségi kritérium

All-uses fedésnek hívják azt a kritériumot, amely minden „v”-re, minden def(v)-hez teszttel vizsgál egy def-clear(v) útvonalat minden use(v)-hez. Ezt mutatja be 12. ábra.



12. ábra: All-uses fedettségi kritérium

A cél tehát az, hogy a tesztkészlet előállításához ilyen típusú kritériumokat is meg tudjunk adni. A fedések definícióiból látható, hogy ennek érdekében az eddigitől eltérő felműszerezésre lesz szükség, hiszen:

- tudnunk kell azonosítani az egyes def(v) átmeneteket,
- és tudnunk kell azonosítani a use(v) átmeneteket

Egy ilyen felműszerezés elkészítése nem jelent gondot. Legyenek „DEF” és „USE” az egyes átmenetcsoportok azonosítására szolgáló változók[1]. Egyedi értéket (azonosítót) kapnak akkor, ha a tranzíció def(v) vagy use(v), megfelelően. A legkisebb kapható érték mindkét esetben 1. Ilyen definíció után megfogalmazhatjuk az all-defs fedéshez tartozó temporális logikai kifejezéskészletet:

$$\{ A[] \; !(DEF==d_i \;\&\&\; USE!=0) \},$$

[1] Ezek minden „v” változó esetén újradefiniálandók (vagy valamilyen elnevezési szisztéma szerint a teljes felműszerezés egyszerre is elvégezhető). Az érthetőség kedvéért a formulákat egy változó fedésére adtam meg.

ahol D={d_0, d_1, … d_n} a def(v) értékeinek halmaza (ebben az esetben a d_i azonosítójú def(v) élhez keresünk példát). Csak akkor kapunk ellenpéldát, amennyiben létezik átmenetsorozat a vizsgált DEF és valamelyik USE átmenetek között, egyébként a verifikáció sikeres lesz. Az all-uses fedés is megadható egyszerűen:

$$\{ A[] \; !(DEF==d_i \; \&\& \; USE==u_j) \},$$

ha változatlan D mellett U={u_0, u_1, …, u_n} jelöli a felvehető értékek halmazát USE-ra.

Természetesen a fenti kifejezések csak egy változóra (v) végzik el a vizsgálatot, a teljes fedéshez az aktuális formulát minden változó minden lehetséges értékére el kell végezni:

$$\begin{aligned} &\text{forall v:} \\ &\quad \text{forall } d_i, u_j: \\ &\qquad A[] \; !(\ldots) \end{aligned}$$

Ezzel a kifejezéssel gyakorlatilag az összes def(v)-use(v) páros összehasonlítását elvégzem, ami nagyon nagy kifejezéskészletet eredményez. Absztrakt teszteset azonban csak a kifejezések egy töredékéből készül (a többi esetben valóban nincs út DEF és USE között, így sikeres lesz a verifikáció – ugyan úgy, mint az all-defs fedés esetében). Azonban a 4.2.2.3. fejezet alapján ez nem egy szerencsés megoldás, hiszen a fölösleges kifejezések kiértékelése nagyon lelassíthatja a teszteset-generálás folyamatát.

Ennek elkerülése érdekében kidolgoztam egy formulát, amely megfogalmazásában ugyan bonyolultabb, mint az előzőek, viszont az ellenőrzés módszere sokkal letisztultabb. Az alapvető célom a def-clear útvonalak felismerése volt.

Definiálok még két változót, legyenek ezek „flag” és „SmartUSE”. Legyen flag=1 a use(v) átmeneteken, és legyen flag=0 azokon az átmeneteken, melyek se nem use(v)-k, se nem def(v)-k. SmartUSE értéke legyen (1) SmartUSE=1 a def(v) tranzíciókon, (2) SmartUSE+=2 a use(v) tranzíciókon és (3) SmartUSE=SmartUSE egyéb (se nem def(v), se nem use(v) átmenetek) esetekben.

Ezek a változók pusztán a tesztesetek előállításához szükségesek, a tesztek futtatásához és a verifikációhoz természetesen elegendőek az azonosításra szolgáló változók.

Az így felműszerezett modellre először fogalmazzuk meg az All-defs fedés CTL logikai kifejezéskészletét:

$$A[] \; !(flag \; != \; 0 \; \&\& \; SmartUSE\%2==1 \; \&\& \; SmartUSE!=1 \; \&\& \; DEF==d_i),$$

A teljes kifejezés megértését segítheti az egyes elemek feloldása:

- SmartUSE modulo 2 azért szükségszerűen 1, mert a számunkra kedvező élsorozatok esetében történik egy SmartUSE=1 értékadás (a def(v) átmeneten), majd minden egyes use(v) ezt növeli kettővel. Ha SmartUSE modulo 2 nulla, akkor úgy ment végbe egy use(v) átmenet, hogy nem előzte meg def(v), tehát a feladat szempontjából érdektelen.

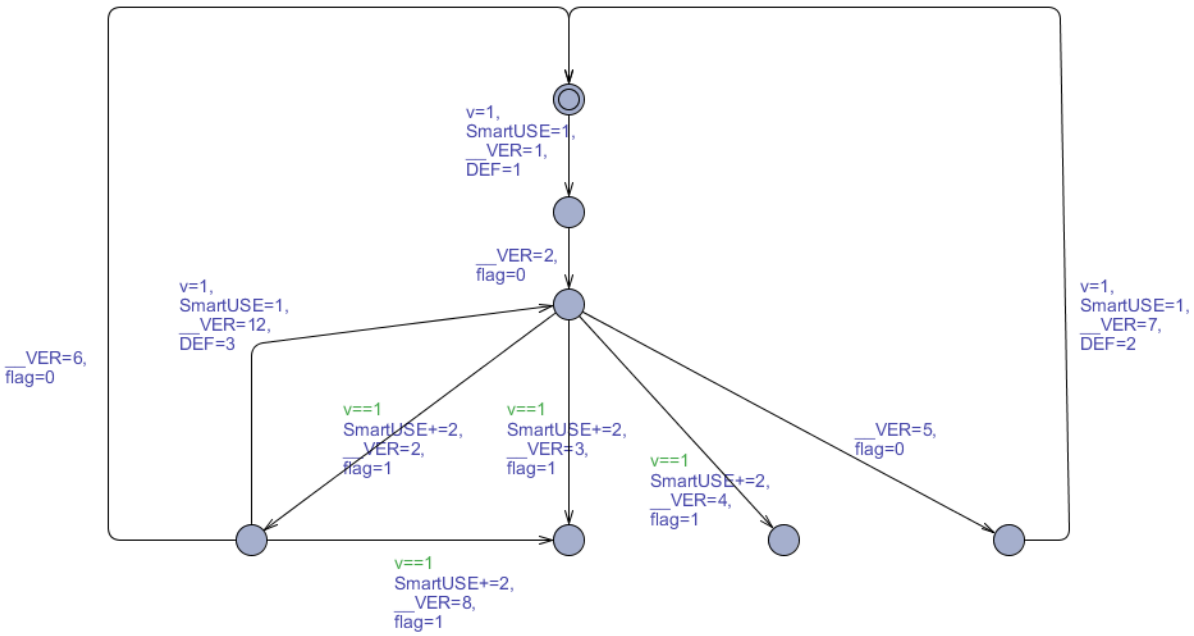
- A SmartUSE!=1 kifejezéssel a d_i azonosítójú def(v) él végállapotát zárjuk ki (ebben az állapotban – közvetlenül az értékadás után – ugyanis természetesen 1 lenne a változó értéke, pedig még nem értünk el use(v) átmenetet).
- A def-clear átmenetsorozat biztosítására szolgál a DEF==d_i (ezt az értéket egyszer kaphatja, ez az átmenet pedig pont a kívánt út elejét jelenti), hiszen a def-clear tulajdonságot pontosan az tudja elrontani, ha DEF új értéket kap (tehát egy def(v) tranzíció).
- A flag használata azért szükséges, hogy különbséget tudjunk tenni a use(v) és az „érdektelen” átmenetek között – SmartUSE modulo 2 akkor is lehetne 1, ha egy use(v) tranzíciót egy olyan követ, amely se nem def(v), se nem use(v), ezekre az esetekre pedig nem vagyunk kíváncsiak.

Az all-uses fedést is hasonlóképpen fogalmazhatjuk meg. Az adott def(v)-ből elérhető összes use(v) megtalálása érdekében kihasználom ugyan azt a módszert, ami az állapot és az állapotátmenet alapú fedettség ellenőrzésénél még csak optimalizációs célt szolgált. A konkrét kifejezés az alábbi formában írható föl:

$$A[]\ !(flag\ !=\ 0\ \&\&\ SmartUSE\%2==1\ \&\&\ SmartUSE!=1\ \&\&\ DEF==d_i\ \&\&\ \_\_VER!=\{V\})$$

Az all-defs fedés formuláját annyival egészítem ki, hogy – az élek azonosítóját felhasználva – a már megtalált use(v)-ket folyamatosan kiemelem a lehetséges ellenpéldák halmazából. Ezt úgy tudom megtenni, hogy először verifikálom az all-defs fedés formuláját, majd a naplóból kiolvasom, melyik __VER értékű use(v)-t kaptam ellenpéldának. Ez után ezt a tranzíciót kizárom a következő vizsgálatból, a fenti formula segítségével. Mindezt addig kell ismételni, amíg a kifejezés kiértékelése igaz nem lesz – ekkor megkaptuk a d_i def(v) átmenethez tartozó all-uses fedést.

Egy példa modellt mutat a 13. ábra:



13. ábra: Példa „v” értékének vizsgálatához felműszerezett modellre.

Ebben az esetben a DEF=1 def(v) átmenet vizsgálatához a kezdeti kifejezés a következő lehet:

$$A[]\ !(flag!=0\ \&\&\ SmartUSE\%2==1\ \&\&\ SmartUSE!=1\ \&\&\ DEF==1)$$

Több iteráció (verifikáció, majd az ellenpéldaként kapott él kizárása) után pedig eljutunk a végső formulára:

$$A[]\ !(flag!=0\ \&\&\ SmartUSE\%2==1\ \&\&\ SmartUSE!=1\ \&\&\ DEF==1\ \&\&\ \_\_VER!=4\ \&\&\ \_\_VER!=3\ \&\&\ \_\_VER!=2\ \&\&\ \_\_VER!=8)$$

Ez a kifejezés már teljesül. A lépések között pedig megkaptunk olyan teszteseteket, amik a {2, 3, 4, 8} __VER értékű élek vizsgálatára szolgálnak, így DEF=1 esetre egy all-use(v) fedést adnak.

Az utóbbi két formula használata esetén nincs szükség fölösleges verifikációra, a plusz változókkal való kiegészítés pedig sokkal kisebb overheaddel jár – tehát javasolt ezeket az „okos” formulákat alkalmazni.

Természetesen a szinkronizációkból fakadó értékadási probléma az újonnan bevezetett változóknál is jelen van, a megoldás itt is a sorszámozás.

4.2.3 Absztrakt tesztesetek leképezése

A kifejezés kiértékelése után kapott napló általános esetben a következő rövid példához hasonló formátumot követ:

```
4
1
.
.
0
0
0
.
4
0
.
.
0
5
1
(...)
```

Egyszerűbb esetekben ez a reprezentáció is értelmezhető, ám a *verifyta* (az UPPAAL modellellenőrzője) lehetőséget ad a naplók olvashatóbb formában (szimbolikus napló, symbolic trace) való megjelenítésére is. Ehhez a modellellenőrző parancssori felületét kell használnunk. Legyen obsv.xml a modellt tartalmazó állomány, és jelölje query.q a CTL kifejezéseket tartalmazó fájlt. Ekkor az utasítás a következő:

```
verifyta.exe –Y –t0 obsv.xml query.q
```

Az így kapott, olvashatóbb formátumban a napló látható alább:

```
STATE
```

```
( P0.START P1.START PI._ID3 )
P0.X<=3, P0.X-P1.X<=0, P1.X<=3, P1.X-P0.X<=0 B=0 A=0 P0.ID=0 P0.A=0 P1.ID=1
P1.A=0

TRANSITIONS:
 PI._ID3->PI._ID3 { 1, INPUT!, 1 }
 P1.START->P1.A { ID == 1, INPUT?, A := 1 }

STATE

( P0.START P1.A PI._ID3 )
P0.X<=3, P0.X-P1.X<=0, P1.X<=3, P1.X-P0.X<=0 B=0 A=1 P0.ID=0 P0.A=0 P1.ID=1
P1.A=0

TRANSITIONS:
 P1.A->P1.B { 1, SYN!, A := 2, X := 0 }
 P0.START->P0.START { 1, SYN?, A++, B := 1, X := 0 }

STATE

( P0.START P1.B PI._ID3 )
P0.X-P1.X<=0, P1.X-P0.X<=0 B=1 A=2 P0.ID=0 P0.A=1 P1.ID=1 P1.A=0
```

Ez a forma sokkal könnyebb feldolgozási lehetőséget biztosít. A nyers kimenettel szemben a változókra nevükkel hivatkozik, a rendszer állapotát egyszerűen, az egyes komponensek állapotainak halmazával írja le, stb.

A szimbolikus napló állapotleírásának a szintaktikája:

State

(([Processz neve] . [állapot neve])*)
(([Processz neve]? . [változó neve]) = változó értéke)*

Az órák értékének definiálása a közöttük értelmezhető relációk segítségével történik [3].

Ezt a formátumot értelmezve a kapott naplón, megkapjuk a bejárt állapotokat, és a változók felvett értékeit. Az átmenetek leírásának formátuma pedig a következő:

Transitions:

([Processz neve].[kiinduló állapot neve]->[Processz neve].[végső állapot neve]{őrfeltételek, szinkronizáció, értékadások})*

Ez a pár sor leírja, hogy a rendszer egyes komponensei melyik állapotból melyikbe kerülnek az átmenet végrehajtása után. Amennyiben történik szinkronizáció, nem egy ilyen sor található, hiszen ekkor több komponens lép egyszerre: hagyományos csatorna esetén kettő, üzenetszórás esetén pedig bárhány résztvevő válthat állapotot. Fontos megjegyezni, hogy az absztrakt tesztesetből (napló) kiderül a kommunikáció iránya is: az UPPAAL szintaktikájának megfelelően „?” jelöli az üzenet fogadását és „!” az üzenet küldését.

Ezek után egyértelműen azonosíthatóak az egyes állapotátmenetek, és a hozzájuk tartozó bemenetek, kommunikáció is. Az elnevezési konvenciók (ld. 2.2.2. fejezet) ismeretében a felhasználói beavatkozásokat, a külvilág felé való kommunikációt is fel lehet ismerni. Ezek megkeresésével könnyen kinyerhetőek a konkrét tesztesetek előállításához szükséges adatok:

- A CUT által felvett állapotok listája,
- A CUT által végrehajtott tranzíciók listája,
- A tranzíciók végrehajtásához szükséges bemenetek listája.

Szintén könnyen kinyerhető az elvárt kifelé történő kommunikációs üzenetsor.

4.2.3.1 Felhasználói bemenetek

A külvilággal való interakciók kezelése nem ér véget ott, hogy megtaláljuk helyüket az egyes tesztesetekben. Mivel ezek közvetlenül a külvilággal történő kapcsolatot valósítják meg (pl. gombnyomás érzékelése), ezért külön szót érdemel az automata tesztelésük megvalósítása.

Ezek a szolgáltatások a szokványos üzenetküldéstől független platformszolgáltatásként vannak implementálva, így felmerül a kérdés, hogy ebben az esetben hogyan lehet az automatikus tesztelést megvalósítani (hogyan tudunk gombnyomást szimulálni)?

A megoldás a modellezésből és a kódgenerátor specifikációjából következik. Modell szintjén az elnevezési konvenciók követésén kívül semmi nem különbözteti meg az ilyen kommunikációt, tehát ezekben az esetekben is ugyanúgy figyel a CUT az adott csatornára, mintha az üzenet a másik komponenstől származna. Ez igaz marad a kódgenerálás után is.

Vétel esetén tehát a bemenetek kezeléséhez kapcsolódó függvény implementációja tartalmazza a (példánál maradva) gombnyomás figyelését, valamint gombnyomás esetén üzenet küldését a szokványos kommunikációs alrendszeren keresztül. Ez egy nem blokkoló üzenet[2], melynek címzettje az adott komponens, tartalma pedig a gomb állapota.

Kifelé történő kommunikációnál a helyzet hasonló: itt a platformszolgáltatás feladata az adott csatornán való figyelés, majd a kapott üzenet továbbítása a megfelelő interfészen.

Az ilyen üzeneteket azonban tudja szimulálni a teszt végrehajtó is. Így tehát az automata tesztelés során a felhasználói kommunikáció egyszerűen helyettesíthető szokványos üzenetküldéssel.

Amennyiben a platformszolgáltatásokat teljes körűen a tesztelés hatókörébe szeretnénk vonni, az esetben természetesen a közvetlen interakciókat azok fizikai felületén kell elvégezni (pl. a gombokat megnyomni). Ez manuális tesztelési lépéseket is igényelhet, amit a fent leírt, korlátozottabb hatókörű automatizált tesztelés után, csak a releváns tesztesetekre szorítkozva kell elvégezni.

4.2.3.2 Óraváltozók kezelése

Az óraváltozók használatának lehetősége az UPPAAL egyik nagy előnye, ezért ezek kezelésére figyelmet fordítottunk a kódgenerátor megalkotásakor is. A megvalósítás ebben az esetben is platformfüggő, ezért külön illesztjük a kódhoz.

[2] Így illeszkedik a működés pontosan a modellhez: nem feltétlenül figyelünk minden állapotban a bemenetekre, ilyenkor elveszhet az üzenet.

A tesztek megalkotása és futtatása szempontjából ezek semmilyen fennakadást nem jelentenek – az alkalmazások nagy részében ezek a változók az egyes komponensek belső működésének részei maradnak, a teszteseteknek való megfelelés szempontjából csak a naplóba kerülő érték számít. Az órákhoz köthető (fejlesztő által implementált, majd a generált kódhoz csatolt) függvények arra szolgálnak, hogy megadják mennyi az adott platformon az egységnyi idő. Tehát a kimeneten, és a kommunikáció folyamán ugyanúgy egész értékeket kezelünk (hasonló módon működik a 3.1. fejezetben bemutatott TRON eszköz is).

Az egyetlen kihívás abból adódik, ha a komponensek közötti kommunikáció időbeli keretekhez van kötve (pl. egy őrfeltételen keresztül). Ennek kezelését a következőekben tárgyaljuk.

4.2.4 Teszt végrehajtás

A teszt végrehajtó modul feladata, hogy az absztrakt tesztesetekből (az előző fejezetben leírt módon) kinyert információk alapján a kívánt működéshez szükséges bemenetekkel ellássa a vizsgált komponenst.

A megvalósítás nehézsége, hogy a kommunikációhoz – mivel platformfüggő alrendszerről van szó – ugyanazt az implementációt kell használnunk, amit a generált kódhoz is illesztünk. Erre azért van szükség, hogy kizárhassuk a kommunikációs csatornára vonatkozó megkötéseket, így szabadon tesztelhessünk bármilyen platformon – ez megfelel eredeti célkitűzéseimnek. A megoldásom hátránya, hogy így legalább még egy futtató platformnak rendelkezésre kell állni a teszteléshez.

A teszt végrehajtó modul önmagában egyszerű. A program a megadott sorrendben próbálkozik az üzenetküldéssel úgy, hogy az egyes üzeneteket esetlegesen többször egymás után is elküldi, amíg a kommunikációs protokoll (ld. 2.2.3.) szerint visszajelzést nem kap a sikeres üzenetküldésről.

Az absztrakt tesztesetek a kiadandó bemenetek között késleltetéseket is tartalmazhatnak, amennyiben ilyenek a vizsgált komponens viselkedésének teszteléséhez szükségesek (időfüggő viselkedést tartalmaz a komponens). Természetesen a teszt végrehajtónak a bemenetek kiadása során ezeket be kell tartania. A megvalósítást illetően ez annyit jelent, hogy az egyes üzenetküldések között a minimálisan szükséges időtartam erejéig várakozik.

A teszt végrehajtó továbbá két üzenet küldése között figyel az összes lehetséges kommunikációs csatornán, hiszen a CUT is kezdeményezhet kommunikációt. A sikeresen küldött és fogadott üzenetek naplózásra kerülnek, így megkapjuk a vizsgált komponens és a zárt világ többi része közötti összes kommunikációt.

A megvalósítást tekintve a teszt végrehajtó egy C nyelvű kódból áll, mely hivatkozik a kódgenerátorban is használt függvényekre – a platformok egyezése esetén bátran használhatjuk ezeket.

4.2.5 Tesztek értékelése

Az absztrakt tesztesetek értelmezésével megkaptuk az elvárt kimeneteket. A tényleges kimenetet pedig a futás közben történő naplózás eredményeképp kapjuk.

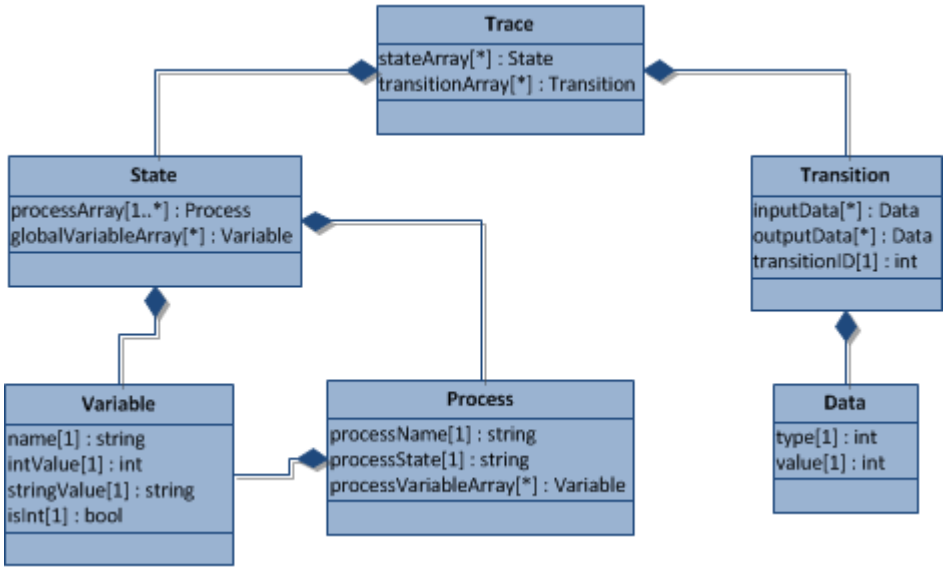
A komponenseken futó kódba a kódgenerátor kiterjesztéseként illeszthetünk naplózó funkciót, ami a platformszolgáltatások illesztéséhez hasonló technikával történik. A

naplózás történhet online vagy offline módon: online esetben az alkalmazás egy kimeneti csatornára rögzíti az adatokat, offline módban pedig egy lokális állományban tárolja a naplót.

A dolgozat készítése során módosítottam a kódgenerátorhoz elkészült naplózó kiegészítést a könnyebb bővíthetőség érdekében. A demonstrációs célú állapotazonosító helyett jelenleg a végrehajtott átmenetek azonosítói kerülnek a naplóba.

A napló segítségével megkaptuk a tényleges lefutás menetét, így a továbbiakban a feladatunk ennek az absztrakt tesztesetben megadott lefutással való összehasonlítása.

Az összehasonlítás támogatására a két naplóból egy-egy belső Java reprezentációt készítek. A struktúra, melynek felépítését szemlélteti a 14. ábra, általánosan leírja egy napló tartalmát, tehát univerzálisan használható. Ha esetleg a naplózás kimeneti formátuma változna, csak egy új parser-t kell implementálni (vagy egy korábbit módosítani). A tárolt adatok sokkal nagyobb rugalmasságot adnak a későbbi elemzéshez.



14. ábra: Adattárolási struktúra

A struktúra egyes elemei egyszerűen megfeleltethetők az UPPAAL napló egyes részeinek. Egy „Trace” tartalmaz egy teszteset tetszőleges irányához kapcsolódóan minden szükséges információt. Az átmenetekben (Transition) elkülönítve tárolom a teszt végrehajtásához szükséges bemeneteket, és az elvárt kimeneteket. Az állapotok esetében pedig eltárolok minden értéket, ami a naplóból kiolvasható (tehát kimeneti napló olvasásakor ide kerülnek a megfigyelt értékek).

A kiértékelés tehát egy Trace állapotainak egy másik Trace átmeneteinek kimenetével történő összehasonlítását jelenti.

Ennek megvalósítására egy praktikus módszer a viselkedési relációk használata. Ez a modellalapú szoftvertesztelésben elterjedt módszer: a relációk segítségével precízen meghatározható, hogy a modellhez képest milyen viselkedésbeli eltérést tekintünk megengedhetőnek (pl. „átmegy-e a teszten” a tesztelt komponens, ha a modellben meghatározott viselkedéshez képest több kimenetet ad, pl. extra üzeneteket küld). Az ilyen típusú tesztelési relációk alapjául a modellel (és a hozzá tartozó lefutásokkal)

adott specifikáció, és a megfigyelhető viselkedés által definiált kimeneti modell szolgál. A megfigyelhető és a tényleges (belső) működés közti különbséget a modellhez definiált interfészek határozzák meg (alapesetben az nem megfigyelhető, ami nem jelenik meg a tesztelt komponens interfészein). A megfigyelhető viselkedésre határozunk meg ekvivalencia illetve finomítási relációkat. Ekvivalencia relációról akkor beszélünk, ha az reflexív, tranzitív és szimmetrikus, míg a finomítási reláció reflexiv, tranzitív és antiszimmetrikus.

Amennyiben egy definiált ekvivalencia reláció nem áll fönn, további vizsgálatok lehetségesek az eltérés irányának meghatározására, a finomítási relációk segítségével. Ezek vizsgálata praktikus, hiszen a modelltől való eltérés nem szükségszerűen jelent problémát: ha az implementáció során megszűnik a modellben jelen lévő nemdeterminizmus, már nem áll fönn az ekvivalencia reláció, viszont ez nem jelent olyan korlátozást, ami miatt ne lenne használható az alkalmazás kritikus rendszerekben. További példa lehet a holtpontok számának csökkenése, de természetesen az eltérés részeként az is lehet, hogy a viselkedés bővül (nem várt kimenet jelenik meg). Finomítási relációkkal tehát tudjuk vizsgálni modell és modell között az eltérések irányát. A tesztelésekhez használt konformancia relációknak hasonló a célja.

4.2.5.1 A k-ekvivalencia reláció

A relációk közül a legegyszerűbb a k-ekvivalencia reláció, az absztrakt teszteset hosszát tekintve k lépésnek: ekkor ugyanis elég a teszt végrehajtó modul egy-egy teszthez tartozó naplóját összevetnünk az absztrakt teszteset megfelelő részeivel. A reláció akkor áll fönn, ha az absztrakt tesztesetben szereplő végrehajtási szekvencia és a hozzá tartozó feljegyzett végrehajtási szekvencia (üzenetek, átmenetek) tartalmukban és sorrendjükben is egyezőek, minden tesztesetre.

4.2.5.2 Az IOCO reláció

A meglehetősen egyszerű, de szigorú kritériumot alkalmazó k-ekvivalencia relación kívül más relációk szerinti vizsgálatok is megvalósíthatók a tesztelési keretrendszerben.

A „fekete doboz” alapú teszteléshez szükséges a bemenetek (vezérelhető akciók) és a kimenetek (megfigyelhető akciók), valamint a belső (nem megfigyelhető) akciók közötti különbségtétel. Így adódik az LTS formalizmus kiterjesztéseként az IOLTS (Input-Output Labeled Transition System) formalizmus: az állapotátmenetekhez rendelhető akciókat bemeneti, kimeneti és belső akciókra kell felosztanunk. A konformancia relációknál a különbségtétel alapja lehet az is, ha egy elvárt kimeneti akció nem hajtódik végre.

A lehetőségek közül a 3.1. fejezetben már említett IOCO relációt [17] emelném ki.

Az IOCO formális definícióját a következőképpen fogalmazhatjuk meg: minden, a specifikációban felvehető megfigyelhető akciószekvenciára igaz, hogy az így elérhető állapotokban az implementáció által nyújtott kimeneti akciók részhalmazát képezik a specifikáció által nyújtott kimeneti akcióknak.

Informálisan azt mondhatjuk, hogy az IOCO reláció teljesül, ha az implementáció „befér” a specifikáció által meghatározott keretek közé – tehát a specifikáció a megengedhető viselkedés kereteit rögzíti.

A reláció megenged szűkített és bővített viselkedést is: a specifikációhoz képest a kimeneten megjelenhet kevesebb akció is, viszont a specifikációban nem szereplő akciószekvenciákra az implementációban plusz kimenet is megjelenhet. A szűkítés fakadhat részleges implementációból vagy a nemdetermisztikus működés egyféle feloldásából, míg plusz kimenetek megjelenése csak nem teljes specifikáció esetén fordulhat elő.

Ezek alapján tehát az IOCO egy praktikus reláció kimenetekkel és bemenetekkel rendelkező rendszerek fekete doboz alapú tesztelésére. A reláció fennállása esetén biztosak lehetünk benne, hogy a specifikált működés kereteit az alkalmazás nem lépi túl, viszont támogatja a részleges megvalósításokat is.

4.2.5.3 Modellek és komponensek konformanciájának általános vizsgálata

Az előző fejezetek során sokszor hivatkoztam a kódgenerátor adottságaira. Felmerül a kérdés, hogy a módszer alkalmazható-e általánosan, tetszőleges alkalmazás és UPPAAL modell összevetésére?

A keretrendszer tervezésénél a programmal szemben támasztott feltételezések alapját a kódgenerátor adta, ám ez csak az általunk definiált interfészek és belső naplózás meghatározásában játszott közre. Amennyiben egy alkalmazás (kézzel írt vagy készen kapott) megfelel a megfigyelhetőségi kritériumoknak, és a használandó platformszolgáltatások rendelkezésre állnak nincs akadálya a tesztelésnek.

Amennyiben a felműszerezés nem lehetséges (pl. nem hozzáférhető a forráskód), akkor a viselkedési konformancia vizsgálatának a bemenetek és az elsődleges kimenetek megfigyelésére kell korlátozódnia.

4.3 Automatikus eszközök

A dolgozat célja egy teljesen automatikus ellenőrző megalkotása volt, így a fejezetben bemutatott modulok kidolgozása során ügyeltem, hogy az automatizálás lehetséges legyen. Az egyes modulok esetén ez a következő technológiák használatát jelentette:

- A modell felműszerezése: Az UPPAAL a modelleket egy viszonylag kötött sémára [10] illeszkedő XML fájlban tárolja. Az XML fájlok kezelésére számos módszer van, mivel az UPPAAL fájljai jellemzően kis méretűek, ezért kezelésükhöz egy DOM[3] [11] fát hozok létre a fájlból. A módosításokat (változó hozzáadása, értékadási akció minden átmeneten) ezen a fa struktúrán végzem, majd a folyamat befejeztével visszavezetem őket a fájlba. Ezen folyamat semmilyen beavatkozást nem igényel.
- A temporális logikai kifejezéskészlet előállítása: Ennek során az egyes kifejezéseket fájlokba írom. A vizsgálandó változó értékkészlete ismert, megegyezik a modellben található állapotátmenetek számával. Mivel a felműszerezés során már a keretrendszer adta hozzá a modellhez az értékadásokat, ezért a maximális értéket akár el is tárolhatjuk (jelenleg a

[3] Document Object Model

program így működik). Az elkészült fájlok innentől szabadon megadhatók az UPPAAL modellellenőrzőjének bemeneteként. A verifyta egymás után történő futtatása szintén könnyedén automatizálható, vagyis a tesztkészlet automatikus előállítása nem jelent gondot.

- Az absztrakt tesztesetek értelmezése: A 4.2.5. fejezetben tárgyalt adatstruktúra automatikus felépítése nem igényel specifikus technológiát, egyszerűen megvalósítható.
- A tesztek végrehajtásának automatizálása: a teszt végrehajtó modul célplatformra való fordítása és telepítése mindenképpen kézi művelet. A végrehajtáshoz a teszteseteket (a tesztekhez szükséges a küldendő üzeneteket és azok sorrendjét) meghatározott formában meg kell adni teszt végrehajtónak. A kimenetként keletkező napló értelmezése is teljes egészében automatizálható (hasonlóan az absztrakt tesztesetek értelmezéséhez).
- A tesztek értékelése: A kiértékelés bonyolultsága függ a vizsgálni kívánt relációtól. A k-ekvivalencia ellenőrzése a korábban leírtaknak megfelelően egyszerű, automatizálható. Az általános relációk kezeléséről a 4.2.5. fejezetben szóltunk.

Ezekkel a modulokkal tehát a lehetőséget megadtuk a teljesen automatikus tesztgenerálásra és futtatásra, így a dolgozat elején kitűzött fő célt teljesítettem.

5 Megvalósítás és mintapéldák

5.1 Az eszközök ismertetése

A példák előtt tekintsük át röviden, hogy a 4. fejezetben részletesen bemutatott eszközök jelenleg milyen formában vannak megvalósítva:

- Felműszerező modul: Bemenete a kézzel készített és verifikált modell, kimenete pedig egy másik modell, amely jelenleg az átmenetek megfigyelhetőségét biztosító változókkal van kiegészítve. A megvalósítás Java alapú, és teljesen automatizált.
- TL formula készítő modul: A bővített modell alapján elkészíti az ellenőrzendő CTL logikai kifejezéseket. Kimenetként ezen kifejezéseket tartalmazó szöveges fájlokat kapunk. Ez is Java alapon működik, az előző egységgel szorosan összekötve. A használható fedettségi kritériumok korlátját a felműszerezés jelenti: jelenleg nem tud az eszköz adatfolyam alapú kritériumok szerint tesztkészletet automatikusan előállítani.
- Modellellenőrző modul: A formula készletet bemenetként használjuk egy külső eszköz, az UPPAAL modellellenőrzője számára. Kimenetként absztrakt tesztesetként használható futási naplókat kapunk. Ez a modul tehát az UPPAAL használatából fakadóan rendelkezésre áll.
- Értelmező: Bemenetéül a futási naplók szolgálnak. Ez a modul kezeli az absztrakt teszteseteket és a tényleges futás eredményeként kapott naplókat egyaránt. Ennek eredménye egy belső reprezentáció, amiben a tárolt adatokon könnyebb műveleteket végezni. Kimenetként megjelenik az egyes lefutások reprodukálásához szükséges bemeneti adatsor.
- Teszt végrehajtó: Az értelmezőtől kapott bemeneteket megfelelő sorrendben és időben megadja a CUT-nak. Ezt a modult jelenleg még kézzel kell vezérelni, az automatizálás nem volt szükséges a „proof of concept” megvalósításhoz.
- Konformancia vizsgáló modul: A saját ellenőrzőm az értelmezővel összefonódva működik. Az adatok megfelelő elemeit összehasonlítva k-ekvivalenciát állapít meg a modell és a futó kód között.

A fejezet további részében egy példán keresztül demonstrálom a keretrendszer működését.

5.2 Mintapéldák

5.2.1 Demonstráció szimulátorral

A kódgenerátorral párhuzamosan, az előző TDK munka eredményeként elkészült egy Mitmót szimulátor is.

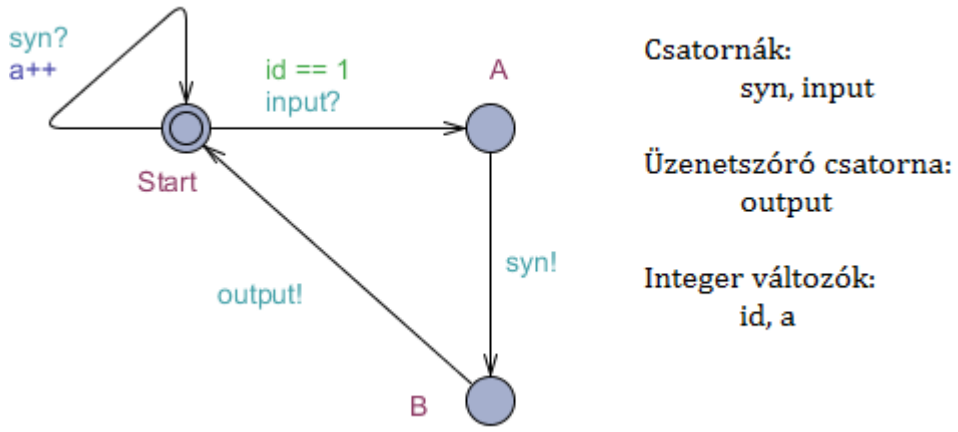
A Mitmót a MIT tanszéken fejlesztett moduláris beágyazott rendszer. A modulok között a 8 vagy 32 bites processzor modulok mellett található ki- és bemenetkezelő, valamint rádiós modul is, ezek használatával több Mitmót segítségével jól

demonstrálható a kódgenerátor alkalmazási környezete (lazán csatolt elosztott rendszer). A Mitmót szimulátor a fejlesztést segíti, mivel az elosztott alkalmazás kódja a szimulátorban még a tényleges telepítés előtt kipróbálható.

A szimulátoron tetszőleges számú Mitmót hozható létre, ezeken a kódgenerátor által készített kód fut. A platformszolgáltatásokat természetesen itt a „szimulátor platformhoz" külön kellett implementálni, ehhez az SDL [12] könyvtárat használtuk. Így természetesen a szimulációs vizsgálat nem a konkrét fizikai platformon való működést, hanem csak az alkalmazás kódjának logikai helyességét vizsgálja.

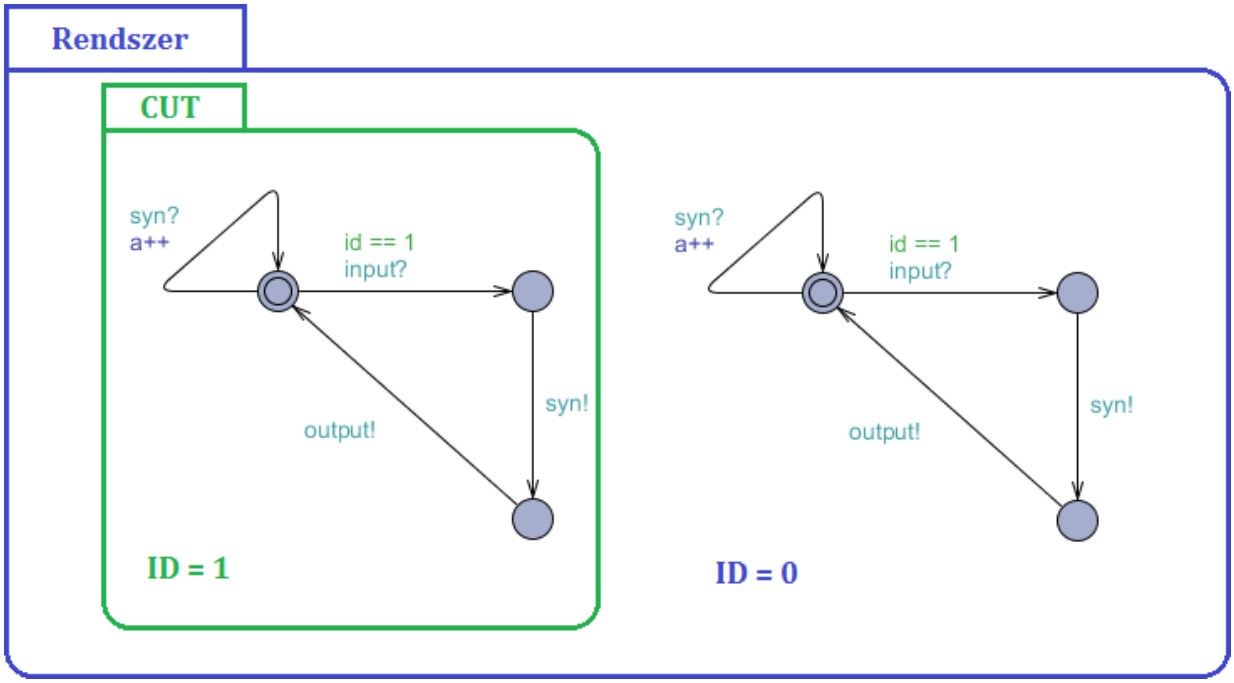
Az eszköz magját a csatorna szerver képezi, ennek az egységnek a feladata a komponensek közti üzenetek továbbítása. A tesztelő keretrendszer fejlesztése során a teszt végrehajtó ellenőrzéséhez is ezt a szervert használtam. Így a kódgenerátor és a tesztelő keretrendszer működése is megoldható volt PC platformon (Windows környezetben).

Példának a 15. ábrán látható modellt választottam, amely két komponensre van tervezve. Ez a modell egy nagyon egyszerű távirányítót ír le: gombnyomás (felhasználói bemenet) után a jelzést megpróbálja továbbítani az eszköz, és ha sikerül, visszajelzést ad a felhasználó felé (output szinkronizáció). A gombnyomás hatására egy változó értéke növekszik.



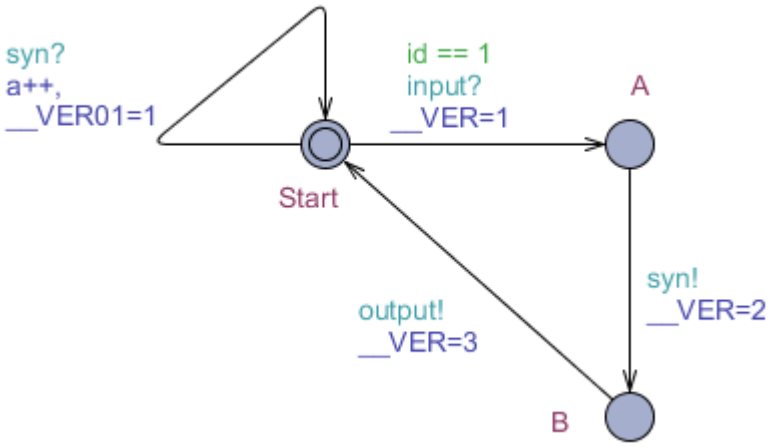
15. ábra: Demonstrációs mintapélda

Mivel két komponensből áll a teljes rendszer (ezek közül a bonyolultabb működésűt fogjuk vizsgálni), ezért a CUT és a rendszer kapcsolata a 16. ábrán látható módon rajzolható föl.



16. ábra: A rendszer és a CUT modelljei a mintapéldában

A modell kívánt szempontok szerinti verifikációja után[4] elkezdődhet a tesztelési folyamat. Ennek első lépéseként elkészíttetem a felműszerezést, melynek eredménye a 17. ábrán látható:



17. ábra: A mintamodell felműszerezés után

Ez után következett a tesztkészlet előállítása, amely 4 fájlt eredményezett. Ezek tartalma:

I. A[] (__VER!=1),

II. A[] (__VER!=2),

III. A[] (__VER!=3),

IV. A[] (__VER01!=1)

[4] A verifikációhoz zárt világ modellt kell alkotnia a rendszernek, tehát meg fog jelenni egy felhasználói bemeneteket szolgáltató komponens is.

Az absztrakt teszteseteket eredményül kapva azokat az értelmező bemenetére megadtam, és így azokat a végrehajtási útvonalakat kaptam meg, melyek a fenti ábra alapján triviálisnak tekinthetők.

Az egyes tesztesetek során végrehajtandó utasítások a következők:

I. {input!}

II. {input!, syn?}

III. {input!, syn? output?}

IV. {}

A negyedik kifejezéshez tartozó végrehajtási útvonal a CUT-on nem következhet be, így ebben a környezetben nem tesztelhető. Az egyes tesztesetek során a naplózott értékek a következők voltak:

I. {__VER=1}

II. {__VER=1, __VER=2}

III. {__VER=1, __VER=2, __VER=3}

Ez az összehasonlítás alapján megfelel az elvárt kimenetnek, így a futó kód és a modell k-ekvivalencia szerinti konformanciáját igazoltam.

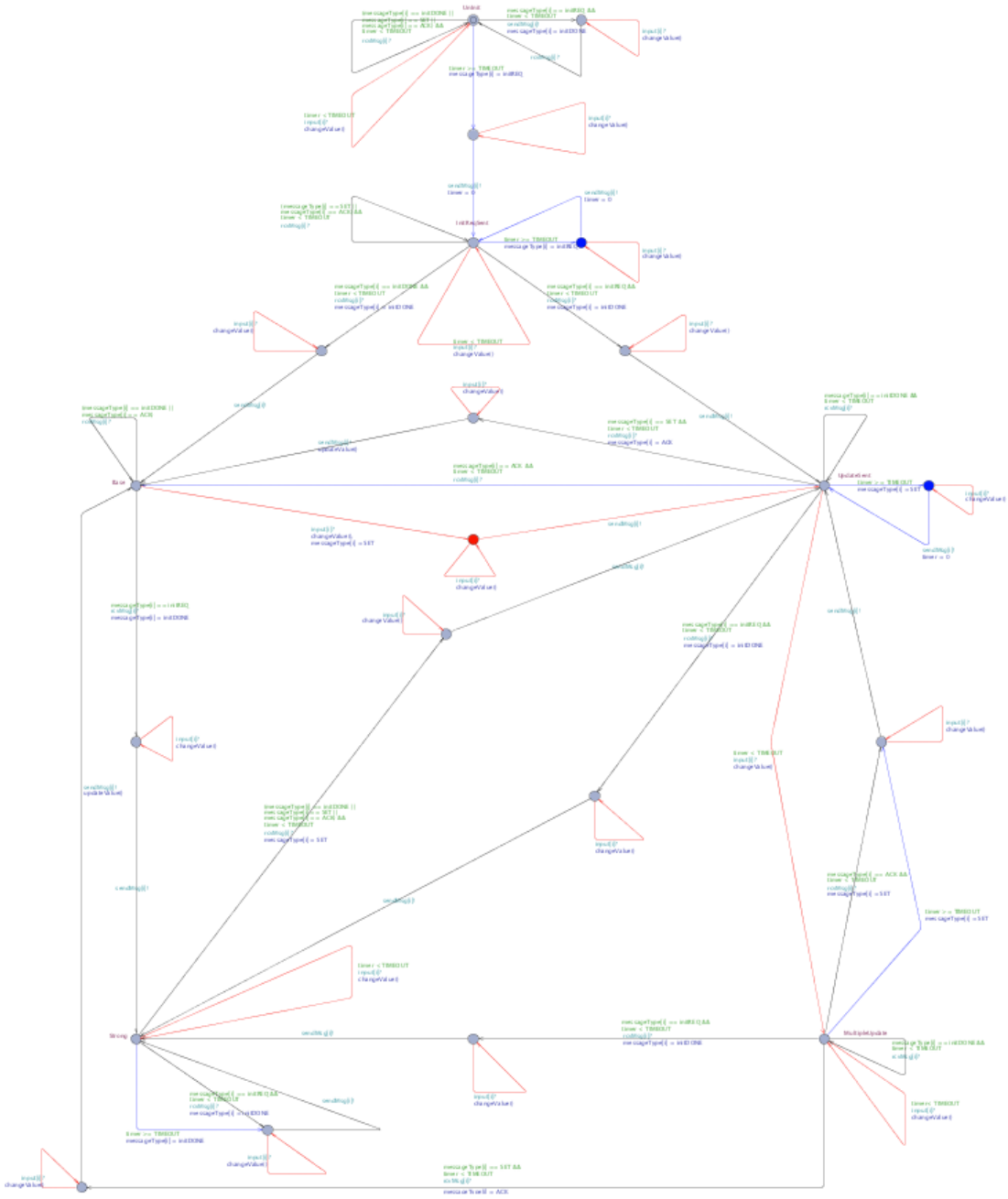
5.2.2 Mitmót konfiguráció ismertetése

A szimulátor alkalmazása előnyös volt a fejlesztés során, hiszen egy munkaállomáson tudtuk tesztelni az elosztott rendszer logikai működését. Azonban mivel a cél a konkrét alkalmazás tesztelése volt, ezért a tényleges tesztelés már a valós környezetben kell, hogy történjen. A teszteléshez szükséges konfigurációt ismertetem a továbbiakban.

Tesztelendő probléma gyanánt egy iparban is használatos protokoll – a bitszinkronizáló protokoll – vizsgálatát választottam, amelynek a modellje korábbi TDK munka [1] keretében készült. Ez egy kellően nagy (méretű és állapotterű) modellt bocsát rendelkezésünkre, így a sikeres kifejezéskészlet- és teszteset-generálás bizonyítja, hogy a bemutatott módszer ipari környezetben is megállja a helyét.

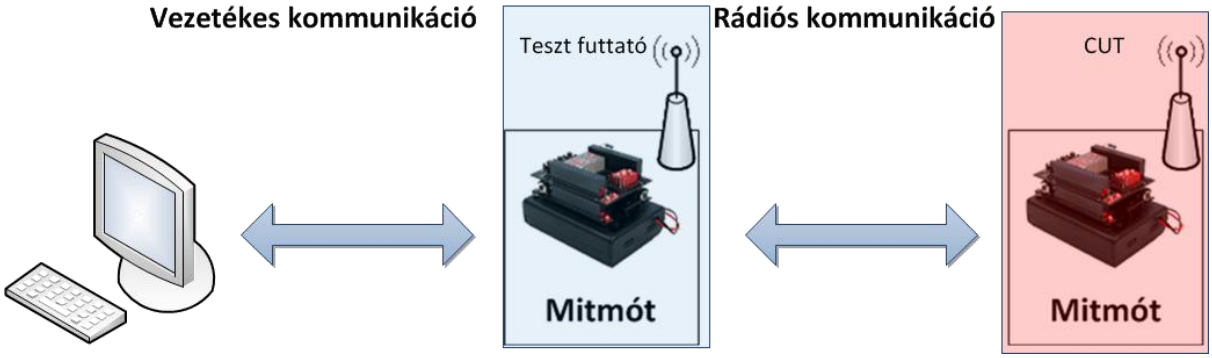
A megvalósítandó cél lazán csatolt rendszerek esetén két komponens között egy változó (bit) értékének szinkronban tartása, figyelemmel a konkurens működésre és az üzenetekkel történő kommunikációra. Az ezt megvalósító – igen terjedelmes – modell látható az 18. ábrán. Az algoritmus viselkedésének ismerete nem szükséges a tesztelési konfiguráció megértéséhez, a modell közlésével a célom pusztán a nagyságrendek szemléltetése.

A rendszerben szereplő mindkét komponensen ugyanaz az alkalmazás futtatható.



18. ábra: Bitszinkronizáló protokoll UPPAAL modellje

A tesztkonfigurációban ezek alapján a CUT szerepét egy mitmót fogja betölteni, míg egy másik mitmót szolgál teszt végrehajtóként. A teszt végrehajtó számára előnyös a számítógépes kapcsolat (pl. soros vonali kapcsolattal a tesztesetek végrehajtásának vezérlése, teszt napló begyűjtése). Egy ilyen konfigurációt mutat be a 19. ábra.



19. ábra: Mitmót tesztkonfiguráció

A fenti modell alapján a kód- és tesztgenerálás elvégezhető, ugyanakkor a mitmót platformra minden szükséges platformszolgáltatás implementációja még nincs teljesen kész, így az alkalmazás teljes tesztelése egy hátralévő feladat.

6 Összefoglalás és értékelés

6.1 Elért eredmények

Munkám során kidolgoztam olyan algoritmusokat, melyek segítségével a modell alapú fejlesztési folyamat tesztelési fázisa nagymértékben támogatható. Ennek egyik legjelentősebb pontja a különböző fedettségi kritériumoknak megfelelő tesztkészletek szisztematikus generálásának kidolgozása.

Definiáltam egy megfigyelhetőséget biztosító felműszerezési módszert, amely megkötések nélkül alkalmazható minden modellen. A felműszerezés tulajdonságait kihasználva konkrét módszereket adtam az egyes fedettségi igényeket kielégítő tesztkészletek előállítására. Mivel az alkalmazott technológiák (felműszerezés, ellenpélda segítségével történő teszteset előállítás) nem kötődnek szorosan az időzített automaták formalizmusához, ezért dolgozatom hasznos segítség lehet a hasonló területen kutatók számára.

Az absztrakt tesztesetek konkrét tesztekre történő leképezését megvalósítottam. Olyan általános módszereket alkalmaztam, amivel lehetővé vált adott platformra telepített teljes alkalmazások vizsgálata.

A tesztek kiértékeléséhez napló alapú konformancia reláció vizsgálatot alkalmazok. Az erre a célra készített értelmező modul (egy parser megvalósításával) egyszerűen illeszthető a vizsgálandó alkalmazás kimenetére. A módszert a szimulátorhoz történő illesztéssel validáltam, melyet a dolgozatban is demonstrálok egy egyszerű példán keresztül.

A tesztelt alkalmazás elkészítése kódgenerátor segítségével történik, de kézzel írt és készen kapott alkalmazások tesztelésére is alkalmazható a megadott interfészek betartása és megfigyelhetőségi igények teljesítése mellett.

Az elkészült keretrendszer a fenti eredményeket figyelembe véve praktikusan alkalmazható kritikus rendszerekhez generált forráskód (szabványok által is megkövetelt) verifikációjához. Előnyei közül kiemelhetők a teljes alkalmazás tesztelésének lehetősége, valamint a fedettségi kritériumok többféle definiálásának lehetősége.

6.2 Problémák

A keretrendszer jelenlegi legnagyobb problémája a nemdeterminisztikus működés kezelésének hiánya. Ez megkötést jelent ugyan a modellezés szempontjából, ugyanakkor – ahogy azt a 3.5. fejezetben is említettem – az ilyen modellek alkalmazása kritikus rendszerek esetén ellenjavallt.

6.3 Továbbfejlesztési lehetőségek

A vezérelhetőségre való felműszerezés megvalósítása jelentős előrelépés lenne, hiszen így a nemdeterminisztikus működés is kezelhetővé válna.

A napjainkban használt viselkedés konformancia ellenőrzők jelentős része az LTS vagy IOLTS formalizmus valamilyen reprezentációját használja a bemenetek és a kimenetek specifikálására [18]. Így a bonyolultabb relációk vizsgálatának egy lehetséges

módszere, hogy az ellenőrzés implementálása helyett a tesztesetek specifikációit és a megfigyelt kimeneteket ilyen formátumba exportálom. Ezáltal a keretrendszer tetszőleges külső ellenőrzővel kiegészíthető, így bővítve a funkcionalitást.

A nem specifikált bemenetekre adott reakciók, illetve az őrfeltételek megsértése esetén lezajló viselkedés vizsgálatához egyelőre csak a dolgozatban ismertetett módszer lett kidolgozva, ezek implementálása értékes kiegészítés lehet. Az adatfolyam alapú tesztelés támogatása (felműszerezés, fedettségi kritériumok megadása) is hasonlóan nagy kiterjesztést jelent.

A módszer alapvetéseit ugyan nem befolyásolja, de mindenképpen növelné az eszközök használhatóságát, ha rendelkezésre állna egy felhasználóbarát implementáció. A jelenlegi megvalósítás készítésekor az elsődleges cél a módszer használhatóságának igazolása, az automatizálási lehetőségek vizsgálata volt, így a modulok többsége különálló programként vagy szkriptként készült el. Egy kompakt implementáció lehetővé tenné az ipari környezetben való alkalmazást is.

Hivatkozások

[1] Horányi Gergő, Jeszenszky Balázs: Elosztott beágyazott rendszerek formális modellek alapján történő fejlesztése paraméterezhető kódgenerálás segítségével. BME-VIK TDK konferencia, 2010.

[2] UPPALL: an integrated tool environment for modeling, validation and verification of real-time systems. http://uppaal.org. (használt verzió: 4.0.13)

[3] K. G. Lasrsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. Int. Journal on Software Tools for Technology Transfer, 1(1-2):134-152, October 1997.

[4] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In W. Reisig and G. Rozenberg (eds.), Lecture Notes on Concurreny and Petri Nets, Springer LNCS-3098, 2004.

[5] MSZ EN 50128:2011. Railway application. Communication, signalling and processing systems. Software for railway control and protection systems. http://www.cenelec.eu

[6] UPPAAL CoVer: UPPAAL CoVer is a tool for creating test suites from UPPAAL models with coverage specified by coverage observers a.k.a. observer automata. http://www.hessel.nu/CoVer/index.php

[7] UPPAAL TRON: Uppaal TRON is a testing tool, based on Uppaal engine, suited for black-box conformance testing of timed systems. http://people.cs.aau.dk/~marius/tron/

[8] JTorX: JTorX is a tool to test whether the ioco testing relation holds between a given specification and a given implementation. https://fmt.ewi.utwente.nl/redmine/projects/jtorx/wiki/

[9] F. Laroussinie, About the expressive power of CTL combinators, Information Processing Letters, Volume 54, Issue 6, 23 June 1995, Pages 343-345, ISSN 0020-0190

[10] UPPAAL DTD Flat System 1.1/EN, http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd

[11] W3C DOM. http://www.w3.org/DOM/

[12] Simple DirectMedia Layer: Simple DirectMedia Layer is a cross-platform multimedia library. http://www.libsdl.org/

[13] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, Sung Deok Cha: Automatic Test Generation from Statecharts Using Model Checking, Technical Report MS-CIS-01-07, Feb 2001.

[14] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, Sung Deok Cha, Hasan Ural: Data flow testing as model checking, International Conference on Software Engineering, ISBN: 0270-5257, 2003.

[15] A. Hessel, and P. Pettersson. COVER - A test case generation tool for real-time systems. Tool paper, 6th Int. Workshop on Formal Approaches to Testing of Software (FATES'07), Tallinn, Estonia, 2007.

[16] K. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using UPPAAL. In J. Grabowski and B. Nielsen, editors, Formal Approaches to Software Testing, pages 79-94, LNCS 3395, Springer Verlag, 2005.

[17] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools, 17(3):103-120, 1996.

[18] Jan Tretmans. Model based testing with labelled transition systems. In R.M. Hierons, J.P. Bowen, and M. Harman, editors, Formal Methods and Testing, volume 4949 of Lecture Notes in Computer Science, pages 1-38. Springer, 2008.