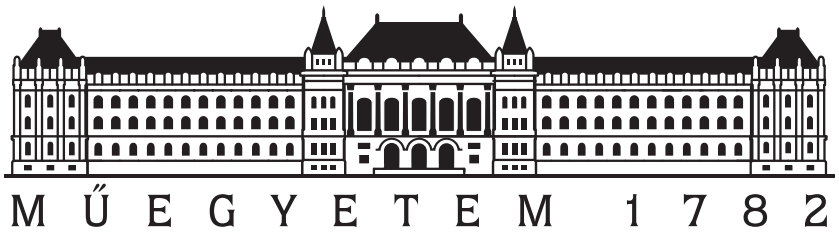


Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Monitorok automatikus szintézise elosztott beágyazott rendszerek futásidőbeli verifikációjához

TDK dolgozat

Készítette:
Horányi Gergő

Konzulens:
dr. Majzik István

2011.

Tartalomjegyzék

1. fejezet

Bevezető

Manapság számos olyan helyen alkalmaznak informatikai rendszereket, amelyek valamilyen szempontból biztonságkritikusak, azaz nem engedhető meg a helytelen működés, hiszen az komoly anyagi károkat vagy akár balesetet is okozhatna. Ilyen alkalmazási területek például a repülőgép irányítás, vasúti közlekedés vagy akár az egészségügy.

A biztonságkritikus rendszerek fejlesztése és működtetése során nagyon komoly anyagi és időbeli ráfordítást jelent a hibák detektálása. Alapvetően két különböző megközelítés létezik. Az első, hogy a tervezés során olyan módszereket használunk, amelyek valamilyen módon képesek a hibákat felderíteni, így ki tudjuk azokat javítani. Ezeket *tervezési idejű hibadetektálásnak* hívjuk. Egy viszonylag elterjedt tervezési idejű hibadetektálási módszer az úgynevezett *modellellenőrzés*. A modellellenőrzés lényege, hogy a rendszer formális modelljén megvizsgálunk különféle formalizált követelményeket. A modellellenőrzők feltérképezik a rendszer modellje alapján a rendszer összes elérhető állapotát és ezen belátják, bebizonyítják, hogy a követelmények teljesülnek. Amennyiben nem teljesülnek, sok esetben képesek megmutatni, hogy a rendszer állapotok milyen szekvenciáján került a követelménynek nem megfelelő állapotba. Ilyen megoldás például a PetriDotNet rendszer [2], vagy az általam is használt UPPAAL modellellenőrző eszköz [5].

A rendszer működése közben bekövetkező hibák észlelését megvalósító módszert *futásidejű hibadetektálásnak* nevezik. A módszer célja, hogy a rendszer működése közben fellépő állandó vagy tranziens hibák észlelhetőek legyenek és a megfelelő hibakezelő eljárások elindíthatóak legyenek. A hibák adódhatnak a rendszer hibás konfigurációjából, véletlen hardver hibákból vagy a tervező téves környezeti feltételezéseiből.

A biztonságkritikus rendszerek esetén használatos szabványokban sok esetben megjelenik a futásidejű hibadetektálás alapelve. Több szabvány is rendelkezik arról, hogy egyes rendszerek esetén kötelező egy elkülönítetten implementált, futásidejű hibadetektáló modul. Ilyen rendelkezéseket olvashatunk az *IEC 61508 szabványban*, amely az általános, biztonsághoz kötődő elektronikus eszközök szabványa, az *EN 50129 szabványban*, amely az elektronikus vasúti berendezések szabványa, vagy a légi közlekedésben használt szoftverek *DO-178B szabványában* is.

Tervezési időben csak a tervezés során elkövetett koncepcionális hibákat lehet detektálni, így nem elegendő kizárólag a tervezési fázisban foglalkozni a biztonsággal, hanem ez a teljes működés alatt fontos feladat marad. Rushby egy írása [39] alapján a mai rendszereknek már a *tanúsítását* is futásidőben lehet végezni. Dolgozatom nem vállalkozik a rendszerek megfelelőségének tanúsítására, de egy olyan megoldást mutatok be, amely akár felhasználható lenne ebben a folyamatban is.

Futás közbeni hiba detektálása esetén tipikusan az alábbi hibakezelési stratégiák választhatóak:

- a rendszert biztonságos alaphelyzetbe kell vinni és ott tartani (például a vasúti jelzőket vörös állásba ejtjük, hiszen ha minden vonat megáll, nem fog baleset történni),
- egyes rendszerek esetén (például repülőgépek) ez nem megoldható, ilyenkor szükséges a biztonságos működéshez nélkülözhetetlen alapszolgáltatások működésének fenntartása, ami valamilyen *hibatűrési technikát* igényel (rendszerint hardver, szoftver, idő vagy információ redundancia felhasználásával kell a hibaállapotot megszüntetni vagy a szolgáltatásokat átkonfigurálni).

Dolgozatom kizárólag a hiba észlelésének megvalósításával foglalkozik részletesen, hiszen minden további lépésnek is ez az alapja.

A futásidejű hibadetektáló technikák közül dolgozatom a futásidőbeli verifikációval foglalkozik részletesen. Futásidőbeli verifikációnak nevezzük azt a technológiát, amellyel a tervezési fázisban specifikált rendszerkritériumokat a rendszer tényleges működése során ellenőrizhetjük és így megbizonyosodhatunk arról, hogy a rendszer nem sérti meg a rendszer felé támasztott elvárásokat. A dolgozatban a futásidőbeli verifikációt egy *hierarchikus monitorozó rendszer* végzi, amelynek komponenseit *automatikusan szintetizáljuk* a rendszer formális modelljéből illetve a formalizált rendszerkövetelményekből. A követelmények leírására mind temporális logikai kifejezéseket, mind forgatókönyv leíró nyelveket használhatunk.

1.1. Alkalmazási terület

A bemutatott technikák és elsősorban a hierarchikus monitorozás koncepciója (3. fejezet) az elosztott beágyazott rendszerek területére koncentrál. A beágyazott rendszerek egy alapvető tulajdonsága, hogy a rendszer erőforrásai korlátozottak. Ebből kifolyólag bármely olyan megoldásnak, amely beágyazott rendszerekhez készül, rendkívül fontos tulajdonsága az erőforrás igényessége. Ez megjelenhet, mint nagyobb memória vagy processzorhasználat (*futási idő*) igény, de megjelenhet, mint kommunikációs többletterhelés, hiszen az elosztott rendszerek esetén a rendszer komponensei egymással üzenetekkel kommunikálnak és a monitorozó komponensek a rendszer számára további kommunikációs terhelést jelentenek. Munkám során az említett két tény hangsúlyos szerepet kapott. A 7.5. fejezetben bemutatott eredményekkel kívánom alátámasztani, hogy a bemutatott rendszer jól használható elosztott beágyazott rendszerek esetében.

A beágyazott rendszerek, amelyekhez a monitorozó komponenseket szintetizálni fogom eseményvezérelt, állapot-alapú rendszerek. Olyan rendszerekre fókuszáltam, amelyek nem túl bonyolult vezérlési szerkezettel rendelkező komponensek hálózataként írhatóak le. Az alábbi tulajdonságokat is feltételeztem a vizsgált rendszerek esetén:

- korlátozott számú szenzorral rendelkezik, mint a rendszer bemenete,
- korlátozott számú beavatkozóval rendelkezik, mint a rendszer kimenete, valamint
- kommunikációs képességekkel is bír (adott esetben vezeték nélkül is).

A rendszerek rendelkezhetnek valós idejű követelményekkel is, hiszen ez gyakori elvárás a beágyazott vezérlőkkel szemben. A következő fejezetben bemutatom, hogy milyen formalizmust választottam, amellyel a most bemutatott rendszereket hatékonyan lehet leírni.

Munkám során egy *motivációs mintapélda* is fontos szerepet kapott, amelyet a következőkben bemutatok.

Az alkalmazás egy *modellvasút rendszer*, amely vezérlését az egyes vasúti objektumok lokális vezérlését végző mikrokontrollerekből álló, *elosztott rendszer* végzi el (A függelék).

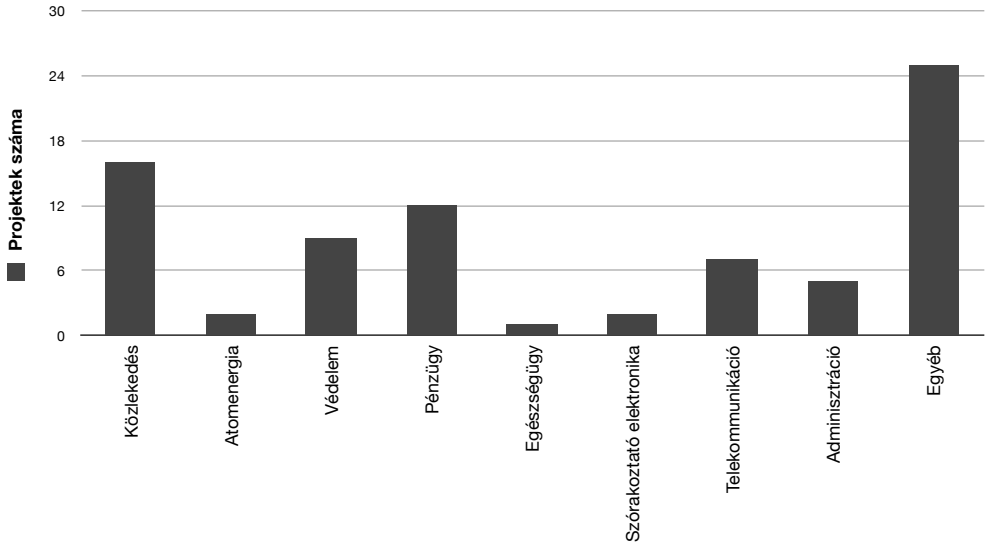
A mikrokontrollerekre a program automatikusan generálható egy verifikált formális modell alapján. A kódgenerálás a korábbi munkám [40] felhasználásával történik. Ez az alkalmazás rendelkezik a korábban leírt összes tulajdonsággal. A 7. fejezetben részletesen bemutatom, hogy a munkámban kidolgozott megoldásokat hogyan illesztettem ehhez a rendszerhez.

1.2. Formális módszerek használata ipari környezetben

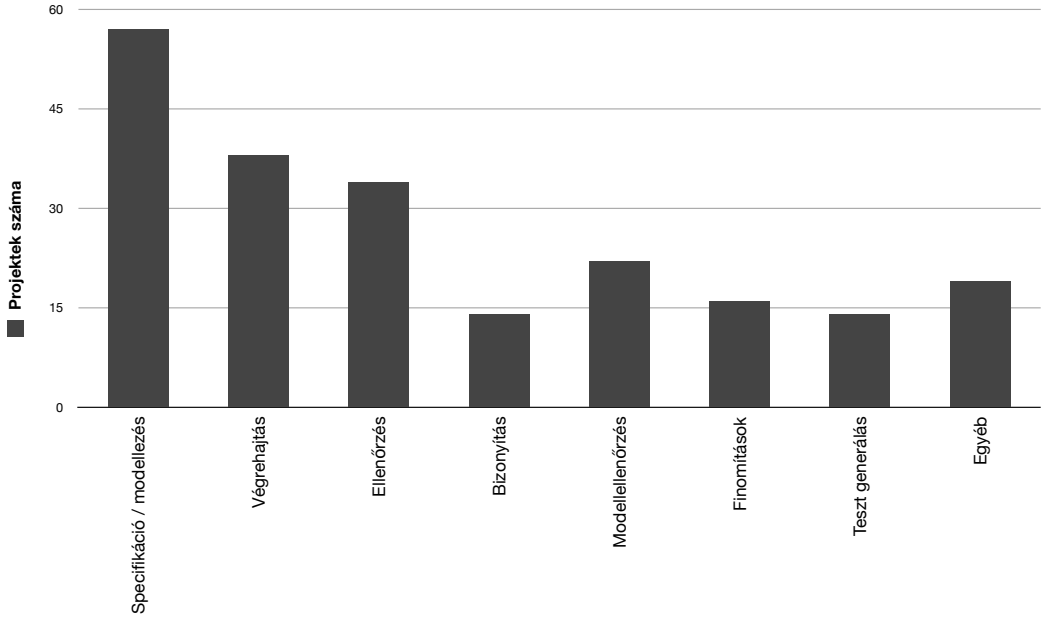
A futásidejű verifikáció illetve a modellellenőrzés egy-egy használati esete a formális módszereknek. Annak ellenére, hogy a formális modellezési módszerek már régóta ismertek, az ipari alkalmazásuk az utóbbi években kezdett el megjelenni. 2009-ben egy részletes és széleskörű felmérést készítettek a formális módszerek ipari elterjedéséről [43] és többek közt az 1.1. ábrán látható eredményeket kapták. A felmérés szintén kimutatta, hogy annak ellenére, hogy már a 90-es évek elején több publikáció született, hogy ezen módszereket hogyan érdemes felhasználni - ezek közül kiemelendő például John Rushby cikke [38], amely a formális módszerek NASA-nál történő felhasználásának lehetőségét mutatja be - a tényleges felhasználások legjava 2000 illetve 2006 környékén kezdődött meg. Az is jól látható [43]-ban, hogy a formális módszereket alkalmazó projektek száma egyre nő az idő előrehaladtával. Mindez igazolja, hogy a dolgozatomban bemutatott, formális módszereken alapuló automatikus monitorszintetizáló rendszernek van létjogosultsága.

1.3. A dolgozat felépítése

Dolgozatom az alábbi részekre bontható. A 2. fejezetben a munkám technológiai hátterét mutatom be, ismertetem a már korábban leírt technológiákat és definíciókat. A 3. fejezetben a hierarchikus monitorozó rendszer koncepcióját mutatom be. A 4. fejezet a komponens szintű, vezérlési folyam ellenőrző monitorok szintézisét mutatja be, míg az 5. és 6. fejezetekben a rendszer szintű monitorok előállítását mutatom be részletesen. A 7. fejezetben bemutatom a tényleges megvalósítását a futásidejű verifikációs eszközömnek. Végezetül a 8. fejezetben összefoglalom munkámat.



(a) Formális módszereket alkalmazó projektek száma alkalmazási területenként



(b) Formális módszereket alkalmazó projektek száma a módszerek alapján csoportosítva

1.1. ábra. A 2009-es formális módszerek ipari felhasználását vizsgáló felmérés [43] néhány eredménye

2. fejezet

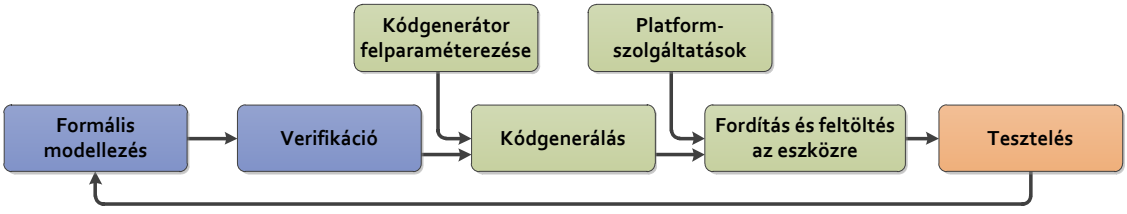
Technológiai háttér

Az 1.1. fejezetben bemutattam, hogy munkám során milyen alkalmazási környezet támogatását tűztem ki célul. Az alkalmazási környezethez illő formalizmus kiválasztása rendkívül fontos, így az általam felhasznált formalizmust szeretném bemutatni ebben a fejezetben. Úgy találtam, hogy a *formális modelleken alapuló fejlesztési megoldások* jól felhasználhatóak ebben az alkalmazási környezetben, így a következőkben ezt részletesen bemutatom a választott *időzített automata formalizmus* mellett.

2.1. Formális modelleken alapuló fejlesztés

Ahogy az 1. fejezetben olvasható, biztonságkritikus alkalmazások esetén a tervezés során olyan megoldásokat kell alkalmazni, amelyek lehetővé teszik, hogy a rendszer lehetőleg hibamentesen, a specifikációnak megfelelően működjön. A formális modelleken alapuló fejlesztési módszer a tervezési hibák elkerülésére egy hatékony megoldás, különösen a modellellenőrző technológiákkal (lásd 2.2. fejezet) párosítva. Ehhez a tervezendő rendszernek egy pontos, formális modelljét kell először elkészíteni, amelyen a tulajdonságok megvizsgálhatóak, a követelmények ellenőrizhetőek. A módszer igazi előnye, hogy az elkészített modellek precíz szemantikával rendelkeznek (ezért nevezethető *formális* módszernek), így amellett, hogy szemléletesek lehetnek, különféle automatikus ellenőrzéseket is végrehajthatunk a modelleken, amelyek képesek lehetnek különféle állításokat bizonyítani vagy megcáfolni még az alkalmazás implementálása előtt.

Előző munkámban [40] egy ilyen fejlesztési folyamathoz automatikus kódgenerátor eszközt készítettem el. A dolgozatban bemutatott fejlesztési folyamat a 2.1 ábrán látható.



2.1. ábra. A korábbi munkámban bemutatott fejlesztési folyamat [40]

2.1.1. Az időzített automata formalizmus

Munkám során úgy találtam, az időzített automaták formalizmusa [6] az alkalmazási területnek megfelelő kifejező erővel rendelkezik. Ebben a fejezetben röviden bemutatom és definiálom az időzített automaták szintaxisát és szemantikáját.

Az időzített automaták alapvetően véges automaták, azaz állapotok és állapotátmenetek véges halmazai. Ez a formalizmus a véges automaták formalizmusát valós értékű változókkal (*óraváltozókkal*) egészíti ki. Egy időzített automata egy időzített rendszer absztrakt modelljeként tekinthető. A változók a rendszer logikai óráit jelenítik meg, amelyek az inicializálás pillanatában 0 értéket vesznek fel, majd egymással szinkronban növekednek. A rendszer éleihez feltételként szolgáló órakifejezések, az állapotokhoz pedig órainvariánsok rendelhetőek.

Az időzített automaták szintaxisa

Tekintsük valós változók $\mathcal{C}$ véges halmazát, mint óraváltozók véges halmazát és Σ-t, mint az akciók véges halmazát. Órakifejezéseknek tekintjük az alábbi atomok konjunktív formáját: $x \sim n$ vagy $x - y \sim n$, ha $x, y \in \mathcal{C}, \sim \in \{\leq, <, =, >, \geq\}$ és $n \in \mathbb{N}$. Órakifejezések az időzített automata őrfeltételeiként használhatóak. Az órakifejezések halmazát $\mathcal{B}(\mathcal{C})$-vel jelöljük.

1. Definíció (Időzített automaták). *Az időzített automata $\mathcal{A}$ egy olyan $\langle N, l_0, E, I\rangle$ négyes, ahol*

- *N állapotok véges halmaza,*
- *$l_0 \in N$ kezdőállapot,*
- *$E \subseteq N \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times N$ élek (állapotátmenetek) véges halmaza,*
- *$I : N \longrightarrow \mathcal{B}(\mathcal{C})$ függvény, ami invariánsokat rendel az állapotokhoz.*

Az $l \xrightarrow{g,a,r} l'$ jelölés akkor használható, ha $\langle l, g, a, r, l'\rangle \in E$, azaz vezet l állapotból átmenet l' állapotba, amelyhez g őrfeltétel, a akció és r órahalmaz van rendelve. Az r órahalmaz azokat az órákat tartalmazza, amelyeket az állapotátmenet során le kell nullázni.

Az időzített automaták szemantikája

Az időzített automaták szemantikája egy *tranzíciós rendszerként* írható le [13], ahol a tranzíciós rendszer egy állapota az időzített automata egy állapotával illetve az óraváltozók aktuális értékével írható le. Kétféle átmenet lehetséges a tranzíciós rendszerben. Egy automata várakozhat egy állapotban, azaz csak az óraváltozók értéke változik, vagy egy engedélyezett élen elhagyhatja az állapotot, tehát ilyenkor az óraváltozók értékén felül az automata állapota is megváltozik.

Az óraváltozók értékeit a $\mathcal{C}$ óraváltozókhoz nem negatív valós számokat rendelő függvények adják meg. Tekintsük u-t és v-t ilyen értékhozzárendelő függvényeknek és ha u függvény által jelölt érték kielégíti g őrfeltételt, azt jelöljük $u \models g$ módon. Az $u + d$ jelölje $d \in \mathbb{R}_+$ esetén azt a függvényt, amely minden $x \in \mathcal{C}$-hez $u(x)+d$-t rendel. Jelöljük $[r \mapsto 0]u$ módon azokat a függvényeket, amelyek minden $x \in r \subseteq \mathcal{C}$ órához nullát rendelnek és u-t rendelnek minden $x \in (C \backslash r)$ órához.

2. Definíció (Időzített automaták szemantikája). *Az időzített automaták szemantikája egy tranzíciós rendszer, ahol az állapotok $\langle l, u\rangle$ párosok és az átmeneteket az alábbi két szabály adja meg:*

- $\langle l, u\rangle \xrightarrow{d} \langle l, u+d\rangle$, *ha* $u \in I(l)$ *és* $(u+d) \in I(l)$ *egy adott nem negatív* d *valós szám esetén, illetve*
- $\langle l, u\rangle \xrightarrow{a} \langle l', u'\rangle$, *ha* $u \in g, u' = [r \mapsto 0]u$ *és* $u' \in I(l')$.

2.1.2. Automaták hálózata

Lehetőség van időzített automaták hálózatát is leírni, amely a párhuzamos kompozíciója $\mathcal{A}_1 \ldots \mathcal{A}_n$ időzített automatáknak, vagy más néven folyamatoknak. Az automaták között szinkronizációs akciók segítségével szinkron kommunikáció hozható létre ($a?$ és $a!$ fogadó és küldő szinkronizációk segítségével), míg közös (globális) változók segítségével aszinkron kommunikáció is megvalósítható. A szinkronizáció előre definiált csatornákon történhet. A lehetséges állapotátmeneteket az időzített automata formalizmusánál bemutatott szabályok kissé kibővített változata adja meg [13].

3. Definíció (Időzített automata hálózat átmeneti szabályai). *Az időzített automata hálózatok esetén egy állapotot az* $\langle l, u\rangle$ *páros jelöl, ahol* l *a egyes folyamatok állapotát tartalmazó lista, míg* u *a rendszer összes óráját leíró függvényhalmaz. Az alábbi átmenetek léteznek egy időzített automata hálózatban:*

- $\langle l, u\rangle \xrightarrow{d} \langle l, u+d\rangle$, *ha* $u \in I(l)$ *és* $(u+d) \in I(l)$ *egy adott nem negatív* d *valós szám esetén,*
- $\langle l, u\rangle \xrightarrow{r} \langle l[l_i'/l_i], u'\rangle$, *ha* $l_i \xrightarrow{g,\tau,r} l_i', u \in g, u' = [r \mapsto 0]u, u' \in I(l[l_i'/l_i])$, *valamint*
- $\langle l, u\rangle \xrightarrow{r} \left\langle l[l_i'/l_i][l_j'/l_j], u'\right\rangle$, *ha létezik olyan* $i \neq j$, *hogy*
  1. $l_i \xrightarrow{g_i,a?,r_i} l_i', l_j \xrightarrow{g_j,a!,r_j} l_j'$ *és* $u \in g_i \wedge g_j$, *és*
  2. $u' = [r_i \bigcup r_j \mapsto 0]u$ *és* $u' \in I(l[l_i'/l_i][l_j'/l_j])$,

ahol $l[l_i'/l_i]$ *jelölje az* l *helyek olyan listáját, ahol* l_i*-t* l_i'*-re cseréltük,* τ *pedig a belső akciók halmazát (nem szinkronizáló akciók).*

2.1.3. A követelmények formalizálása

A rendszer követelményeinek specifikálása különféle módokon történhet meg. Sok esetben temporális logikai kifejezésekkel írhatóak le a követelmények. A temporális logikai nyelveket 1977-ben Pnueli vezette be [37]. A temporális logikák az adott tulajdonságú állapotok elérhetőségét fogalmazzák meg a rendszer állapotterében. Tipikus temporális operátorok a *„mindig"*, *„következő állapotban"*, *„valamikor a jövőben"* illetve az *„addig amíg"* jelentésű operátorok.

A ma elterjedt modellellenőrző rendszerek különféle változatait használják a temporális logikai nyelvnek. Ezek közül a legelterjedtebbek az *Linear temporal logic* logikai kifejezésekkel lehet követelményeket specifikálni a klasszikus SMV eszközben [1] (CTL és LTL) vagy a SPIN modellellenőrzőben [3] (LTL). Az általam használt UPPAAL eszköz verifikációs modulja úgynevezett *időzített CTL (TCTL)* kifejezéseket képes ellenőrizni, amely a CTL formalizmus egy időzített variánsa.

A CTL formalizmus

A CTL formalizmus egy elágazó idejű temporális logikai nyelv, amelyet 1985-ben vezetett be Emerson és Halpern [22]. Segítségével kijelentések logikai időbeli változását vizsgálhatjuk. A kezdőállapotból kiindulva az úgynevezett elágazó idővonalak mentén az egymás utáni állapotokat egy fa-szerű struktúrában ábrázolhatjuk, ahol minden állapotnak

legalább egy rákövetkező állapota lehet, amennyiben a rendszer nem jut holtpontra. A kifejezéseket ebben a számítási fában fogjuk kiértékelni. A CTL kifejezések szintaxisát az alábbi szabályok határozzák meg [34]:

- Minden P atomi kijelentés egy állapotkifejezés. (Atomi kijelentésekkel írjuk le az állapotok lokális tulajdonságait.)
- Ha p és q állapotkifejezések, akkor $\neg p$ és $p \wedge q$ is állapotkifejezés.
- Ha p és q állapotkifejezések, akkor Xp, pUq, Fp és Gp útvonalkifejezések.
- Ha s útvonalkifejezés, akkor Es és As állapotkifejezések.

A CTL segítségével állapotkifejezések logikai értékét lehet vizsgálni, így minden útvonalkifejezés elé kerül egy útvonalkvantor (E vagy A).

A CTL kifejezések szemantikája a következőképpen definiálható. Az útvonalkvantorok szemléletes jelentése a következő:

- A: az adott állapotból kiindulva minden lehetséges útra *(forAll)*.
- E: az adott állapotból kiindulva legalább egy útra *(Exists)*.

Az állapotoperátorok szemléletes jelentései a következők:

- Fp: az útvonal egy tetszőleges állapotán p igaz *(Future)*.
- Gp: az útvonal összes állapotán p igaz *(Globally)*.
- Xp: a következő állapotban p igaz *(neXt)*.
- pUq az útvonal egy állapotában igaz lesz q és addig minden állapotban igaz p *(Until)*.

1. példa. *Néhány konkrét CTL kifejezés a jelentésükkel:*

- $AG!p$*: A p kifejezés logikai értéke sose legyen igaz.*
- EFp*: Legyen olyan lefutása rendszernek, hogy p kifejezés logikai értéke igaz.*
- AFp*: Minden lehetséges lefutás során p logikai értéke legyen igaz.*
- $AG(!p \vee AFq)$*: Ha p kifejezés igazzá válik, akkor minden lehetséges további útvonalon q is igazzá fog válni. (A kifejezés megegyeik az* $AG(p \rightarrow q)$ *implikációt tartalmazó kifejezéssel.)*

A Live Sequence Chart (LSC) formalizmus

A Live Sequence Chart (röviden LSC) formalizmust 2001-ben publikálták először [19] és azóta különféle alkalmazásai találhatóak meg az irodalomban – kiemelendő a PlayEngine automatikus LSC „lejátszó” eszköz [25] és annak egy konkrét telekommunikációs alkalmazása többek közt a Microsoft Research által [18]). Az LSC segítségével egymással kommunikáló komponensek közötti üzenetek szekvenciáit, úgynevezett forgatókönyveket lehet leírni.

A formalizmust megalkotókat az a tény motiválta, hogy az iparban elterjedt forgatókönyv leíró nyelv a Message Sequence Chart (MSC, ITU-T által szabványosítva [27]) nem rendelkezik megfelelően definiált szemantikával. A problémát jól mutatja az is, hogy az évek során több különböző szemantika leírás (ilyen például [28]) és analizátor megoldás (például [12] vagy [7]) is készült az MSC nyelvhez.

Az LSC formalizmus sokban hasonlít az MSC nyelvre. A legfőbb bővítései a modalitások rendelése a forgatókönyvekhez, illetve precíz szemantikával is rendelkezik. Egy LSC forgatókönyv az MSC ellenében lehet opcionális vagy kötelező érvényű is. Az alábbiakban olvasható az LSC formális definíciója [31].

4. Definíció (Live Sequence Chart). *Egy $\mathcal{L}$ Live Sequence Chart folyamatok véges I halmazának egymás közti interakcióját írja le. Minden $i \in I$ folyamat életvonalán pozíciók véges halmaza található ($pos(i) = \{0, 1, \ldots p\_max_i\}$), amely a kommunikáció illetve a számítások lehetséges pontjait jelöli. $\mathcal{L}$ összes pozícióját $L = \{\langle i, p\rangle \,|i \in I \wedge p \in pos(i)\}$ módon jelöljük.*

Egy LSC két részre bontható. Az első az úgynevezett prechart, amely a forgatókönyv előfeltételeit (egy feltételnek tekinthető üzenetküldési mintáját) írja le. Ha a prechart-ban leírt viselkedés bekövetkezik a rendszerben, akkor az LSC második részének, azaz a mainchartnak be kell következnie ahhoz, hogy a forgatókönyv teljesüljön.

Jelölje Σ a lehetséges üzeneteket (megegyezik az UPPAAL-beli csatornákkal). Egy üzenet $m = (\langle i, p\rangle, \sigma, \langle i', p'\rangle) \in L \times \Sigma \times L$ az i folyamat p pozíciójából σ címkével küldődik i' folyamatnak, amely a p' pozícióban tartózkodik. Az $\mathcal{L}$ forgatókönyvben ténylegesen megjelenő üzenetek véges halmazát jelölje M. Az üzenetküldés azonnali. Az üzeneteknek két típusa van: Forró üzenetek, amelyek soha nem vesznek el, illetve hideg üzenetek, amelyek a küldés után nem feltétlenül érkeznek meg. A dolgozatban kizárólag forró üzenetekkel foglalkozom[1]*. Szintén nem foglalkozom párhuzamos üzenetküldéssel, azaz egy pozíció csak egy üzenetnek lehet a forrása vagy a végpontja.*

Jelölje $\mathcal{L}$ óraváltozóinak véges halmazát $C_{\mathcal{L}}$, valamint az eredeti $\mathcal{S}$ rendszer óraváltozóinak halmazát $C_{\mathcal{S}}$. A forgatókönyv által olvasható órák a $C = C_{\mathcal{L}} \bigcup C_{\mathcal{S}}$ halmaz. A forgatókönyv – mivel az eredeti rendszer működésébe nem avatkozik be – kizárólag a $C_{\mathcal{L}}$ órákat nullázhatja le.

A forgatókönyv feltételeinek halmazát jelölje G. Egy feltétel órakifejezések véges konjunkciójából áll elő. Az órakifejezések az LSC-ben megegyeznek 2.1.1. fejezetben leírtakkal. Egy feltétel lehet forró vagy hideg. Hideg feltételek nem teljesülése a forgatókönyv meghiúsulását vonja maga után, míg a forró feltételek nem teljesülése esetén a forgatókönyv megsérül.

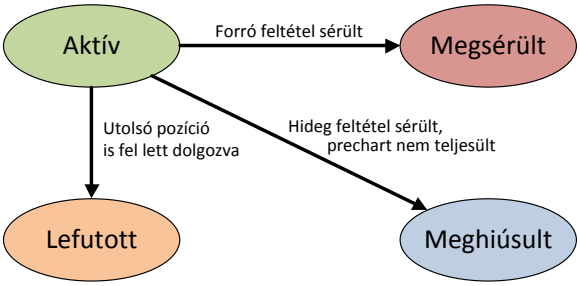
Az akciók halmazát jelölje $A = 2^{C_{\mathcal{L}}}$. Egy akció órák egy a halmazát nullázhatja le. Egy m üzenet elküldése esetén az i_1 forrás és i_2 címzett megfelelő pozíciójához g feltétel és a akció rendelhető. A g feltétel egy predikátum, amely az üzenetküldés után azonnal kiértékelődik (gyakran ASAP szemantikaként hivatkoznak erre a tényre). Ha a kiértékelés eredménye hamis, akkor az a akció nem hajtható végre. Az akció, amely az a-ban található órákat nullázza le, akkor fut le, ha a feltétel teljesül. Az üzenet, feltétel és akció együttesen egy atomi lépése a forgatókönyvnek, amelyre az irodalom szimrégióként hivatkozik [30].

Egy forgatókönyv lefutása után három végállapot létezhet (2.2. ábra):

- Sikeresen lefut a forgatókönyv, azaz a prechart teljesül illetve a mainchart által meghatározott üzenetszekvenciák is megjelennek a rendszerben, valamint a feltételek is teljesülnek (forró és hideg egyaránt).

- Meghiúsul a forgatókönyv, amikor egy hideg feltétel sérül, vagy a prechart nem teljesül. Ez nem számít hibás működésnek, azt jelenti, hogy az aktuális lefutásra nem illeszthető a forgatókönyv.

[1] Feltételezem a megbízható üzenetküldést (amit a modellezett alkalmazás szint alatti üzenetkezelő réteg biztosít).



2.2. ábra. A forgatókönyvek lehetséges végállapotai

- Megsérül a forgatókönyv, azaz a lefutás nem teljesíti a forgatókönyv által leírt specifikációt (nem teljesül egy forró feltétel, vagy olyan kommunikációs szekvencia történik a rendszerben, amely nem megengedett).

5. Definíció (Szimultán régió (szimrégió)). *Az s szimrégió egy halmaz, amely egy üzenetet, feltételt és akciót tartalmazhat $s \subseteq (M \bigcup G \bigcup A)$ és az alábbi követelményeknek felel meg:*

- *Nem üres:* $\exists e \in (M \bigcup G \bigcup A) : e \in s$
- *Egyedi:* $\forall m, n \in M : (m \in s \wedge n \in s) \Rightarrow m = n$ *(feltételekre és akciókra hasonlóan)*
- *Nem átlapolódó:* $\forall s, s' : \forall e \in (M \bigcup G \bigcup A) : (e \in s \wedge e \in s') \Rightarrow s = s'$

A forgatókönyv összes szimrégióját jelölje $S \subseteq 2^{(M \bigcup G \bigcup A)}$. Amennyiben egy szimrégió nem tartalmaz üzenetküldést, akkor a szimrégiót azonnal végre kell hajtani (ASAP szemantika).

Végezetül az LSC szemantikáját is részben definiálom. A teljes szemantika definíció [19] és [31]-ben olvasható. A forgatókönyv pozíciói részlegesen rendezettek a következő szabályok alapján:

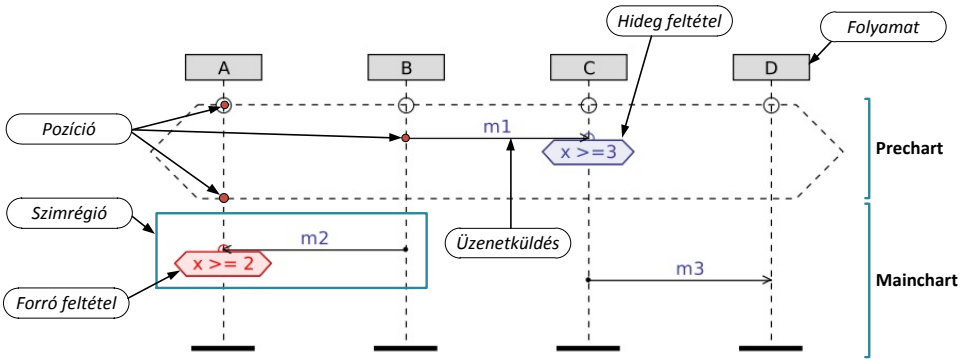
- Egy folyamaton belül ha l_1 l_2 felett van, akkor $l_1 \leq l_2$
- Egy szimrégión belül egyik pozíció sincs a másik után:
  $\forall s \in S, \forall l, l' \in L : (\lambda(l) = s) \wedge (\lambda(l') = s) \Rightarrow (l \leq l') \wedge (l' \leq l)$

ahol λ függvény a pozíciókat szimrégióhoz rendeli. A részleges rendezés reláció $\preceq \subseteq L \times L$ a $\leq$ reláció tranzitív lezártjaként definiálható. A dolgozatom szempontjából még fontos szerepet kapnak a forgatókönyvek vágásai, amelynek definíciója alább olvasható.

6. Definíció (Vágás). *Egy forgatókönyv vágása pozíciók alulról zárt halmaza, amely az összes folyamat életvonalán végighúzódik. Az alulról zártság az jelenti, hogy ha egy pozíció a vágásban szerepel, akkor minden olyan pozíció is a vágás része, amely a sorba rendezés reláció szerint előtte van, azaz $\forall c \subseteq L, \forall l, l' \in L : (l \in c \wedge l' \preceq l) \Rightarrow l' \in c$.*

Az LSC-ben mindig található egy minimális illetve maximálás vágás. A minimális vágás üres, azaz nem tartalmaz pozíciókat. A maximális ezzel ellentétben az összes pozíciót tartalmazza. Ha az LSC-nek van nem üres prechart-ja, akkor létezik egy másik kiemelt vágás a mainchart „elején”, amely az összes prechart-beli pozíciót tartalmazza.

A Live Seqeuence Chart formalizmus alapelemeit a 2.3. ábra mutatja be.



2.3. ábra. Példa LSC diagram az alapelemek bemutatásával

2.2. Követelmények ellenőrzése a tervezés során

Ahogy az 1.1(b). ábrán látható, a formális módszerek tervezési időben történő alkalmazásának módja sok esetben a modellellenőrzés. A modellellenőrző eszköz bemenete a rendszer formális modellje illetve a rendszer formalizált követelményei. A modellellenőrző ebből a két bemenetből előállítja a kimenetét, amely lehet igenleges vagy nemleges. Sok modellellenőrző rendszer – mint például az UPPAAL is – képes ellenpéldát vagy bizonyítékot mutatni a követelmények megsérülésére vagy teljesülésére. Maga a modellellenőrzés problémája az alábbi módon formalizálható. Legyen M a rendszer állapotait és állapotátmeneteit leíró Kripke struktúra. Jelölje f a rendszerkövetelményeket leíró temporális logikai kifejezést és s a rendszer kezdőállapotát. A modellellenőrző feladata, hogy eldöntse $M, s \models p$ teljesülését [16], azaz, hogy adott rendszer adott kezdőállapotból kiindulva teljesíti-e az adott követelményeket.

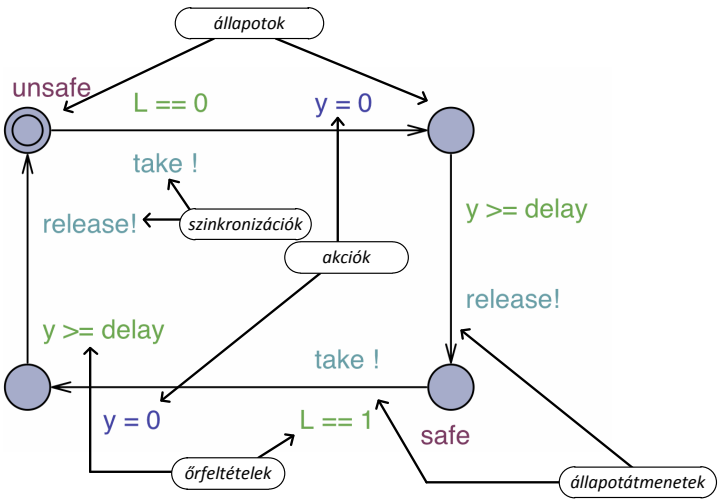
2.2.1. Az UPPAAL modellellenőrző eszköz

Az UPPAAL modellellenőrző eszközt Alur és Dill munkája [6] alapján fejlesztették ki, így a modellezés során időzített automaták hálózatát lehet elkészíteni. Követelményeket a TCTL formalizmus segítségével lehet megadni az eszköz verifikációs moduljának, amely képes ellenpéldát mutatni a kifejezésre, amennyiben az nem teljesül. A modellellenőrző képes LSC forgatókönyveket is ellenőrizni az időzített automata hálózat modellen. Az UPPAAL képes az automata hálózat szimulációjára, azaz lépésenként végig lehet követni a rendszer működését.

Az UPPAAL további bővítéseket is illeszt az időzített automata formalizmushoz, mint például atomi állapotok, globális változók vagy sürgős csatornák. Ezekről részletesen [4]-ben lehet olvasni. A 2.4. ábrán egy UPPAAL-ban elkészített vezérlő látható a szintaxis részeit magyarázó feliratokkal.

2.3. Minta alapú forráskód szintézis beágyazott vezérlőkhöz

Korábbi munkám során egy formális modelleken alapuló forráskód generátor eszközt készítettem el. Dolgozatomban ezt a kódgenerátor eszközt használtam fel, bővítettem tovább, így röviden bemutatom az alapvető koncepciókat a kódgenerátor mögött. A teljes értékű, részletes leírás [40]-ben olvasható. A kódgenerátor bemenete, vagyis a generált kód alapja az előzőekben bemutatott időzített automata formalizmus segítségével leírt rendszermodell.



2.4. ábra. Mintapélda az UPPAAL-ban készíthető vezérlőkre

Az elkészített kódgenerátor egy általános célú és platformfüggetlen megoldás. Ennek eléréséhez le kellett választani a generátortól a platformfüggő részeket. Ennek megoldására a kódgenerátor számára absztrakt platformszolgáltatások definiálhatóak, amelyeket minden platformra külön-külön implementálni kell. A generátor az implementációt nem ismeri, csak a szolgáltatásdefinícióban leírtakat tételezi fel a szolgáltatásról.

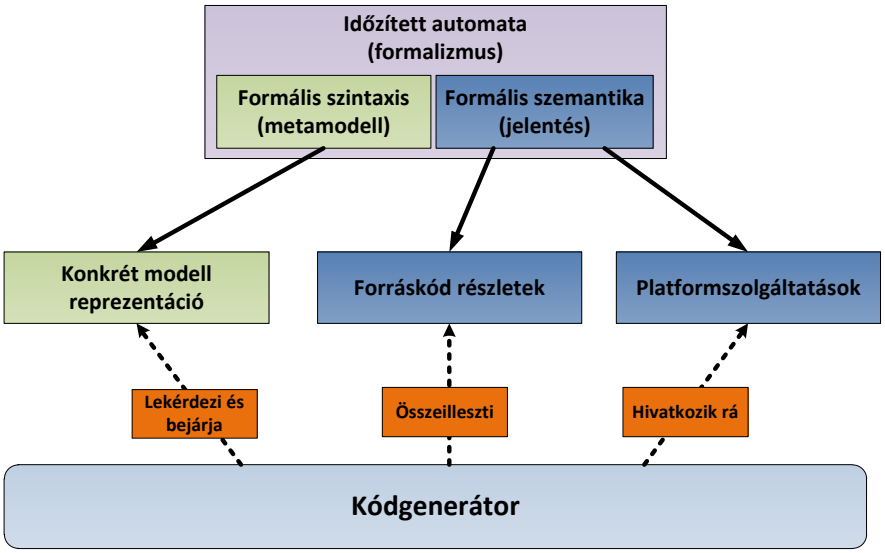
Két típusú platform szolgáltatást különböztethetünk meg:

- Az időzített automata szemantikájához kötődő szolgáltatások, mint például a *kommunikációs szolgáltatás*, amely az automaták közötti szinkronizációs akciók megvalósításához szükséges, illetve
- a szemantikától független, azt kibővítő szolgáltatások, mint például a *naplózó szolgáltatás*, amely a rendszer nyomonkövethetőségét teszi lehetővé.

A szolgáltatásdefiníciókat külső konfigurációs fájlokkal írhatjuk le. Szintén a platformfüggőség elkerülése végett került bevezetésre egy másik leíró állomány, amely a platform perifériáihoz történő hozzárendeléseket írja le. Ennek segítségével az UPPAAL modell egyes változóit közvetlen kivezethetjük a beágyazott rendszer egy kimenetére. Ehhez meg kell adni, hogy a kimenetre hogyan lehet értéket kiírni, illetve bemenet esetén a beolvasás mechanizmusát kell definiálni. A kódgenerátor paraméterezhető, azaz ezek a szolgáltatások, változókötések külső konfigurációs állományokból testreszabhatóak.

Miután a vezérlő formális modellje elkészült – legyen ez a modellvasút vezérlő modellje, vagy tetszőleges más alkalmazásé – és a platformszolgáltatások leírása és implementációja is rendelkezésünkre áll, a kódgenerátor képes automatikusan fordításra kész forráskódot előállítani. A következőekben szeretném bemutatni a generált forráskód szerkezetét.

A generált forráskód két blokkra bontható. Az első, az *automata szintű* rész, míg a második az *állapot szintű* részek. Automata szinten létrejön egy rendszerciklus és természetesen sok inicializáló kód is. A szemantika megvalósításáért alapvetően az állapotonként létrehozott három függvény felel. Minden állapothoz generálódik egy belépő–, várakozó– és kilépőfüggvény. A belépőfüggvény felelőssége, hogy az állapotba történő belépést kezelje, az állapothoz szükséges adatstruktúrákat alaphelyzetbe állítsa. A várakozó függvény kezeli, az állapotban való tartózkodást illetve a kilépési feltételek vizsgálatát. Az állapotból való kilépést a kilépőfüggvény kezeli le.



2.5. ábra. A kódgenerátor alapvető működése

A kódgenerátor egy minta alapú eszköz, azaz rendelkezik egy kódgenerálási mintával, Java Emitter Templates keretrendszerrel leírva. A kódgenerálási minta bővítése akkor válik szükségessé, ha valamilyen olyan kiegészítő szolgáltatást szeretnénk illeszteni a rendszerhez, amely jól köthető a bemutatott szemantikai koncepcióhoz (belépés, várakozás az állapotban, kilépés). A dolgozatom témáját jelentő futásidejű verifikációt lehetővé tevő monitorozó komponensek is ilyen módon lettek illesztve a monitorozott rendszerhez.

A kódgenerátor alapvető működését a 2.5. ábra mutatja be. A kódgenerátor feladata a konkrét időzített automata modell reprezentációt bejárni, a kódgenerálási mintában meghatározott forráskódrészleteket összeilleszteni, miközben a platformszolgáltatásokra hivatkozásokat illeszthet be.

3. fejezet

A hierarchikus monitorozás koncepciója

A monitorozó rendszerek egy rendszer működését vizsgálják és ellenőrzik, hogy megfelel-e a rendszer specifikációjának. A monitorozott rendszer általánosan lehet program, hardver vagy akár egy hálózat is. A monitorozás legtöbb esetben a funkcionális viselkedést vizsgálja [20], de előfordulnak kivételek, amikor például a rendszer teljesítményét ellenőrzi a monitor – ilyen például a Falcon eszköz, amely nagyméretű párhuzamos programok on-line monitorozására alkalmas [24].

3.1. Irodalmi áttekintés

A publikált irodalomban sok különböző megoldás található futásidejű verifikációhoz monitorok előállítására. Munkám részletes bemutatása előtt szeretnék röviden beszámolni a már megtalálható megoldásokról. Irodalomkutatásomban nagy segítségemre volt a monitorozás megoldásait leíró [20] írás illetve a NASA kutatói által írt, a monitorozás jelenét és jövőjét összefoglaló cikk [23].

Az irodalomban a monitorozó rendszerek alapvetően két csoportba sorolhatóak: *on-line* illetve *off-line* monitorokra. Az off-line monitorozás során az adatokat a monitorozott komponensről (*system under observation*, SUO) a monitor begyűjti, majd az adatok analízisét a futás után végzi csak el. Ennek megfelelően a hibára csak utólag tudja egy ilyen rendszer felhívni a figyelmet, amely biztonságkritikus alkalmazások esetén nyilvánvalóan nem elégséges. Az előnye azonban az ilyen monitorozó rendszereknek, hogy lényegesen kevesebb erőforrást igényelnek. Ezzel a megközelítéssel leginkább a monitorozással kapcsolatos irodalom korai munkái foglalkoznak: [42] valós idejű rendszerek esetén rögzíti a végrehajtási szálakat és visszajátszhatóvá teszi azt az analízis számára, míg [21] elosztott rendszerek és azok megosztott változóinak analizálását mutatja be.

Az utóbbi időben a rendszerek erőforrás kapacitásai jelentősen megnőttek és így akár beágyazott rendszerekben is lehetségessé vált az on-line monitorozás. Az on-line monitorok is két fő csoportra bonthatóak, annak függvényében, hogy az információk kiértékelése hol történik. Az *in-line* monitorozó rendszerek esetében a program kódjába illesztjük az ellenőrző utasításokat. Egy korai realizációja ennek a megoldásnak az Anna (*annotated Ada*) nyelv, ahol az Ada nyelvet bővítették ki, hogy különféle annotációk segítségével illeszthessünk be követelményeket a kódba, amelyeket a végrehajtás során a rendszer ellenőrizni tud [41]. Szintén ide sorolható a Java 5 *assertion* funkciója valamint a JML (*Java Modelling Language*) alapú futásidőbeli ellenőrzés is.

Abban az esetben, ha az ellenőrzések egy külön folyamatban történnek, akkor *out-line* monitorozásról beszélhetünk. Ilyen megoldást mutat be a [35] írás, amely COTS (*commer-*

cial off-the-shelf) komponensek monitorozását teszi lehetővé. Ilyen megoldásnak minősíthető az általam bemutatott megoldás is, hiszen az alkalmazás kódját csak instrumentáló kódokkal egészítem ki, de az információk kiértékelése egy külön monitor komponensben történik meg. Gyakori, hogy a két koncepciót vegyítik, azaz bizonyos (egyszerűbb) ellenőrzéseket a programban (in-line) végeznek el, míg bizonyos ellenőrzéseket egy külön folyamat végez el. Az általam prezentált megoldás ötvözve a korábbi munkámmal [40] szintén képes ilyen működésre, hiszen a kódgenerátor alkalmazás képes invariáns– és intervallumellenőrzéseket illeszteni a generált kódba.

A monitor szintézis fő kihívása, hogy hatékony monitorokat állítsunk elő egy magas szintű leírásból. A korábbi publikációkban rendkívül sok írást lehet találni az LTL formalizmusból történő monitor szintézisre. Arafat és társai [8]-ban egy háromértékű szemantikát mutatnak be az LTL formalizmus időzített változatához, majd ehhez monitorokat szintetizálnak. Az LTL formalizmusnak különféle változataihoz is készítettek futásidejű verifikációs megoldásokat. Ezek közül kiemelném Pintér és Majzik munkáját [36], ahol az UML nyelv állapotgépeihez definiáltak egy SC-LTL formalizmust és a megoldásuk segítségével ehhez generálható verifikációs modul.

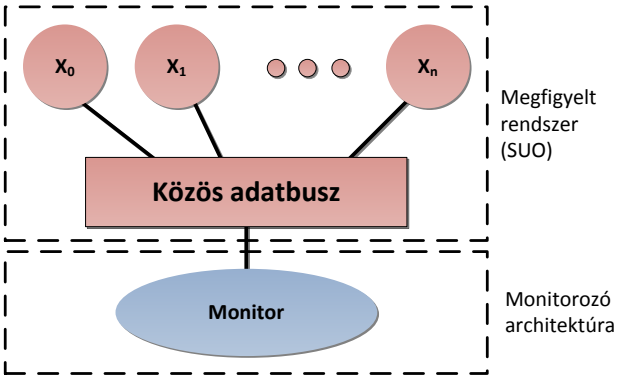
Különféle megoldások léteznek monitor szintézisre az LTL formalizmustól eltérő követelmény specifikációkra is. A Monitoring and Checking (MaC) keretrendszer Java nyelven írt valós idejű rendszerek ellenőrzését teszi lehetővé [29] olyan módon, hogy a követelményeket a Meta Event Definition Language (MEDL) nyelv egy változatában (PEDL) lehet leírni. A PEDL segítségével a program eseményeit lehet leírni absztrakt módon. Egy érdekes megközelítés továbbá a PathExplorer (PaX) keretrendszer [26], ahol szintén Java programokon lehet futásidejű verifikációt végezni. A PaX sajátossága, hogy képes hibaminta analízisre, azaz felismer olyan programozási mintákat, amelyek hibához vezetnek. Szintén egyedi, hogy a PaX követelményspecifikáló formalizmusa nem kötött, lehetőség van egyedi formalizmusok definiálására a Maude nyelv [17] segítségével.

3.1.1. Elosztott rendszerek monitorozása

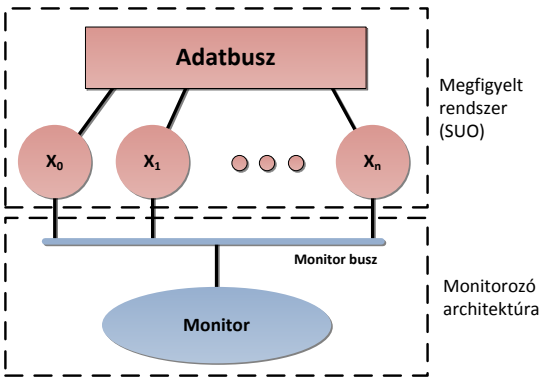
A korábban bemutatott monitorozó megoldások alapvetően monolitikus alkalmazásokhoz készültek. Természetesen az elosztott rendszerek monitorozásának is megtalálható a saját irodalma. Az 1992-ig történő kutatásokat Mansouri-Samani illetve Sloman foglalja össze írásában [33]. Bauer és társai egy olyan megoldást publikáltak [10], ahol komponens szintű monitor modulok elemzik a rendszer működését, és hiba esetén egy központi egységgel kommunikálva képesek a rendszert egy biztonságos állapotba irányítani. Bhargavan és társai a TCP protokoll elosztott monitorozásával foglalkoztak [14]-ben.

A *monitorozási architektúra* egy, vagy több monitor integrációja a megfigyelt rendszerrel (SUO). Ez az architektúra magában foglalja az összes hozzáadott hardvert és kapcsolatot a rendszerben. Valós idejű, hibatűrő rendszerek monitorozásához néhány alapvető megkötést kell tennünk az architektúrára, amelyet [23] ír le összefoglalva:

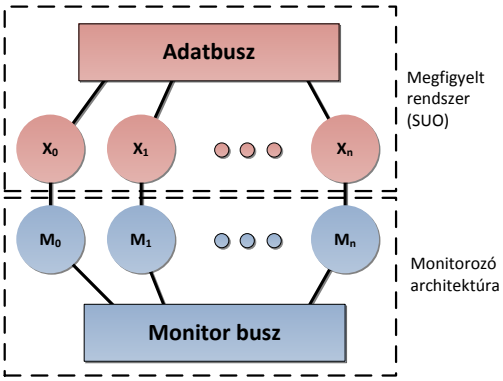
- *Funkcionalitás:* a monitor nem változtathatja meg a SUO funkcionalitását, kivéve akkor, ha az megsérti a specifikációt.
- *Időzítés:* a monitor architektúra nem okozhatja, hogy a SUO megsértse a valós idejű követelményeket, kivéve, ha a SUO megsérti a specifikációt.
- *Megbízhatóság:* a SUO megbízhatósága a monitorozó architektúrában nagyobb vagy egyenlő kell, hogy legyen, mint a különálló SUO megbízhatósága.
- *Tanúsíthatóság:* a monitor architektúra rendszerhez illesztése nem igényelhet a SUO forráskódjában akkora módosítást , amely szükségessé tenné az alkalmazás tanúsításának ismételt elvégzését.



(a) Egy monitor osztott adatbuszon



(b) Dedikált monitor busz, egy monitorral



(c) Dedikált monitor busz, több monitorral

3.1. ábra. A [23]-ban ajánlott monitorozó architektúrák

Ezeket az alapelveket munkám során figyelembe vettem és a dolgozatban bemutatott monitorozó rendszer megfelel ezeknek a követelményeknek amennyiben a monitorozáshoz szükséges forráskód kiegészítések miatt nem sérülnek az eredeti határidők (valós idejű követelmények).

A már említett [23] írás a monitor architektúrákat három alapvető csoportba sorolja (lásd 3.1. ábra), amelyeket most szeretnék röviden bemutatni.

- Az első az *egy monitor, osztott adatbuszon*, amely esetén a monitor modul a rendszer egyetlen, közös adatbuszára van kötve, és ugyanolyan módon lehet neki üzenetet küldeni, mint tetszőleges egyéb komponensnek. A monitor azonban ezen a buszon nem küld adatot (kivéve rendszerhiba esetén), kizárólag fogadja az üzeneteket. Ez tekinthető a legegyszerűbb architektúrának és ennek megfelelően ez a megoldás képes a legkevesebb hibát detektálni. A monitornak a küldött üzenetek alapján kell a komponensek belső állapotára következtetni, ami rendkívül bonyolult feladat. Az előnye az architektúrának, hogy tetszőleges COTS komponens illeszthető a rendszerbe, akár olyanok is, amelyek nincsenek felkészítve a monitorozásra.

- Egy közös busz könnyen problémákat jelenthet. Ezt hivatott megoldani a második architektúra, amely *dedikált monitor busszal* rendelkezik, de *egyetlen monitor komponensből* áll a monitorozó architektúra. Ebben az esetben minden alkalmazást fel kell műszerezni, hogy információkat küldjön a monitor számára. Az instrumentáló kódok nem lehetnek túl bonyolultak, hiszen abban az esetben nem állna meg a *tanúsítha-*

tóság megkötés. A dolgozatban bemutatott rendszer szintű monitorozó megoldások (5. és 6. fejezet) ezt az architektúrát valósítják meg.

- Az utolsó architektúra az előző tovább bővített változata, ahol *dedikált monitor buszon több monitorozó komponens* áll rendelkezésre. Minden komponens számára egy különálló monitor modul készül, amely csak az adott komponens hibáit képes detektálni. A rendszer elosztottan működik, és bármely komponens hibája esetén a teljes rendszert értesíteni kell. A hierarchikus monitorozó rendszer komponens hierarchia szintje (4. fejezet) felel meg ennek az architektúrának.

3.2. A monitor szintézis alapjai

A futásidejű verifikáció célja, hogy a rendszer működése közben detektálja a rendszerben bekövetkező hibákat. Ehhez monitorozó komponensek előállítására van szükség. Az általam kidolgozott monitorozó rendszer képes a monitorok automatikus szintézisére, amelyhez az alábbi referencia információkat használja fel:

- Az időzített automata modell, amely a vezérlőkben implementált alkalmazást írja le. Ez a modell leírja, hogy milyen állapotok és állapotváltozások lehetnek a rendszerben, illetve a rendszer komponensei közötti szinkronizációkat is magában foglalja. Az időzített automata modell a rendszer időzítéseit is definiálja az állapotokhoz rendelt időinvariánsok és óraváltozók segítségével.

- Biztonsági és élőségi kritériumok temporális elérhetőségi kifejezésekként specifikálva a TCTL formalizmus segítségével. A TCTL kifejezések hivatkozhatnak egy automatára vagy a teljes rendszerre vagy annak egy részhalmazára.

- A rendszer viselkedését leíró forgatókönyvek is specifikálhatják a rendszer helyes működését, legfőképpen az automaták közötti interakciókat. Az UPPAAL eszköz által támogatott LSC forgatókönyvek szintén a bemenetét képezhetik a monitor szintézis folyamatnak.

Az előállított monitorok futásidőben különböző információkat igényelnek a működő rendszertől az ellenőrzések végrehajtásához. A vezérlők programkódját úgy kell módosítani, hogy ezeket az információkat elküldjék. Ezt a folyamatot felműszerezésnek nevezzük és részletesebben a 3.4. fejezetben mutatom be. A monitorok által igényelt információk minden esetben eltérőek. Lehetnek állapotinformációk, változók értékei vagy más egyéb információk. A későbbi fejezetekben, ahol az egyes megoldásokat bemutatom a szükséges információkat is részletesen leírom.

3.3. Monitor hierarchia

A bemutatott bemenetek alapján különféle monitor komponensek definiálhatóak, amelyek egy hierarchikus rendszert alkotnak együttesen. Két hierarchia szint definiálható a monitorokhoz:

- *Komponens szint:* Lehetőség van komponensenkénti monitor szintézis elvégzésére. Ebben az esetben a monitorok kizárólag a komponensek szintjén képesek a hibák detektálására, így nem képesek észlelni, ha például a rendszer egészében nem teljesülnek az időzítési feltételek, vagy ha a szinkronizációk közben lép fel valamilyen hibajelenség. Az így szintetizált monitorok a 3.1(c). ábrán bemutatott architektúrának felelnek meg.

- *Rendszer szint:* Szintén lehetséges a rendszer egészéhez monitort szintetizálni. Ebben az esetben a monitor minden információt begyűjthet a rendszer összes komponensétől és így képes a rendszer egészét érintő követelmények ellenőrzésére. A rendszer szintű monitor képes a rendszer szinkronizációinak illetve a teljes rendszer időzítésének ellenőrzésére. Ennek a szintnek a 3.1(b). ábrán bemutatott architektúra felel meg.

A rendszer szintű monitorozás szükségességét jól mutatja a következő, 2005-ben megtörtént eset. Egy Airbus A340-642 kódszámú utasszállító február 8-án kényszerleszállást végzett [15]. Az utólagos vizsgálatok alapján a vészhelyzetet az okozta, hogy a hajtóművekhez tartozó tartályokban nem volt megfelelő mennyiségű üzemanyag. Az üzemanyag pumpálását végző két FCMC (*Fuel Control Monitoring Computer*) összehasonlítja egymás eredményét, majd eldöntik, hogy melyik eszköz kimenete helyes és az hajthatja meg a kimeneti adatbuszokat. A konkrét esetben mindkét FCMC több hibával is rendelkezett, de még a repülésre alkalmasak voltak. A kiválasztott eszköznek sajnos a kimeneti buszt meghajtó komponense is hibás volt, így az üzemanyag pumpák nem kaptak a vezérlőktől semmilyen információt, annak ellenére, hogy az FCMC helyes vezérlési információkat küldött. A biztonsági rendszerek nem aktiválódtak, hiszen csak azt figyelték, hogy ne legyen mindkét komponens egyszerre kiesve. Amennyiben alkalmaztak volna rendszer szintű monitorozást, az képes lett volna detektálni a problémát, azaz hogy nem jutnak érvényre a vezérlő által kiadott utasítások.

Természetesen minden komponens szinten elvégzett ellenőrzés elvégezhető lenne a magasabbik szinten is. Ami mégis a komponens szintű monitorok mellett szól az, az a tény, hogy az ilyen monitorok kizárólag a komponenstől gyűjtenek információt. Ennek következményeként ezek a monitorok közel helyezhetőek a monitorozott egységhez. Ez csökkentheti a teljes rendszer komplexitását, illetve jelentősen csökkentheti a monitorozáshoz szükséges információk küldésének költségét, ezáltal kisebb többletterhelést okozva a rendszernek. A monitorok közelségéből az is adódik, hogy a monitorok az információt kisebb késleltetéssel érhetik el, ami a hiba hamarabbi detektálását jelentheti, amely biztonságkritikus alkalmazásoknál nagyon fontos.

3.3.1. A monitorozás által detektálható hibák

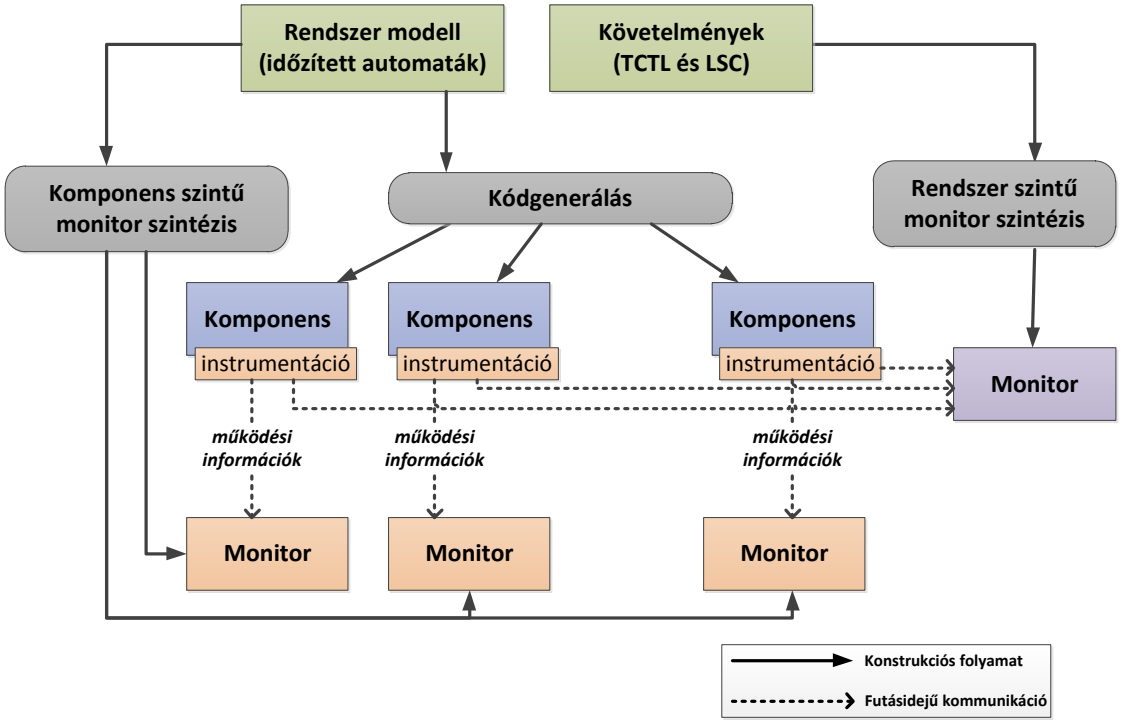
Munkám során megvizsgáltam, hogy az elosztott beágyazott rendszerek esetén gyakran fellépő meghibásodások milyen hibajelenségeket okoznak. A hierarchikus monitorozás segítségével a hibás rendszerállapotokat különböző szinteken lehet detektálni. Ehhez elsőként a rendszer hibamodelljét építettem fel az alábbi módon:

- A rendszerben futás közben fellépő tranziens meghibásodások nagy többsége az utasítások végrehajtási szekvenciájában okoz anomáliákat. Ezeket a hibákat *vezérlési folyam hibáknak* nevezzük. Az ilyen hibák detektálásához a rendszernek minden állapotváltozásáról információt (*állapotjelzőszámokat*) kell küldenie a monitor számára.

- A komponensek a rendszerben véletlen hardverhiba esetén teljesen meghibásodhatnak, illetve megszakadhat a kapcsolatuk a rendszerrel. A komponensek kiesését vagy a kapcsolat megszakadását előírt periodikus üzenetküldés (életjel) kimaradása alapján lehet detektálni.

- A rendszer komponensei szinkronizációs hibák miatt egymással inkonzisztens állapotba kerülhetnek, amely a komponens szintjén nem jelentkezik hibaként, de a rendszer egészében mégis hibajelenséget okozhat.

A fenti hibamodell alapján egyértelműen meghatározhatóak, hogy az egyes hierarchia szinteknek milyen felelőssége van a monitorozás során.



3.2. ábra. A monitorszintézis áttekintése

- A komponens szintű monitorok felelőssége a komponens vezérlési folyamának ellenőrzése, a vezérlő időinvariánsainak teljesülésének ellenőrzése valamint a komponens életjelinformációinak vizsgálata. A monitorok a rendszert leíró időzített automata modellből generálhatóak.
- Komponens szinten kell ellenőrizni az olyan temporális logikai kifejezéseket, amelyek kizárólag egy automata változóira és állapotaira hivatkozik. Az ilyen monitorok a rendszert leíró TCTL követelményekből szintetizálhatóak.
- A rendszer szintű monitorozó komponensek felelőssége a teljes rendszert érintő temporális logikai kifejezések ellenőrzése. Azokból a TCTL kifejezésekből, amelyek a rendszer legalább két automatájának változóit vagy állapotait tartalmazzák, rendszer szintű monitorok szintetizálhatóak.
- A rendszer viselkedését specifikáló LSC forgatókönyvek alapján szintén rendszer szintű monitorozó komponensek állíthatóak elő, amelyek a teljes rendszer viselkedését specifikálják - legfőképpen a komponensek közötti interakciókat.

A különböző szinteket és azok szintetizálását hivatott áttekinteni a 3.2. ábra. A monitorozó komponensek szintetizálásának részletes bemutatása a következő fejezetekben lesz olvasható.

3.4. Programkód és monitor kód generálás összehangolása

A futásidejű verifikációhoz a monitorok különböző információkat igényelnek a rendszertől működés közben. Ezt megszerezhetnék egy-egy komponens illetve a rendszer elsődleges kimeneteit (jeleit, kommunikációit) figyelve, azonban így a komponensek belső állapotára csak nehezen és sok esetben pontatlanul tudnának következtetni, ezzel meghenezítve

a gyors és nagy hibafedésű hibadetektálást. Ehelyett a komponensek explicit elküldik a monitor számára ezeket az információkat. Ehhez azonban módosítani kell az eredeti programot, azaz az eredeti alkalmazást fel kell műszerezni, más szóval *instrumentálást* kell végezni.

Ez COTS komponensek esetén ez csak bizonyos költségekkel és korlátozásokkal lenne megvalósítható (bináris vagy bájtkód szintű program módosítási technikák segítségével), hiszen a komponensek forráskódja nem elérhető, de a 2.3. fejezetben bemutatott automatikus kódgenerátor használata esetén ez megoldható, a kódgenerálási minta bővítésével. A kódgenerátor különböző absztrakt szolgáltatásdefiníciókkal rendelkezik (lásd a 2.3. fejezetet), amelyek tovább bővíthetőek. Mivel a monitorozási üzenet elküldése jól illeszthető a kódgenerátor szemantika leképzéséhez (hiszen a monitorozó üzenetek állapotokhoz és szinkronizációkhoz köthetők), így az új platformszolgáltatások definiálása viszonylag egyszerű feladat. A monitorozás számára különféle üzenetküldési szolgáltatásokat definiáltam illetve ezeket a kódgenerálási mintába is beillesztettem.

Munkám egyik újdonsága, hogy a program kód illetve a monitor kód generálása összehangolt módon történik. Ennek előnye, hogy a felműszerezést optimalizálni lehet az ellenőrzendő kritériumokra, ezzel csökkentve a rendszer többletterhelését. A munkámban bemutatott monitor szintetizáló eszközök, illetve a hozzá tartozó automatikus program instrumentáló megoldások minden esetben a lehető legkevesebb számú üzenetet küldenek el a monitor felé. Ezen megoldásokat az egyes technológiák ismertetésénél fogom részletezni.

3.5. Használati esetek

A futásidejű verifikációs megoldások általában *egy konkrét lefutását* elemzik a rendszernek. A *lefutás* alatt a rendszer bekapcsolása és kikapcsolása (vagy újraindítása) közt bejárt állapotszekvenciát értem. A klasszikus megoldások LTL formalizmussal specifikált kritériumokat ellenőriznek, hiszen egy LTL kifejezés egy-egy lefutásra ad követelményeket. Az általam használt CTL formalizmus azonban elágazó idejű temporális logika, azaz egy kifejezés képes lefutások sorozatára illeszkedni. Ennek megfelelően a hagyományos futásidejű verifikációs megoldásoktól némileg eltér a felhasználhatósága az eszköznek. Két használati esetet mutatok be a következőkben.

Az első, a már sokat emlegetett *futásidejű verifikáció*, amelynek esetében a rendszer fejlesztési folyamata után végzünk ellenőrzéseket. Ezáltal a rendszer biztonságosabbá tehető, hiszen a rendszerspecifikációktól való eltérés detektálható és a hibákat kezelő algoritmusokat lehet elindítani amennyiben szükséges.

A második használati eset a CTL formalizmusból adódóan a *tesztelés*. Erre az ad lehetőséget, hogy a lefutások sorozatára tudunk kritériumokat specifikálni, így a tesztkészlet különböző tesztesetei külön-külön lefutásokként jelenhetnek meg és a kritériumok együttesen ellenőrizhetőek rajtuk. A CTL formalizmus rendelkezik egy adott állapotból induló lefutások halmazán definiált úgynevezett útvonal kvantorokkal, amely segítségével például az vizsgálható, hogy egy meghatározott állapot a rendszer lefutásai között legalább egy esetén fennáll-e. A monitorozó komponenseket a tesztelés folyamatában a tesztelést kiértékelő *teszt orákulumként* lehet felhasználni.

4. fejezet

Komponens szintű monitorok szintézise a vezérlési folyam ellenőrzésére

A 3.3. fejezetben bemutatott hierarchia szintek közül elsőként a komponens szintű megoldásokat szeretném tárgyalni. Ahogy korábban írtam, a vezérlési folyam ellenőrző monitorok segítségével a tranziens hibák nagy részét lehet kiszűrni.

4.1. A referencia információ

A vezérlési folyam ellenőrzéséhez elengedhetetlen, hogy az ellenőrző komponens számára rendelkezésére álljon a program állapotok megengedett sorozata, azaz minden állapot esetén kideríthető legyen, hogy az megengedett rákövetkezője-e a program előző állapotának. Az ellenőrzéshez szükséges referencia információt a szintetizált monitorna kell tárolnia.

A megfigyelt rendszer forráskódját automatikus kódgenerálás állítja elő, amelynek alapja az időzített automaták hálózata, azaz $\mathcal{A}_1, \mathcal{A}_2 \ldots \mathcal{A}_n$ időzített automaták kompozíciója. A komponens szintű ellenőrző monitorok egy-egy $\mathcal{A}_i$ időzített automata komponenshez készülnek, így a referencia információt, azaz a komponens állapotainak megengedett sorozatait, az időzített automata modell tartalmazza. A monitorgenerálás során a komponens szintű monitorokat tehát a komponens formális időzített automata leírásából lehet előállítani. A monitor szintézis eszköz bejárja ezt a $\mathcal{A}_i$ modellt és a következő információkat nyeri ki belőle:

- N_i a komponens állapotainak véges halmazát,
- az l_{0_i} kezdőállapotot,
- E_i, a komponens állapotátmeneteinek véges halmazát, illetve
- I_i, a komponens időinvariánsait meghatározó függvényt.

Ezek segítségével meghatározható, hogy futás közben milyen állapotsorrend jelent megengedett működést (ahol a modellben található állapotátmenetek mentén történik a program végrehajtása), és milyen állapot esetén kell hibát jelezni (amikor az aktuális állapot nem érvényes rákövetkezője az előző elfogadott állapotnak, mivel nincs közöttük a modellben állapotátmenet). Az I_i függvény segítségével az állapotokhoz rendelt időinvariánsok is rendelkezésre állnak, így ezek is ellenőrizhetőek lesznek a komponens szintű monitorban.

4.2. Az ellenőrzés algoritmusa

Az előzőleg bemutatott referencia információk segítségével elvégezhetőek a szükséges ellenőrzések. Az 1. algoritmus az ellenőrző logika pszeudokódját mutatja be. A fejezet további részében részletesebben bemutatom az ellenőrzéseket.

Algoritmus 1: Vezérlési folyam ellenőrző monitor kiértékelő logikája

bemenet: m_i : monitorozó üzenet

```
1  if m_i küldője nem a monitorozott egység then
2  |  üzenet eldobása()
3  |  return
4  end
5  if m_i sorszáma ≠ m_{i-1} sorszáma then
6  |  hiba jelzése(üzenetvesztés történhetett)
7  |  return
8  end
9  Jelzőszám d ← m_i jelzőszáma
10 if m típusa állapotváltási üzenet then
11 |  if d ∈ aktuális állapotból kilépő állapotváltozások jelzőszámai then
12 |  |  if aktuális állapothoz van invariánsfeltétel then
13 |  |  |  if aktuális állapot invariánsfeltétele nem teljesül then
14 |  |  |  |  hiba jelzése(invariánsfeltétel nem teljesült)
15 |  |  |  |  return
16 |  |  |  end
17 |  |  end
18 |  |  aktuális állapotátmenet ← d
19 |  else
20 |  |  hiba jelzése(nem megengedett állapotátmenet)
21 |  |  return
22 |  end
23 else if m típusa állapot üzenet then
24 |  if d ≠ aktuális állapotátmenet célja then
25 |  |  hiba jelzésenem megengedett állapot
26 |  |  return
27 |  end
28 |  aktuális állapot ← d
29 end
```

4.2.1. Vezérlési folyam ellenőrzés

A vezérlési folyam ellenőrzése során a kiértékelő modul a monitorozott komponenstől állapotjelzőszámokat illetve állapotátmenet jelzőszámokat kap, amelyek egyértelműen azonosítják a futás közben elért állapotot illetve állapotátmenetet. Ezeket a jelzőszámokat összehasonlítja a referencia modellben tárolt jelzőszámokkal (állapot illetve állapotátmenet azonosítókkal). A monitor tudja, hogy aktuálisan – az utolsó információi szerint – melyik állapotban volt a komponens, illetve ismeri az állapotból lehetséges állapotátmeneteket is. Ezek alapján egy állapotátmenet jelzőszám alapján el tudja dönteni, hogy a modell által megengedett vezérlési folyamnak megfelelő-e a megfigyelt komponens működése. Az

állapotjelzőszámok nagyon hasonlóan vizsgálhatóak. Azért szükséges mind az állapotokat azonosító jelzőszámokat, mind az állapotváltozások azonosítóit elküldeni, mert a lefutás pontos vezérlési folyama csak így lesz reprodukálható *(például diagnosztikai célokra)*, hiszen előfordulhat, hogy két állapot között két átmenet is található a modellben, de ezeken eltérő utasítások találhatóak, így fontos a két különböző átmenetet megkülönböztetni.

4.2.2. Invariánsok ellenőrzése

Egy fontos kiegészítés az állapotinvariánsok ellenőrzése, amely segítségével a komponens időzítéseit lehet ellenőrizni a komponens szintű monitorban. Ehhez, ahogy előzőleg bemutattam, a monitor eltárolja az I_i függvényeket, amely az állapotokhoz invariánsfeltételeket rendelnek. Ezek után, ha a rendszer egy olyan állapotba kerül, amelyhez van feltétel rendelve, akkor a monitor az állapotból történő kilépés során egy ellenőrzést végez. Az ellenőrzés nem blokkolja a rendszer továbbfutását, így a hibák csak utólag detektálhatóak. Ehhez természetesen az állapotinformációkon felül a monitornak további futásidejű információkra van szüksége. Szükséges az invariánsfeltételben megtalálható változók illetve óraváltozók aktuális értéke. Ehhez az alkalmazást további instrumentáció kóddal kell ellátni, amelyről a 4.4. fejezetben írok részletesebben. Az itt bemutatott megoldás kizárólag az invariánsfeltételek megsértésének utólagos detektálására alkalmas, nem képes semmilyen módon azokat előre jelezni, vagy megelőzni azokat.

4.2.3. Életjelek ellenőrzése

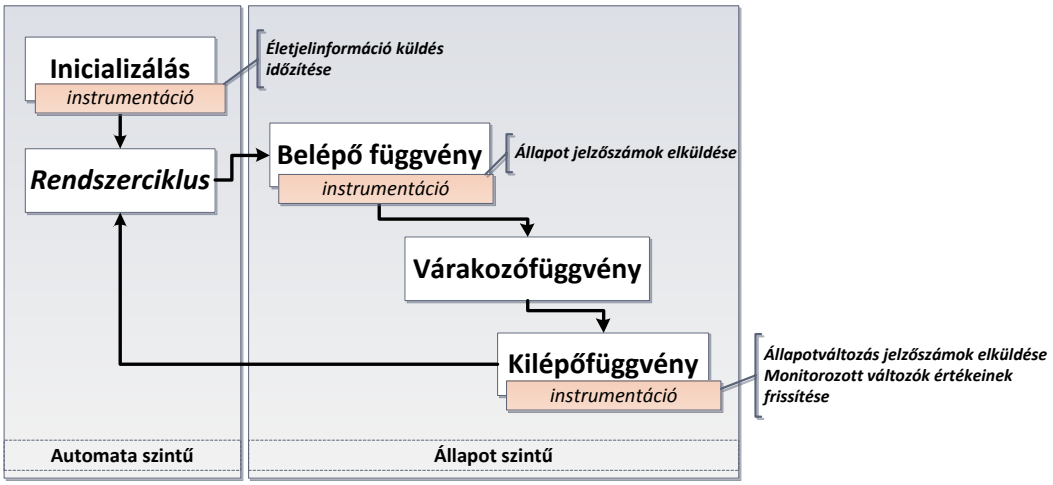
További fontos monitorozandó információ a komponens életjeleinek vizsgálata. A rendszer számára elengedhetetlen információ, ha egy komponense meghibásodik, azaz „kiesik a rendszerből". Az észlelést nehezíti, hogy az egy normális jelenség a működés során, hogy hosszabb ideig várakozik egy állapotban (például szinkronizációra várakozik) és eközben semmilyen üzenetet nem küld. Ennek kezelésére vezettem be azt a megoldást, hogy a komponensek periodikusan üzeneteket küldenek a monitor felé, jelezve ezzel, hogy nem hibásodtak meg. Egy adott időn túl kimaradó életjel üzenet utal tehát a komponens kiesésére.

4.3. A monitor struktúrája

A monitor kódja alapvetően két részre bontható (lásd B függelék). Az első rész tartalmazza az ellenőrzésekhez használandó referencia időzített automatát egy adatstruktúrában. Az adatstruktúra tartalmazza az automata összes állapotát és állapotátmenetét, mindegyik esetén egyedi *jelzőszámokkal*, amelyek a komponens felől érkező különféle monitorozó üzenetek feldolgozásánál szükségesek az azonosításhoz. A monitor szintén tárolja az invariánsfeltételeket is. Minden egyes invariáns ellenőrzéshez egy külön függvény generálódik, amelyre egy hivatkozást tárol a megfelelő állapotnál a monitor.

A második rész független az időzített automatától. Ez a rész tartalmazza a monitor működését leíró algoritmusokat. Az algoritmus a bejövő üzeneteket dolgozza fel. Megvizsgálja a beérkezett jelzőszámot, majd összeveti a referencia modellel, hogy a vezérlési folyamnak megfelel-e a kapott érték. Amennyiben nem, akkor a komponens szintű monitor rendszerhibát jelez. Az 1. algoritmus ezt mutatja be.

A két rész – a vezérlő forráskódját generáló eszközhöz hasonlóan – a Java Emitter Templates keretrendszer felhasználásával készült. A monitor szintézise során a rendszer feldolgozza a konkrét időzített automata modell reprezentációt, majd létrehozza a modellt tároló adatszerkezetet, illetve a kiértékelő logikát.



4.1. ábra. A komponens szintű monitorozáshoz szükséges felműszerezés pontjai a kódstruktúrában

4.4. A program felműszerezése

A vezérlők automatikus és a követelményekre optimalizált felműszerezését a kibővített kódgenerátor eszköz végzi. A felműszerezés (mint forráskód bővítés) olyan platformszintű szolgáltatásokat használ, amelyek a komponens szintű monitor modulnak küldenek üzeneteket működés közben.

A vezérlési folyam ellenőrzése meglehetősen erőforrásigényes feladat (ezért célszerű ezt rendszer szint helyett komponens szinten végezni), hiszen minden állapotról és állapotváltozásról értesíteni kell a monitort. Az instrumentálás során ehhez szükséges volt az állapotok belépő illetve kilépő függvényeit módosítani, olyan módon, hogy ezek hívják meg a megfelelő platformszolgáltatást. Az előző fejezetben bemutatott néhány kiegészítés szintén további instrumentációt igényel a vezérlő kódban. Az életjel információk küldéséhez az alkalmazás inicializációs fázisában egy időzítőt kell létrehozni, amely megadott periódusidővel üzeneteket küld a monitor számára. Némileg bonyolultabb feladat az időinvariánsok ellenőrzése, hiszen ehhez a monitornak bizonyos változók – különösen az óraváltozók – aktuális értékét el kell érniük. Éppen ezért a rendszer minden esetben, amikor megváltoztat egy olyan változót, amely értéke szükséges a monitor számára – tehát olyan változót, amely szerepel időinvariáns kifejezésekben – annak frissített értékét egy üzenetben a monitor számára eljuttatja. Az óraváltozók értéke folyamatosan változik, így ennek elküldését az állapot elhagyásakor kell elküldeni - még mielőtt a monitor elvégezné az invariáns ellenőrzést. A felműszerezés pontjait a 4.1. ábra mutatja be. Röviden összefoglalva a komponens szintű monitor az alábbi futásidejű információkat kaphatja a rendszertől:

- Állapotinformációk
- Állapotváltozás információk
- Életjelinformációk
- Változók aktuális értéke

5. fejezet

Rendszer szintű monitorok szintézise temporális logikai követelmények alapján

A 3.3. fejezetben részletesen bemutattam, hogy mi indokolja a rendszerszintű monitorozás szükségességét. Ennek első konkrét technológiája a temporális logikai kifejezésekkel specifikált rendszerkövetelmények monitorozása.

A fejezetben először bemutatom az UPPAAL által használt TCTL variánst, amely a monitor referencia információját, azaz az ellenőrizendő követelményeket adja. A munkám során az UPPAAL-ban támogatott formalizmust igyekeztem teljes mértékben felhasználni a monitor szintézis alapjaként, amely sok pontban eltér a klasszikusan monitorozáshoz használt LTL formalizmustól. Ezután bemutatom a monitorok kiértékelő modulját, amely az egyes TCTL operátorokból a munkám során képzett elfogadó automatákat implementálja. Végezetül röviden összefoglalom, hogy a vezérlők forráskódjának felműszerezése milyen feladatokból áll.

5.1. A referencia információ: TCTL-ben formalizált követelmények

Az UPPAAL-ban használt TCTL formalizmus variáns a CTL-hez képest néhány egyszerűsítést tartalmaz [11]. A munkám során az UPPAAL által támogatott készletet használtam fel, így a következőekben pontosan definiálom a különbségeket.

Az UPPAAL által támogatott TCTL formalizmus a CTL formalizmustól az alábbi pontokban tér el:

- Temporális operátorokat tartalmazó kifejezések egymásba ágyazása nem lehetséges. Ez azt jelenti, hogy az AG, AF, EG, EF temporális operátorok a formula elején találhatók, ezek után pedig Boole logikai operátokkal összekapcsolt, további temporális operátort nem tartalmazó állapotkifejezések következhetnek.
- Az X és U állapotoperátorok nem támogatottak.
- Létezik egy speciális temporális operátor: $p \rightsquigarrow q$, amely az alábbi kifejezéssel egyezik meg: $AG(p \Longrightarrow AFq)$ – erre az egymásba ágyazás tiltása miatt van szükség. Jelentése, hogy az adott állapotból kiinduló minden végrehajtási útvonalon igaz, hogy ha p teljesül, akkor utána minden útvonalon előbb-utóbb q is teljesülni fog.
- Az állapotkifejezések lehetnek a következőek is:

  - Bármilyen mellékhatás-mentes kifejezés.
  - Egy automata adott állapota (annak azonosítójával megadva).
  - A *deadlock* kifejezés, amely a rendszer holtpontját írja le.

- Ha p és q állapotkifejezések, akkor a $p \Longrightarrow q$ (vagy p *imply* q) is állapotkifejezés.
- Az állapotkifejezésekben használhatóak óraváltozók, órakifejezések.
- Az állapotkifejezésekben használhatóak az automaták lokális, illetve a rendszer globális változóinak értéke.

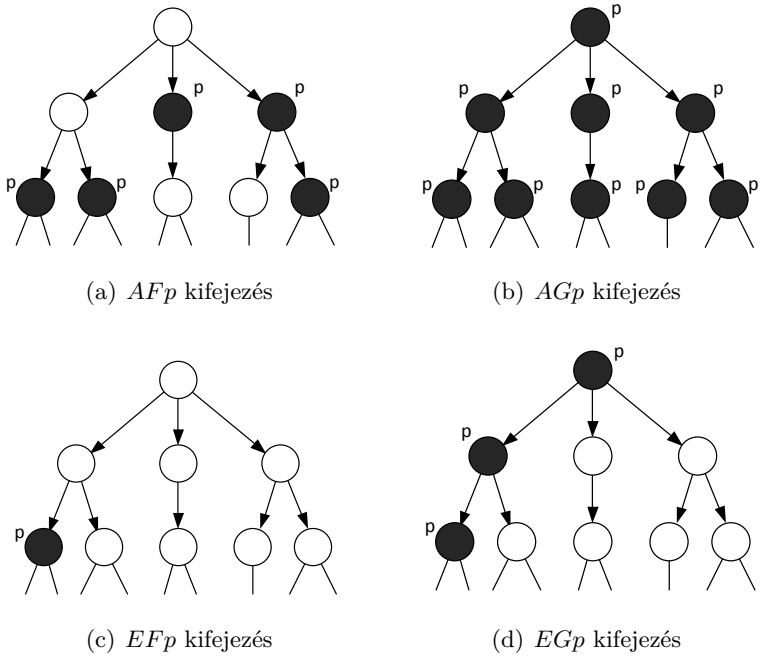
Az UPPAAL követelményleíró nyelvében használható temporális operátorokat az 5.1. ábra szemlélteti. A következőkben megadom a monitor szintézis által támogatott TCTL kifejezések szintaxisát:

```
Prop ::= 'AG' Expression | 'EF' Expression | 'EG' Expression
      | AF Expression

Expression ::= NAT
             |  Expression '[' Expression ']'
             |  '(' Expression ')'
             |  Expression '++' | '++' Expression
             |  Expression '--' | '--' Expression
             |  Expression Assign Expression
             |  Unary Expression
             |  Expression Binary Expression
             |  Expression '?' Expression ':' Expression
             |  Expression '.' ID
             |  Expression '(' Arguments ')'
             |  'deadlock' | 'true' | 'false'

Arguments  ::= [ Expression ( ',' Expression )* ]

Assign     ::= '=' | ':=' | '+=' | '-=' | '*=' | '/=' | '%='
             | '|=' | '&=' | '^=' | '<<=' | '>>='
Unary      ::= '+' | '-' | '!' | 'not'
Binary     ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
             |  '+' | '-' | '*' | '/' | '%' | '&'
             |  '|' | '^' | '<<' | '>>' | '&&' | '||'
             |  '<?' | '>?' | 'or' | 'and' | 'imply'
```

Összefoglalva tehát a követelmények az alábbi információkra hivatkozhatnak (tehát a felműszerezés segítségével ezeket az információkat kell elküldeni futásidőben a monitor számára):

- Állapotinformációk
- Változók aktuális értékei – óraváltozók is
- Komponens szintű lokális holtpont információk

Az instrumentáció számára definiálható a *megfigyelt állapotok illetve változók* halmaza, amelyek azok az állapotok és változók összessége, amelyek szerepelnek egy követelményben.



(a) AFp kifejezés (b) AGp kifejezés

(c) EFp kifejezés (d) EGp kifejezés

5.1. ábra. Az UPPAAL-ban használható temporális operátorok

5.2. Az ellenőrzés algoritmusa

A temporális logikai kifejezéseken alapuló monitorok elfogadó automatákat implementálnak. Ebben a fejezetben elsőként bemutatom TCTL operátorokhoz készített automatákat, majd beszámolok a monitorok tényleges szintéziséről.

5.2.1. Az elfogadó automaták koncepciója

Az elfogadó automatákat az UPPAAL-ban használható TCTL temporális operátorokhoz készítettem el. Ezek az automaták azt a viselkedést fogadják el, amely megfelel az operátor szemantikájának. Az elfogadó automaták a kiértékelő modul kimenetét határozzák meg, amely három értékű lehet. Amennyiben a monitorozott rendszer megfelel a követelménynek, a kiértékelő elfogadott (igaz) állapotot jelez. Ha a rendszer megsérti a követelményt, akkor a kiértékelő hibát (hamis állapotot) jelez. Az elfogadó automata bizonyos állapotaiban a kiértékelő modul nem meghatározott kimenettel rendelkezik, azaz a követelmény se nem teljesült, se nem sérült és a jövőben még bármelyik eredmény adható lehet. Mivel az UPPAAL nem támogatja a temporális operátorok egymásba ágyazását, így elegendő az elfogadó automatákat külön-külön kidolgozni az egyes temporális operátorokhoz, és egymástól függetlenül vizsgálni.

A 3.5. fejezetben bemutattam, hogy a monitorok segítségével nem kizárólag egy konkrét lefutás vizsgálható, hanem lefutások halmazán is lehet ellenőrzéseket végezni, amely elsősorban a tesztelés során hasznos. Az elfogadó automatákat úgy alkottam meg, hogy képesek legyenek lefutások sorozatát kezelni. Ehhez a teszt környezet a monitor kiértékelője felé jelzéseket küld az egyes lefutások végén. A kiértékelő szintén értesül a lefutások sorozatának végéről. Tesztelési terminológiában így külön értesítéseket kap a monitor az egyes tesztesetek végén, illetve a teljes tesztkészlet végén. Amennyiben a monitort nem tesztkörnyezetben használjuk, akkor a monitor egy lefutást vizsgál – azaz úgy viselkedik,

mintha egy tesztesetből állna a tesztkészlet. Az elfogadó automatákat úgy dolgoztam ki, hogy tetszőleges számú teszteset esetén helyesen működjön.

Az elfogadó automatákban megjelölöm azokat az állapotátmeneteket is, amelyek során a rendszer (opcionálisan) eltárolja az aktuális lefutást, mint ellenpélda a követelményre. Ezek rendkívül fontos szerepet kaphatnak később a rendszer diagnosztikája során, hiszen könnyedén megfigyelhető, hogy a rendszer milyen állapotokon keresztül jutott el a követelményeknek nem megfelelő állapotba.

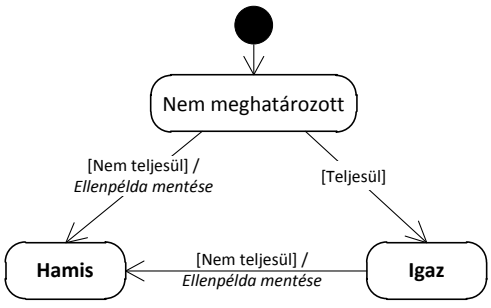
Az elfogadó automaták az alábbi eseményeket és akciókat jelenítik meg:

- Tesztkészlet kezdete
- Tesztkészlet vége
- Új lefutás
- Lefutás vége
- Az állapotkifejezés teljesül
- Az állapotkifejezés nem teljesül
- Ellenpélda tárolása (akció)

Az AG temporális operátor elfogadó automatája

Az első elfogadó automata az AG temporális operátorhoz tartozó, amely az 5.2. ábrán látható. Az automata az 5.1(b). ábrán szemléltetett útvonal halmaznak megfelelő viselkedést fogadja el. Az operátor jelentése, hogy minden esetben teljesülnie kell a feltételnek, így ez az operátor használható az *invariáns jellegű biztonsági kritériumok leírására*, mint például, hogy amennyiben a kiadott jelzéskép vörös, a szakaszon haladó vonat nem haladhatja meg a jelzőt.

A kiértékelés során az alaphelyzet eldöntetlen kimenetet ad, majd minden beérkező üzenet után a rendszer kiértékeli a követelmény állapotkifejezésének logikai értékét. Ez az érintett állapotváltozók, illetve állapot azonosítók alapján történhet meg. Ha a követelményben található állapotkövetelmény teljesül, akkor az igaz kimenetű állapotba vált, ha nem, akkor a hamis állapotba. Az igazból bármikor hamisba válthat, ha nem teljesül a kifejezés egy adott pillanatban. Ha egy kifejezés egyszer nem teljesült, akkor onnantól kezdve az automata sosem lép tovább, így a kiértékelő mindig azt fogja jelezni, hogy a követelmény nem teljesül.



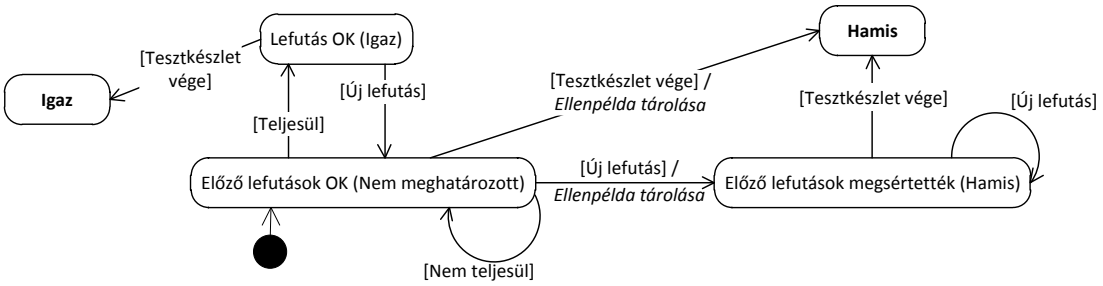
5.2. ábra. Az AG temporális TCTL operátorhoz tartozó elfogadó automata

Ha a rendszer olyan üzenetet kap, amely az elfogadó automata aktuális állapotában nincs megjelenítve, akkor nem történik állapotátmenet.

Az AF temporális operátor elfogadó automatája

A második elfogadó automata az AF operátor automatája, amely egy a tesztelés során jól használható operátorok közül. Az operátor segítségével *élőségi feltételeket* vizsgálhatunk, azaz a meghatározott állapotkifejezésnek minden lefutás során legalább egyszer teljesülnie kell. Ezzel az operátorral a rendszer egy-egy feladatának teljesülését ellenőrizhetjük – például hogy egy vonat mindig eljut-e az állomásra.

Az elfogadó automata mindaddig, amíg az aktuális lefutásban nem teljesül a feltétel, nem meghatározott kimenettel rendelkezik. Ha egy lefutás során egyszer sem teljesül a feltétel, akkor a kifejezés sérül és ez hamis állapotot fog eredményezni. Az 5.3. ábrán látható az elfogadó automata, amely az 5.1(a). ábrán szemléltetett útvonal halmazt fogadja el.

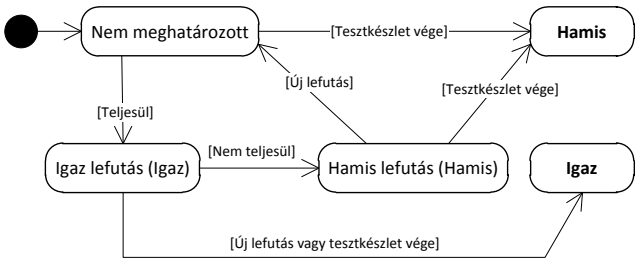


5.3. ábra. Az AF temporális TCTL operátorhoz tartozó elfogadó automata

Az EG temporális operátor elfogadó automatája

Az 5.4. ábrán látható automata az EG operátor elfogadását írja le. Ez a temporális operátor azt jelenti, hogy a lefutások során legalább egy lefutásnak léteznie kell, ahol mindig teljesül a meghatározott feltétel. Az elfogadott útvonal halmazt az 5.1(d). ábra szemlélteti.

Az elfogadó automata ebben az esetben biztos végeredményt csak az utolsó lefutás végén tud adni (a tesztkészlet végeztével), így ez az operátor is legfőképp a tesztelés során használható.

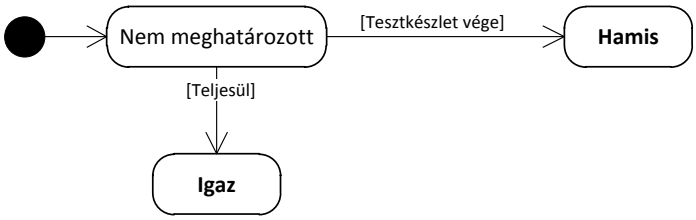


5.4. ábra. Az EG temporális TCTL operátorhoz tartozó elfogadó automata

Az EF temporális operátor elfogadó automatája

Az 5.5. ábrán az EF operátor elfogadó automatáját mutatom be. Az automata az AG operátor automatájához hasonlóan viszonylag egyszerű, amelynek a legfőbb oka, hogy a két operátor könnyen átírható egymásba: $\neg AGp = EF\neg p$. A temporális operátor segítségével azt vizsgálhatjuk, hogy a rendszer a működés/tesztelés során legalább egyszer eljutott-e egy megadott állapotba, azaz például ezzel vizsgálhatjuk a tesztkészletünk lefedettségét.

Az automata mindaddig nem meghatározott értéket ad eredményül, ameddig vagy teljesül a feltétel (innentől minden esetben igaz lesz az eredménye), vagy a konfiguráció lezárul, amikor is hamis eredményt ad. Az operátor által elfogadott útvonal halmazt az 5.1(c). ábra szemlélteti.



5.5. ábra. Az *EF* temporális TCTL operátorhoz tartozó elfogadó automata

A definiált automaták helyességéről a temporális operátorok szemantikájának megfelelő, illetve azt sértő lefutás sorozatok szisztematikus felvételével és ezek elfogadásának illetve hibajelzésének ellenőrzésével győződtem meg.

5.2.2. Holtpont detektálás

Az UPPAAL-ban használható TCTL formalizmus egy speciális kiegészítéssel rendelkezik, amely a *deadlock* állapotkifejezés. A kifejezés akkor válik igazzá, ha a rendszer holtpontba került, azaz az aktuális rendszerállapot nem fog többé megváltozni. Munkám során a kifejezés kiértékelésére egy részleges megoldást adtam. A probléma nehézségét azt jelenti, hogy egyes komponensek időlegesen *„lokális holtpontba”* kerülhetnek, azaz önmagukra hagyva még az idő múlásával sem képesek állapotváltozásra, ehhez valamilyen szinkronizációra lenne szükség. A szinkronizáció csak akkor valósulhat meg, ha a partner komponensek ennek megfelelő állapotba kerülnek, amit további komponensek befolyásolhatnak, és így tovább. A továbblépéshez tehát összetett szinkronizációs szekvenciára lehet szükség. Ennek detektálására a rendszer állapotterét – vagy legalábbis annak egy részét – kellene felderíteni, amely túlmutat munkám keretein. Az általam megvalósított megoldás azt garantálja, hogy amennyiben holtpontot jelez, akkor ténylegesen holtpont van a rendszerben – tehát nincs téves pozitív holtpont jelzés –, de előfordulhat olyan eset, hogy a rendszer nem érzékeli a holtpontok meglétét.

A monitor kiértékelő komponense akkor fogja a rendszert holtpontban levőnek tekinteni, amennyiben az összes komponense lokálisan holtpont állapotban van. Minden egyes komponens – a többitől függetlenül – akkor jelez a monitor felé lokális holtpontot, ha a várakozó függvénye az aktuális állapotnak azt érzékeli, hogy nincs engedélyezett kimenő él (a szinkronizációkat nem figyelembe véve), és olyan kimenő él sincs az állapotból, amely kizárólag egy időfeltétel miatt nem engedélyezhető, de ez az idő múlásával engedélyezetté válik majd. Amikor egy komponens kikerül a lokális holtpontjából, akkor ezt is jelzi a monitor felé, így a monitornak pontos képe van arról, hogy a rendszer melyik komponensei vannak aktuálisan helyi holtpontban.

5.3. A monitorok szintézise a TCTL követelmények alapján

A fent bemutatott elfogadó automatákat a monitor szintézis eszköznek elő kell állítania. Ehhez bemenetként a rendszer TCTL követelményeit használja fel. A követelményeket egy *parser* beolvassa és feldolgozza. Minden állapotkifejezésből egy *kifejezés-fa* épül, amelyet a TCTL monitor eltárol. A monitor feladata a kifejezés-fák logikai értékének ellenőrzése. Az

ellenőrzés során a monitor kiértékeli az állapotkifejezések logikai értékét minden esetben, amikor egy új üzenet érkezik a monitor számára. A monitor a kimenetét az elfogadó automaták által megadott lépéssorozat eredményeként határozza meg.

5.4. A program felműszerezése

A futásidőben történő információ küldéshez a vezérlő komponensek kódját fel kell műszerezni. A temporális logikai kifejezésekből szintetizált monitorok számára nincs szükség az összes állapotinformációra, kizárólag a korábban definiált megfigyelt állapotok és változókkal kapcsolatosan kell értesítéseket küldeni. A megfigyelt állapotokba történő be és kilépés esetén a komponens üzenetet fog küldeni a monitornak. Ehhez a felműszerezés során minden megfigyelt állapotba belépés, illetve állapotból kilépés kódjába be kell illeszteni a monitornak információt küldő platformszolgáltatások hívását. A megfigyelt változók értékét a kilépő függvény során kell megvizsgálni. Amennyiben valamilyen változófrissítő utasítás megfigyelt változó értékét érinti, akkor az új értéket egy üzenet formájában közli az alkalmazás a monitorral.

6. fejezet

Rendszer szintű monitorok szintézise LSC forgatókönyvek alapján

A második rendszer szintű monitorozó megoldás a forgatókönyv alapú monitorok automatikus szintézise. A szintézis eszköz bemenetként az UPPAAL-ban leírható LSC forgatókönyveket várja, amelyekből olyan monitor alkalmazásokat képez, amelyek a forgatókönyv által specifikált viselkedést fogadják el. A fejezetben az előzőekhez hasonlóan, elsőként bemutatom, hogy a monitorok pontosan milyen referencia információ alapján szintetizálhatóak. Ezután bemutatom a szintézis algoritmust, amely az elfogadó automaták konstrukcióját végzi. Ezt követően a monitorok futtatásához szükséges futtatókörnyezet bemutatása következik, majd a fejezetet a felműszerezés leírása zárja.

6.1. Referencia információ a monitor szintézishez: Az LSC forgatókönyvek

A temporális logikai kifejezéseken alapuló kifejezések sok esetben rendkívül hatékonyak, azonban vannak olyan követelmények, amelyek leírása TCTL kifejezésekkel vagy nehéz, vagy lehetetlen. A nehézségeknek két fő oka van [11]:

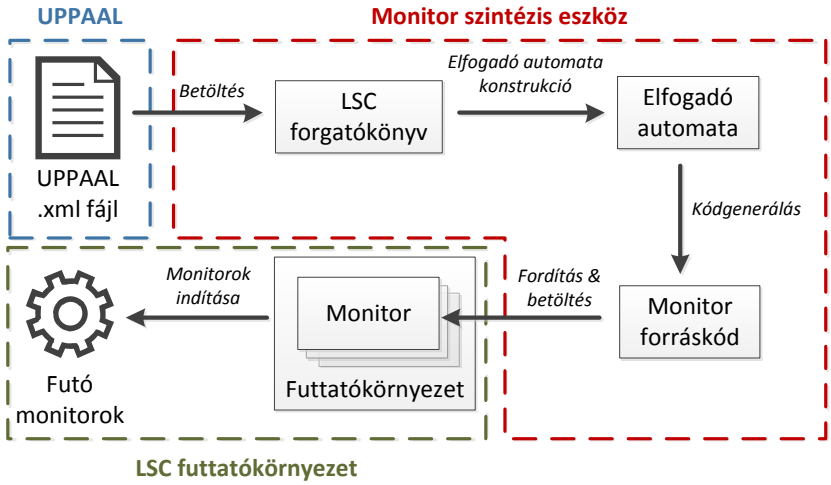
- A temporális kifejezés atomi kijelentései csak állapotkifejezések lehetnek. Események, üzenetküldések nem jelenhetnek meg.
- Nincs lehetőség bonyolult időkifejezéseket megadni (nincs időkorlátos temporális operátor).

Az UPPAAL fejlesztői a problémát szintén érzékelték, és megoldásként a legfrissebb, fejlesztői verzióban megjelent a Live Sequence Chart forgatókönyv alapú követelmény leíró formalizmus. A formalizmus pontos definícióját a 2.1.3. fejezetben adtam meg. Az UPPAAL az általános LSC-hez képest néhány megszorítást tartalmaz:

- Csak *„forró”* üzenetküldés lehet (minden elküldött üzenet meg is érkezik).
- Nem támogatott az LSC-k egymásba ágyazása.
- Nem lehet tiltó forgatókönyveket létrehozni.

A monitoroknak a következő információkra van szükségük futásidőben:

- A rendszer bármely két automatája között történő szinkronizációkra.



6.1. ábra. Az LSC alapú rendszerszintű monitorok előállításának lépései

- Azon változók és óraváltozók aktuális értékeire, amelyek megjelennek a forgatókönyv feltételeiben.

6.2. LSC alapú monitor szintézis lépései

A monitorok szintézisének alapvető lépéseit a 6.1. ábra mutatja. Az ábrán látható, hogy elsőként be kell olvasni az UPPAAL saját formátumából a forgatókönyveket. Ezután egy-egy olyan elfogadó automatát konstruálunk egy-egy forgatókönyvből, amely kizárólag a forgatókönyv által leírt viselkedést fogadja el. Ezt az automatát ugyanolyan időzített automataként konstruáljuk, mint az UPPAAL többi automatája. A megoldás előnye, hogy így a korábbi, időzített automata alapú forráskód generálási algoritmusunkat alkalmazhatjuk az elfogadó automata forráskódjának generálására. A konstrukció algoritmusát a 6.2.1. fejezet írja le.

Tehát az így konstruált automatából a korábban bemutatott kódgenerátor segítségével automatikusan forráskódot generálunk. A kódgenerátoron csupán minimális változtatásokat kellett eszközölni, ennek részleteit a 6.2.4. fejezetben mutatom be. A monitor szintetizáló rendszer ezután egy külső fordítóprogram segítségével előállítja a lefordított monitort.

Terveznem kellett egy olyan futtatókörnyezetet, amely képes a monitorok automatikus indítására, leállítására és hibakezelésére, tekintetbe véve az egyes LSC forgatókönyvekben megadott aktiválási módokat. A futtatókörnyezet felelőssége, hogy megteremtse a kapcsolatot a monitorok és az alkalmazás komponensei között. A futtatókörnyezet részletes bemutatása a 6.3. fejezetben olvasható.

6.2.1. Elfogadó automata előállítása

Az elfogadó automaták előállításához egy olyan algoritmusra van szükség, amely egy LSC forgatókönyvből képes olyan időzített automatát készíteni, amely a rendszernek csak azt a viselkedését fogadja el, amely az LSC forgatókönyvben specifikálva van. Ilyen algoritmust használ az UPPAAL modellellenőrző modulja is, amelynek algoritmusait publikálták [31]-ben, illetve [9]-ben. Munkám során ezt az algoritmust használtam, azonban ezt adaptálni kellett a futásidejű működéshez, hiszen az eredeti algoritmus tervezési idejű modellellenőrzéshez készült. Az én megoldásom azonban futásidejű ellenőrzést végez, amely például azt

eredményezi, hogy nem biztosítható az ellenőrzött komponens és az ellenőrző automata együttes atomi lépése. A továbbiakban elsőként bemutatom az eredeti algoritmust, majd rátérek az elvégzett módosítások ismertetésére.

6.2.2. Az eredeti elfogadó automata konstrukciós algoritmus

Az algoritmus alapgondolata nagyon egyszerű. A bemeneti LSC szimrégióit az elfogadó automatában állapotátmenetekre, míg a vágásokat állapotokra kell leképezni. Az algoritmus lépései a következők:

- Felderítjük a forgatókönyv szimrégióit. A szimrégiók a konstruált elfogadó automatában állapotátmenetekként jelennek meg. Ezek felderítéséhez meg kell találni a forgatókönyv azonos p pozícióihoz rendelt m üzenetküldéseket, c feltételeket illetve a akciókat, amelyek együttesen egy $s = \langle m, c, a \rangle$ szimrégiót adnak.

- Felvesszük a forgatókönyv minimális vágásának megfelelő állapotot az elfogadó automatában. A forgatókönyv c_{min} vágása egy olyan vágás, amely nem tartalmaz egyetlen szimrégiót sem. Ebből képződik az elfogadó automata kezdőállapota és ebből indul ki az elfogadó automata konstrukciós algoritmus további része.

- Ha a forgatókönyv rendelkezik *prechart*-tal, akkor először annak a szimrégióit dolgozzuk fel. A szimrégiók feldolgozását külön részletezem a konstrukciós algoritmus bemutatása után. A prechart feldolgozása után megtalált vágást nevezzük c_{start}-nak. Ez a vágás tartalmazza a prechart összes szimrégióját, azaz $\forall s \in c_{start}$, ha $s \in S \wedge s \in Pch$.

- Feldolgozzuk a *mainchart* szimrégióit. A prechart szimrégióinak feldolgozásához hasonlóan történik ez is. Az algoritmus részletezése a konstrukciós algoritmus ismertetése után olvasható. A main chart feldolgozása után a forgatókönyv maximális vágása is leképezhető egy állapotba az elfogadó automatában. A c_{max} maximális vágás olyan vágás, amely a forgatókönyv összes szimrégióját tartalmazza, azaz $\forall s \in S : s \in c_{max}$.

- Felveszünk egy állapotátmenetet az utolsó állapotból a kezdőállapotba. Ez a lépés azért szükséges, hogy miután a forgatókönyv teljesülését ellenőrző automata a végére ért, induljon el újra és a forgatókönyv ellenőrzése ne álljon meg.

- A forgatókönyvben nem megkötött akciók (implicit engedélyezett akciók) explicit engedélyezése az elfogadó automatában. Az LSC szemantikája nem köt meg olyan üzenetküldéseket, amelyek nem jelennek meg a forgatókönyvben. Tehát, ha $\mathcal{L}$ forgatókönyv csak az a és b üzeneteket jeleníti meg ($\Sigma = \{a, b\}$), akkor a c üzenetek küldésére semmilyen megkötést nem tartalmaz, tehát ilyen szinkronizációk szabadon történhetnek a rendszerben. A modellellenőrzés számára azonban ezt explicit le kell írni, azaz minden állapothoz hozzá kell adni egy önmagába mutató élt minden egyes a forgatókönyvben nem megjelenített üzenethez.

- Hideg feltételek megsértésének kezelése az elfogadó automatában. A hideg feltételek megsértése az LSC definíciója szerint nem okozza a forgatókönyv megsértését, csak meghiúsulását. Azaz a forgatókönyv ellenőrzését újra kell kezdeni, de nem kell a rendszernek hibát jeleznie. Az elfogadó automatában ez úgy jelenik meg, hogy hideg feltételekkel rendelkező állapothoz egy olyan élet kell rendelni, amely a kezdőállapotba mutat és őrfeltétele a hideg feltétel megsértése.

- Ha van *prechart*:

  - Hideg feltételek megsértésének kezelése a *prechart*-ban. Az előző lépéshez hasonlóan a hideg feltételek megsértése esetén a kezdőállapotba kell visszalépni.

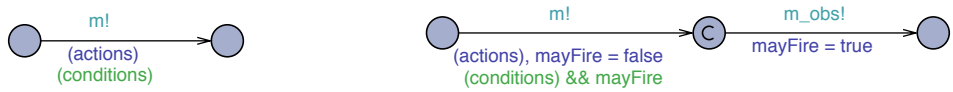
  - *Prechart*-ellenőrzés beillesztése az automatába. A *prechart* a forgatókönyv előfeltételeit írja le. Ha ezek a feltételek nem teljesülnek, akkor a forgatókönyv meghiúsul (nem megsérül). A *prechart* minden vágásából képzett állapothoz egy olyan állapotátmenetet kell hozzáadni, amely a kezdőállapotba mutat. Feltétele pedig minden esetben az állapotból eredetileg kilépő él feltételének tagadása. Ez azt eredményezi, hogy ha a *prechart*-ból nem tud tovább lépni az automata, akkor visszalép a kezdőállapotába.

- A forgatókönyv megsértésének kezelése az automatában. Ha a forgatókönyv nem meghiúsul, hanem megsérül, azaz a megfigyelt rendszerben olyan lépés történt, amit a forgatókönyv nem engedélyez, akkor hibát kell jelezni. Az elfogadó automatában ez egy *Error* állapot létrehozásával tehető meg. Ebbe az állapotba kell vinni a rendszert, ha egy forró feltétel nem teljesül vagy ha egy olyan szimrégió következne, ami nem engedélyezett (az *engedélyezettség* definícióját lásd a következő alfejezetben).

- A rendszerben résztvevő időzített automaták feldolgozása következik ezután. Az eredeti algoritmus ugyanis nem kizárólag egy új elfogadó automatát illeszt a rendszerhez, hanem az eredeti automatákat is módosítja. A módosítások a következőek:

  - Megfigyelő csatornák beillesztése az automatákba. Az elfogadó automatának értesülnie kell az eredeti rendszerben történő szinkronizációkról. Ennek megoldása a modellellenőrző környezetben úgy lehetséges, hogy minden csatornához egy párt kell létrehozni, és az eredeti automatákban történő szinkronizációkat fel kell bontani két szinkronizációra illetve köztük egy atomi állapotra. A probléma csak akkor van, ha miközben a rendszer a köztes atomi állapotban van, egy másik komponens szintén atomi állapotba kerül. Ilyenkor magasabb prioritással kell rendelkeznie a szinkronizáció köztes állapotában levő komponensnek. Ennek kezelése a *mayFire* globális változóval történik, amelyet hamisra állít ameddig a köztes állapotban van és minden szinkronizációval rendelkező állapotátmenet őrfeltételébe bekerül a változó ellenőrzése. A felbontás lépését a 6.2. ábra mutatja.

  - *As-Soon-As-Possible (ASAP)* szemantika kezelése. Az LSC definíciója szerint azok a szimrégiók, amelyek nem tartalmaznak üzenetet, *ASAP* szemantikával rendelkeznek. Ez azt jelenti, hogy ezek a szimrégiók atomi műveletként hajtódnak végre és a legmagasabb prioritással kell rendelkezniük a rendszerben. Ennek megvalósítása két részből áll. Az első, hogy azokat az állapotokat, amelyekből a kimenő él ilyen szimrégióból képződött, atomi (*committed*) állapottá alakítjuk. Ez már garantálja azt, hogy ezek a szimrégiók atomi műveletként fussanak le, de ha eközben az eredeti rendszerben is egy atomi állapot aktív, akkor előfordulhat, hogy az ott meghatározott átmenet történik meg előbb. Ennek elkerülésére az elfogadó automata olyan éleire, amely az elfogadó automata atomi állapotaiba mutat egy globális *nxtCmt* változó értékét igazra állító akciót illesztünk. Az állapotot elhagyó élekre olyan akciót illesztünk, amely ezt a változót hamis értékűre állítja. A rendszer automatáiban minden atomi állapotból kimutató élre egy olyan feltételt illesztünk, amelyik csak akkor értékelődik igazra, ha a *nxtCmt* változó értéke hamis.



(a) Az eredeti automata (b) A módosított automata

6.2. ábra. A megfigyelő csatornák beillesztése az automatákba

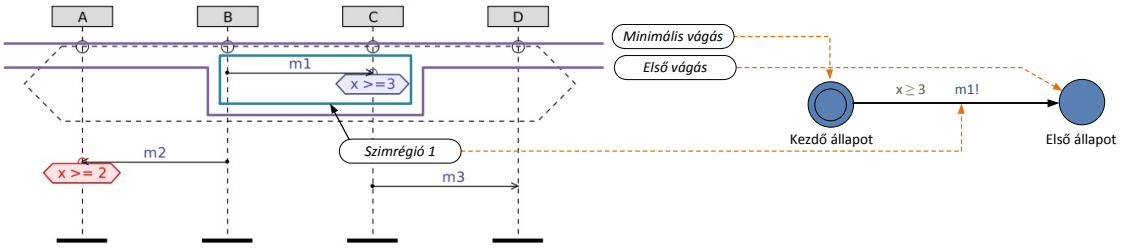
A szimrégiók feldolgozása

A szimrégiók feldolgozása a következő algoritmus alapján történik. Az összes megtalált szimrégió közül kiválasztjuk az *engedélyezett szimrégiókat*. Engedélyezett szimrégiónak tekintjük alapvetően az aktuális vágást követő szimrégiót vagy szimrégiókat. Ha a vágást az s_1 és s_2 szimrégiók követik, akkor az alábbi szabályok határozzák meg az engedélyezett szimrégiókat:

- Ha s_1 és s_2 is rendelkezik üzenetküldéssel, akkor mind a kettő engedélyezett.
- Ha s_1 rendelkezik üzenettel, de s_2 üzenet-nélküli szimrégió, akkor az $ASAP$ szemantika miatt csak s_2 lesz engedélyezett.
- Ha s_1 és s_2 is üzenet-mentes szimrégió, akkor ismét mindkettő engedélyezett lesz.

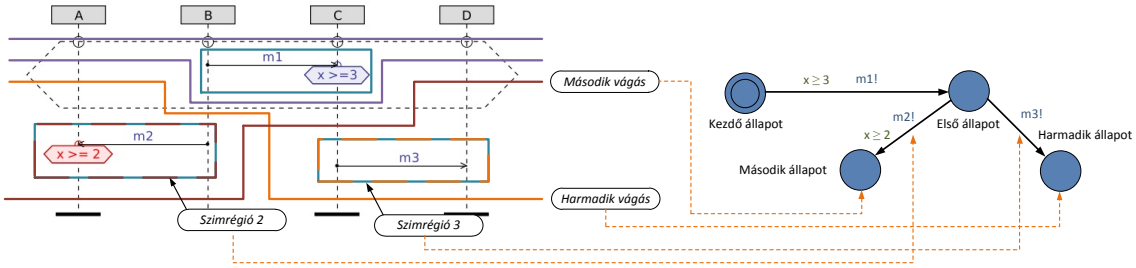
Miután az összes engedélyezett szimrégiót megtalálta az algoritmus, kiválaszt egyet közülük ($s = \langle m, g, s \rangle$), azt hozzáadja az aktuális c vágáshoz. Az így kapott új vágás $c' = c \bigcup \{s\}$, amelyből egy új állapot képezhető az elfogadó automatában. Ezután képzünk egy új állapotátmenetet a c vágásból képzett állapotból kiindulva és a c' vágásból képzett állapotba érkezve. Az új állapotátmenet őrfeltétele g, a hozzá rendelt akció a, illetve a szinkronizációs akciója m. Ezután meghívjuk rekurzív módon az algoritmust az új vágással, mint aktuális vágás. Ha egy ágon elértük a maximális vágást, akkor visszalépünk addig, ahol több engedélyezett szimrégió közül kellett választani, és választunk egy másikat. Ha az algoritmus során olyan vágást érünk el, amelyet korábban már feldolgoztunk, akkor a vágásnak nem veszünk fel egy második állapotot, hanem az eredeti állapotot használjuk fel. Az alábbi 2. példa demonstrálja az algoritmus működését egy egyszerű forgatókönyv segítségével.

2. példa. *A szimrégiók és vágások leképzése az elfogadó automatába.*



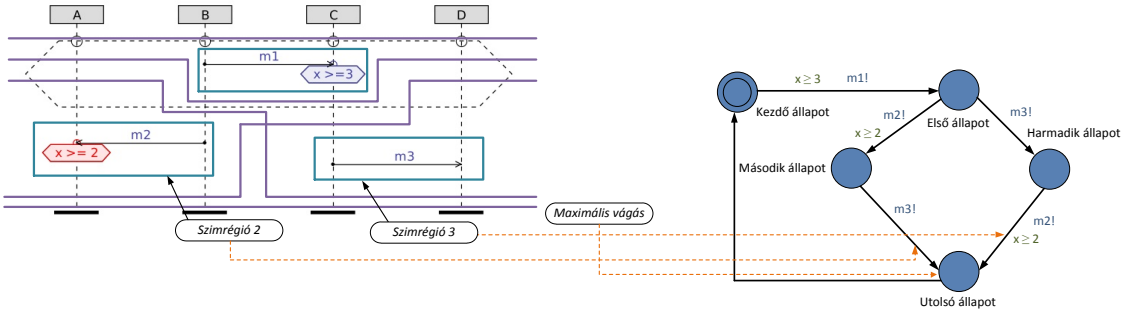
6.3. ábra. Prechart feldolgozása

A 6.3. ábrán egy forgatókönyv prechart-jának vágásai és szimrégiói képződnek le állapotokká és állapotátmenetekké. A minimális vágásból lesz az elfogadó automata kezdő állapota. A szimrégiók feltétele és üzenetküldése őrfeltételekké és szinkronizációs akciókká képződik.



6.4. ábra. Több engedélyezett szimrégió kezelése

A mainchart első lépése során két engedélyezett szimrégió lesz (hiszen a szimrégiók csak részlegesen vannak sorba rendezve). Az algoritmus ilyenkor az összes lehetőséget bejárja. Elsőként leképzi a kettes szimrégiót és második vágást a második állapotra és a hozzá tartozó állapotátmenetre. Ezután az algoritmus visszalép és megteszi a harmadikkal is ugyanezeket a lépéseket. A 6.4. ábra mutatja az elvégzett leképzéseket.



6.5. ábra. Maximális vágás feldolgozása

Miután a forgatókönyv feldolgozása kettévált az algoritmus rekurzív módon külön-külön lefut. A rekurzív hívások addig hívódnak meg, amíg meg nem találja az utolsó állapotot az elfogadó automatában, vagyis a maximális vágást a forgatókönyvben. Az algoritmus nem fogja kétszer hozzáadni az utolsó állapotot, hanem a két állapotátmenetet egy állapothoz köti. Végezetül az algoritmus hozzáad egy állapotátmenetet az elfogadó automatához, amely az utolsó állapotból indul és a kezdő állapotba érkezik. A rekurzív végrehajtás eredménye a 6.5. ábrán látható.

6.2.3. Az elfogadó automata konstrukciós algoritmus módosításai

Az eredeti algoritmus az UPPAAL modellellenőrzőjéhez készült. Az én megoldásom a tervezési idejű modellellenőrzés helyett futásidőben fog működni. Ez sok helyen módosításokat igényelt. Elsőként összefoglalom a módosításokat illetve azok okait, majd a módosított konstrukciós algoritmus pszeudokódját mutatom be.

A következő módosításokat végeztem el az algoritmuson:

- Az eredeti automatákhoz nem szükséges a megfigyelő csatornákat hozzáadni. A monitorok értesítését a kommunikációs platformszolgáltatás kibővítésével érem el. Sikeres kommunikációk végeztével a kommunikáció kezdeményezője értesítést küld a futtatókörnyezetnek a szinkronizációról.

- Az ASAP szemantikát megvalósító atomi állapotok létrehozása szükségtelen a futásidejű verifikáció szempontjából. Az atomi állapotok helyett a monitorok forráskódjának generálásánál a kódot úgy építem fel, hogy amennyiben egy állapotból van olyan

engedélyezett kimenő él, amelyen nincs szinkronizáció (tehát üzenet-mentes szimrégióból generálódott), akkor azon az élen még a szinkronizációs kísérletek előtt ki fog lépni a monitor. Az első két változtatás megtételét két dolog motiválta. Az első, hogy így nincs szükség az eredeti automaták módosítására, amely véleményem szerint egy elegánsabb megoldás, míg a másik, hogy az eredeti kódgenerátor alkalmazás nem támogatja az atomi állapotokat illetve a globális változókat is erősen korlátozottan (hiszen a célkitűzésünk alapvetően a lazán csatolt rendszerek támogatása volt).

- Az utolsó állapotból első állapotba léptető állapotátmenet hozzáadása a rendszerhez szintén elhagyható, hiszen a monitorok újraindítása a futtatókörnyezet felelőssége lesz. Erről részletesebben írok a 6.3. fejezetben.

- Az eredeti algoritmus szerint ha a forgatókönyv meghiúsul, akkor az elfogadó automata visszakerül a kezdőállapotban. Az én megoldásom e helyett jelez a futtatókörnyezetnek, amely le fogja állítani a monitort és ha szükséges, indít egy újat.

- Az implicit engedélyezett akciók, szinkronizációk kezelése az eredeti algoritmus szerint hurokélek hozzáadásával történik. Ehelyett a megoldásomban a futtatókörnyezet nem fogja továbbítani a felesleges szinkronizáció értesítéseket a komponensek felé (illetve egy részük nem is érkezik a monitorozott információkhoz illeszkedő felműszerezés miatt)..

A módosítások hatására az algoritmus némileg egyszerűbb lett, hiszen egyes részfeladatokat a futtatókörnyezetnek kell megoldania. A módosított algoritmus pszeudokódja a 2. algoritmus. Az algoritmus további algoritmusokra hivatkozik, melyek pszeudokódjai szintén megtalálhatóak ebben a fejezetben:

- *nextStep* algoritmus, amely a következő szimrégiókat dolgozza fel, rekurzív módon (3. algoritmus)

- *hidegFeltételek* algoritmus, amely a hideg feltételek megsértését kezeli (4. algoritmus)

- *prechartKezelo* algoritmus, amely a forgatókönyv előfeltételeinek ellenőrzését végzi (5. algoritmus)

- *hibaKezelo* algoritmus, amely a forgatókönyv megsértését kezeli (6. algoritmus)

Algoritmus 2: Módosított elfogadó automata konstrukció

bemenet: $\mathcal{L}$: LSC
kimenet : $obsTA$: IdőzítettAutomata

```
1  IdőzítettAutomata obsTA ← új IdőzítettAutomata
2  lista<Szimrégió > S ← L szimrégiói
3  Vágás c_min ← új (üres) Vágás
4  Állapot l_kezdo ← leképez(obsTA, c_min)
5  obsTA kezdőállapota ← l_kezdo
6  obsTA ← nextStep(L, c_min, obsTA)
7  hidegFeltételek(obsTA, L)
8  prechartKezelo(obsTA, L)
9  hibaKezelo(obsTA, L)
10 return obsTA
```

Algoritmus 3: nextStep

bemenet: $\mathcal{L}$: LSC, c : Vágás, $obsTA$: IdőzítettAutomata
kimenet : $obsTA$: IdőzítettAutomata

```
1  if c maximális vágás then
2  |  return obsTA
3  end
4  Állapot from ← c-ből képzett állapot
5  foreach Szimrégió s : engedélyezett szimrégiók do
6  |  hozzáad (c, s)
7  |  Állapot loc
8  |  if c vágás még nincs leképezve obsTA-ba then
9  |  |  loc = új Állapot
10 |  |  if engedélyezett szimrégiók között van üzenet-mentes then
11 |  |  |  loc legyen atomi
12 |  |  end
13 |  |  hozzáad (obsTA, loc)
14 |  |  obsTA ← nextStep(L, c, obsTA)
15 |  else
16 |  |  loc = c-ből képzett állapot
17 |  end
18 |  Állapotátmenet t ← új Állapotátmenet
19 |  t őrfeltétele ← s feltétele
20 |  t szinkronizációja ← s üzenetküldése
21 |  t akciója ← s akciója
22 |  t forrása ← from
23 |  t célja ← loc
24 |  hozzáad(obsTA, t)
25 end
26 return obsTA
```

Algoritmus 4: hidegFeltételek

bemenet: obsTA : IdőzítettAutomata, $\mathcal{L}$: LSC

```
1  foreach Feltétel c : L feltételei do
2  |  if c hideg feltétel then
3  |  |  t ← új állapotátmenet
4  |  |  t forrása ← c szimrégiójából képzett állapotátmenet forrása
5  |  |  t célja ← L kezdőállapota
6  |  |  t akciója ← futtatókörnyezet értesítése
7  |  |  t őrfeltétele ← ¬c
8  |  |  hozzáad (obsTA, t)
9  |  end
10 end
```

Algoritmus 5: prechartKezelo

bemenet: obsTA : IdőzítettAutomata, $\mathcal{L}$: LSC

```
1  foreach Szimrégió s : S do
2      if s rendelkezik üzenettel then
3          Állapotátmenet t ← új Állapotátmenet
4          t forrása ← l_kezdo
5          t célja ← l_kezdo
6          t szinkronizációja ← s üzenete
7          hozzáad (obsTA, t)
8      end
9  end
```

Algoritmus 6: hibaKezelo

bemenet: obsTA : IdőzítettAutomata, $\mathcal{L}$: LSC

```
1  Állapot Err ← új atomi Állapot
2  hozzáad (obsTA, Err)
3  foreach Állapot l : obsTA állapotai do
4      Vágás c ← l állapothoz tartozó vágás
5      if c ≠ maximális vágás és c ∈ Mch then
6          foreach Szimrégió s : S do
7              if s ∉ c vágás utáni szimrégiók then
8                  Állapotátmenet t ← új Állapotátmenet
9                  t forrása ← l_kezdo
10                 t célja ← l_kezdo
11                 t szinkronizációja ← s üzenete
12                 hozzáad (obsTA, t)
13             else if s rendelkezik egy g forró feltétellel then
14                 t forrása ← l_kezdo
15                 t célja ← Err
16                 t őrfeltétele ← ¬g
17                 t szinkronizációja ← s üzenete
18                 hozzáad (obsTA, t)
19             end
20         end
21     end
22 end
```

6.2.4. Kódgenerálás az elfogadó automatákhoz

Miután az $\mathcal{L}$ forgatókönyvből előállt az $obsTA$ időzített automata, amely az $\mathcal{L}$ által elfogadott viselkedés teljesülését ellenőrzi a rendszerben, forráskódot kell generálni belőle, hogy a rendszerhez illeszthető legyen. Mivel az $obsTA$ egy ugyanolyan időzített automata, mint a rendszer többi automatája, így a 2.3. fejezetben bemutatott kódgenerátor alkalmazás képes ebből is forráskódot készíteni, így ez külön problémát nem jelent.

6.3. Futtatókörnyezet az LSC monitorok számára

Az LSC forgatókönyvek ellenőrzése eltérő módon történik, mint a temporális logikai kifejezéseken alapuló ellenőrzések. A TCTL monitorokhoz hasonlóan ez is rendszer szintű monitor, azaz az összes komponenssel kapcsolatban áll a monitor. A nagy különbség az LSC formalizmus aktiválás módjaiból adódik, amelyek a következőek:

- *Invariáns aktiválási mód*: Az invariáns módú forgatókönyvek azt jelentik, hogy a rendszer teljes futása alatt, a forgatókönyvet bármikor is indítjuk el – azaz próbáljuk illeszteni az aktuális állapottól kezdődően a következő üzenetekre –, a forgatókönyvnek mindig teljesülnie kell. Az invariáns forgatókönyveket ellenőrző monitorok futását szemlélteti a 6.6(a). ábra.

- *Iteratív aktiválási mód*: Iteratív módban a forgatókönyv ellenőrzést a rendszer indulásakor el kell indítani, majd ha a forgatókönyv végére érünk, újra kell indítani. Az invariáns móddal ellentétben a forgatókönyvek „lefutásai" nem lapolódhatnak át. A monitorok futását szemlélteti a 6.6(b). ábra.

- *Kezdeti aktiválási mód*: A kezdeti mód hasonló a temporális logikai kifejezéseken alapuló monitorok ellenőrzésével. Ebben az aktiválási módban a forgatókönyvnek a futás elején kell teljesülnie, így ha egyszer teljesült a forgatókönyv, onnantól a teljes monitorozó rendszert le lehet állítani. A monitorok futását szemlélteti a 6.6(c). ábra.

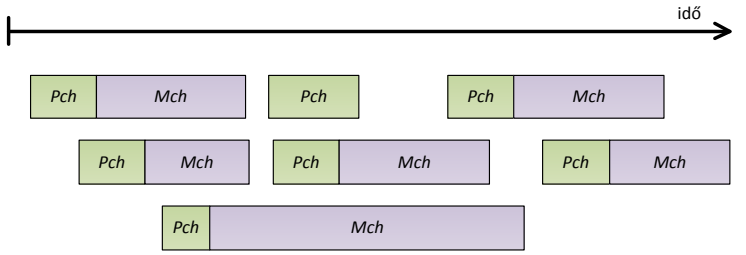
A különféle aktiválási módokkal különböző ellenőrzések végezhetőek. A legkomplexebb ellenőrzési mód az invariáns mód, hiszen itt a monitorok tetszőleges átlapolódását kell kezelni. Az általam bemutatott megoldás képes bármelyik mód használatára, köszönhetően a kifejlesztett *futtatókörnyezet* megoldásnak.

Miután a monitorok forráskódját a szintézis eszköz legenerálta, az lefordítható egy külső fordítóprogram segítségével, és ezt a futtatókörnyezet eltárolja és szükség esetén képes új példányokat indítani belőle, illetve a már futó példányokat képes leállítani. A futtatókörnyezet feladata továbbá, hogy tartsa a kapcsolatot mind a rendszer komponenseivel, mint a futó monitorokkal. Ennek az az oka, hogy a komponensek nem ismerhetik, hogy aktuálisan hány monitor példány fut, így nem is kapcsolódhatnak közvetlenül.

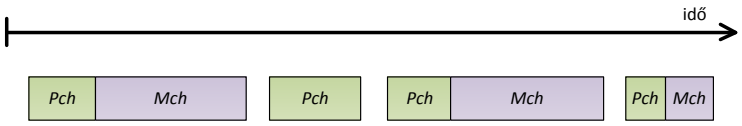
- A felműszerezett komponensek a megfigyelt változóik értékeinek megváltozásáról küldenek értesítést a futtatókörnyezet számára, amelyet továbbküld minden olyan monitor példánynak, amelyik számára van jelentősége a változónak – azaz van olyan feltétele, amelyik a változó értékét vizsgálja.

- A komponensek szintén értesítést küldenek a sikeres szinkronizációkról – ez a platformszolgáltatásba beépítve lett implementálva – a futtatókörnyezet számára, amely értesítést továbbküldi a megfelelő monitoroknak.

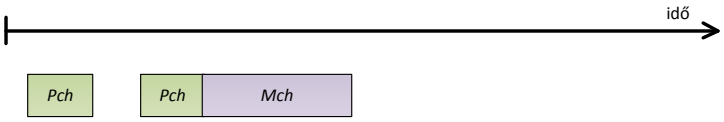
- Azoknak a monitor példányoknak kell kiküldeni az értesítést, amelyek figyelnek arra a csatornára. Erről a monitorok értesítést küldenek a futtatókörnyezet felé.



(a) Az invariáns aktiválási mód



(b) Az iteratív aktiválási mód



(c) Az kezdeti aktiválási mód

6.6. ábra. Az LSC formalizmus aktiválási módjait szemléltető ábrák

- A monitorok szintén értesítik a környezetet arról, ha valamilyen oknál fogva meghiúsulnának – hideg feltétel megsérül – illetve, ha megsérülnek. Meghiúsulás esetén a monitor példányt le kell állítani. Megsérülés esetén természetesen a rendszernek hibát kell jeleznie.

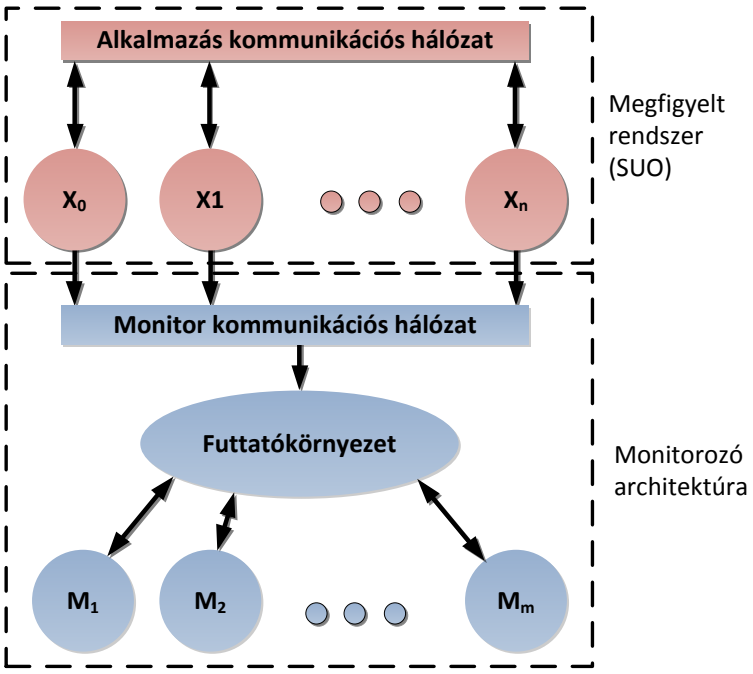
- A monitorok arról is értesítik a futtatókörnyezetet, ha elérik az utolsó állapotukat. A futtatókörnyezet reakciója erre is az, hogy a példányt le kell állítani.

A forgatókönyv alapú monitorozó rendszer architektúráját a 6.7. ábra mutatja be.

A kapcsolattartáson túl a monitorok indítása is komoly feladata a futtatókörnyezetnek. A futtatókörnyezet forgatókönyvenkénti beállítása, hogy a forgatókönyvet milyen aktiválási módban kezelje – mind a három korábban bemutatott mód támogatott. A monitorok indítása a következő módon történik:

- *Invariáns mód* esetén a forgatókönyvnek tetszőleges időpontban elindítva teljesülnie kell és ezt ellenőrizni is kell. Ez azt jelenti, hogy a futtatókörnyezet akkor indít új monitort, amikor csak lehet. A monitor indításának pontos feltétele, hogy ne legyen egyetlen egy forgatókönyvhöz tartozó monitor sem a kezdeti állapotában. Arra azonban a futtatókörnyezetnek figyelnie kell, hogy ha két monitor azonos állapotba került, akkor az egyiket abból le kell állítania, hiszen nincs rá tovább szükség.

- Az *iteratív mód* esetén az invariáns módhoz hasonlóan a monitorokat a rendszer teljes működése során indítani kell (azaz meg kell próbálni a prechartot illeszteni az aktuális működésre), de fontos megkötés, hogy egyszerre csak egy monitor lehet a *mainchart*-ban. Amikor egy monitor belép a *mainchart*-ba, akkor az összes többi monitort, illetve az új monitorok példányosítását le kell állítani.



6.7. ábra. Az LSC forgatókönyv alapú monitorozó architektúra és a futtatókörnyezet szerepe

- Az utolsó mód a *kezdeti mód*, amely a legegyszerűbb. A monitorokat addig kell indítani, amíg egy monitor be nem lép a *mainchart*-ba. Ekkor az összes többi monitort le kell állítani.

6.4. A program felműszerezése

Az LSC alapú rendszerszintű monitorozáshoz szükséges felműszerezések az alábbi néhány pontban foglalhatóak össze:

- A szinkronizációkról értesíteni kell a futtatókörnyezetet. Ennek implementálása a kommunikációs platformszolgáltatás kibővítésével történt meg. A szolgáltatás sikeres szinkronizációk esetén értesíti a futtatókörnyezetet.
- Az $\mathcal{L}$ forgatókönyv összes feltételében szereplő változók megfigyelt változók, így ezek értékeinek frissítése esetén is értesítést kell küldeni a futtatókörnyezetnek.

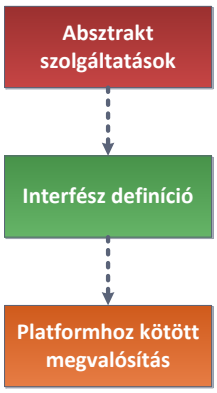
7. fejezet

Megvalósítás

Munkám során a hierarchikus monitorozás koncepcióján felül nagy hangsúlyt fektettem a gyakorlati megvalósításra is, így a különféle monitorozó megoldások tényleges implementációja is elkészült. Ebben a fejezetben a megoldások, felhasznált platformok részleteit szeretném bemutatni, valamint a mérési eredményeimet is közlöm az utolsó részben.

7.1. A kódgenerátor módosításai

A korábbi munkám során készített kódgenerátor eszközön módosításokat kellett végeznem, hogy a vezérlők felműszerezése a kódgenerálás során megtörténjen. A kódgenerátor alapvetően egy platformfüggetlen megoldás, így a platformfüggő részek, mint absztrakt szolgáltatások vannak definiálva (lásd 7.1. ábra).



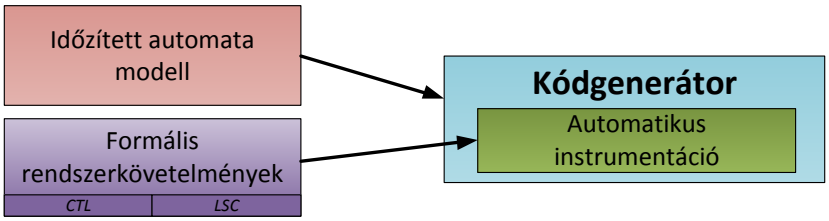
7.1. ábra. Az absztrakt platformszolgáltatások a kódgenerátorban

A monitorozáshoz szükséges információk elküldésének módja a platform által nyújtott lehetőségektől, a platform perifériáitól erősen függ, így ezeket újabb platformszolgáltatásokként kellett definiálni. Új platformszolgáltatás definíciókat hoztam létre az alábbiakhoz:

- Komponens szintű monitorozás
  - Állapotinformációk elküldése
  - Állapotváltozást jelző információ elküldése
  - Életjelinformációk küldése
  - Változóértékek elküldése

- Temporális logikán alapú rendszer szintű monitorozás
  - Állapotinformációk elküldése
  - Változóértékek elküldése
  - Lokális holtpont jelzése
- Forgatókönyv alapú rendszer szintű monitorozás
  - Változóértékek elküldése
  - Szinkronizáció értesítések

A szolgáltatások implementációja is elkészült. További fontos feladat volt, hogy a kódgenerátorba is be kellett illeszteni ezeket a szolgáltatásokat (7.2. ábra). Ennek implementálásához a kódgenerálási mintát a megfelelő pontokban módosítottam. A módosítások részletei korábban olvashatóak az egyes monitorozási technológiáknál.



7.2. ábra. A kódgenerátor automatikus instrumentációval

7.2. Monitor szintézis megvalósítása

A kódgenerátor eszköz módosításán kívül egy önálló eszközként implementálnom kellett a monitorok szintézisét végző alkalmazást. Az elkészült alkalmazás képes betölteni az UPPAAL által használt XML fájlt, illetve az UPPAAL követelményfájljait is képes feldolgozni. A beolvasott fájlokat a monitor szintézis folyamat bemeneteként használja fel. Az időzített automatából komponens szintű ellenőrző modulok generálhatóak (egy példa megtalálható a B függelékben). A TCTL követelményekből illetve a forgatókönyv alapú LSC specifikációkból pedig rendszer szintű monitorozó modulok szintetizálhatóak.

7.3. Felhasználás valós elosztott környezetben

Az eszközöket elosztott beágyazott rendszerekhez terveztem, így ilyen rendszerhez készült megvalósítás. A mintaalkalmazás megvalósításához az alábbi környezetet használtam fel:

- 4 darab *mbed* mikrovezérlővel rendelkező egyedileg tervezett vezérlőkártya
- 1 switch, amely az elosztott rendszer csomópontjainak összeköttetéséért felelős
- 1 monitorozó számítógép, amelyen a monitor alkalmazások futnak

A rendszer felépítése az A függelékben található. A mikrovezérlőket tartalmazó kártyákat a Méréstechnika és Információs Rendszerek Tanszéken tervezték kifejezetten a modellvasút rendszer számára. A monitorozó komponensek alkalmasak a közös, asztali számítógépen történő futtatásra, illetve beágyazott vezérlőkön történő futtatásra egyaránt. Ehhez a monitorozó platformszolgáltatások cseréje szükséges.

A rendszer kommunikációja (az időzített automaták szinkronizációja) az *mbed* platform Ethernet megoldásának UDP üzenetküldésével történik. A monitorok számára is ilyen üzenetek formájában jutnak el az üzenetek. Ez természetesen egy platformszolgáltatás implementációja, így tetszőleges megoldásra könnyedén cserélhető.

Az alkalmazást az UPPAAL eszközzel terveztem, illetve verifikáltam. A négy elosztott vezérlő egyikének modellje szintén az A függelékben található.

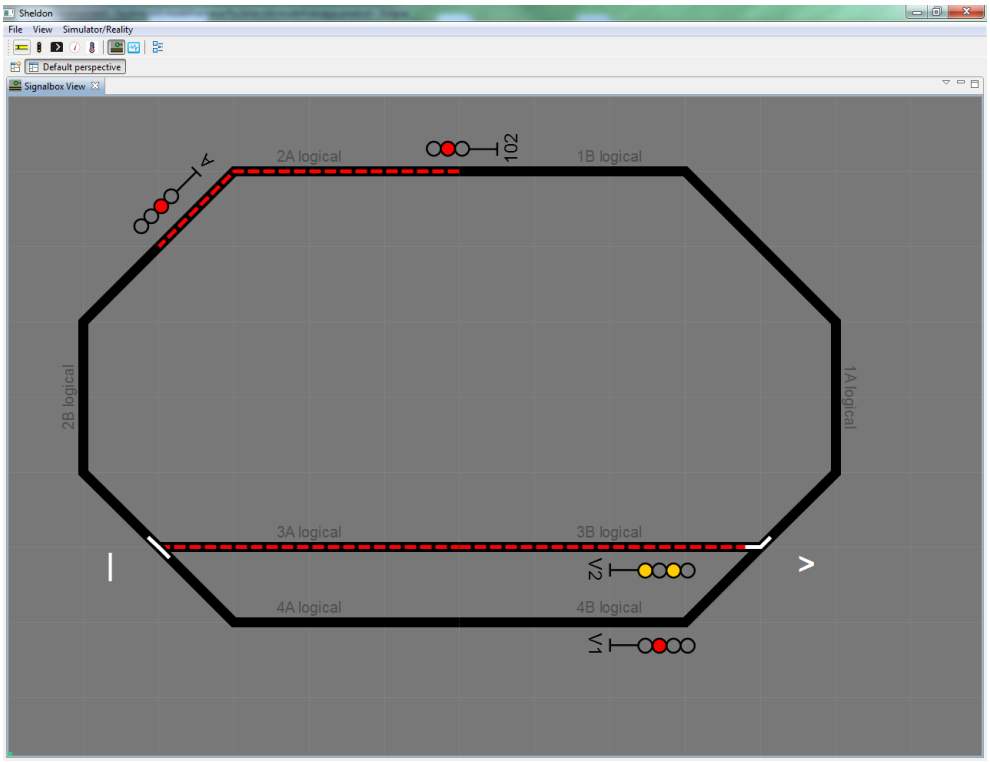
3. példa. *Ellenőrzött követelmények a modellvasút alkalmazáson:*

- *A vezérlés holtpontmentes legyen:* A[](!deadlock)
- *Egy szakaszdarabon csak egy vasút lehet (nem történhet ütközés):* AG(!((Vo1.Section1A and Vo2.Section1A) or (Vo1.Section1B and Vo2.Section1B) or (Vo1.Section2A and Vo2.Section2A) or (Vo1.Section2B and Vo2.Section2B) or (Vo1.Section3A and Vo2.Section3A) or (Vo1.Section3B and Vo2.Section3B) or (Vo1.Section4A and Vo2.Section4A) or (Vo1.Section4B and Vo2.Section4B)))
- *Egy sínszakaszon maximum 1 vasút tartózkodhat (ehhez segédváltozókat hoztam létre, amelyeket a vasút modell automatikusan frissít):* AG(section_count1 < 2 and section_count2 < 2 and section_count3 <2 and section_count4 < 2)
- *Egy jelző csak akkor lehet zöld, ha az utána levő szakaszon nem tartózkodik vasút:* AG((signal0 == GREEN imply (!Vo1.Section2A and !Vo1.Section2B and !Vo2.Section2A and !Vo2.Section2B)) and (signal2 == GREEN imply (valto2 == 0 and not Vo1.Section1A and not Vo1.Section1B and not Vo2.Section1A and not Vo2.Section1B)) and (signal3 == GREEN imply (valto2 == 1 and not Vo1.Section1A and not Vo1.Section1B and not Vo2.Section1A and not Vo2.Section1B)) and (signal1 == GREEN imply ((valto1 == 0 and not Vo1.Section3A and not Vo1.Section3B and not Vo2.Section3A and not Vo2.Section3B) or (valto1 == 1 and not Vo1.Section4A and not Vo1.Section4B and not Vo2.Section4A and not Vo2.Section4B))))
- *Ha egy jelző vörös, nem haladhat el mellette vasút:* AG(((signal0 == RED and (Vo1.Section1B or Vo2.Section1B)) imply section_speed1 == 0) and ((signal1 == RED and (Vo1.Section2B or Vo2.Section2B)) imply section_speed2 == 0) and ((signal2 == RED and (Vo1.Section3B or Vo2.Section3B)) imply section_speed3 == 0) and ((signal3 == RED and (Vo1.Section4B or Vo2.Section4B)) imply section_speed4 == 0))

7.4. Szimulációs környezet

A munkám során egy szimulációs környezet is elkészült. Ennek előnye, hogy különleges hardverek nélkül is könnyedén tesztelhetőek, demonstrálhatóak mind a kódgenerátor eszközök, mind a monitorozó alkalmazások. Az előzőekben ismertetett absztrakt platformszolgáltatásoknak köszönhetően lehetőség van arra is, hogy beágyazott rendszerek helyett asztali számítógépes környezetben is működhessenek az eszközök. A szolgáltatások implementációja elkészült az SDL C könyvtár segítségével Windows, Mac OS X illetve Linux környezetre is. Az 1. fejezetben bemutatott motivációs mintapélda alkalmazás szimulálásához egy külön megjelenítő is készült [1], amely a modellvasút rendszer összes perifériáját képes megjeleníteni. A szimulátor egy képernyőképe a 7.3. ábrán látható. Az alkalmazásba integráltam a kódgenerátor eszközt és a jövőben a monitorozó megoldások is integrálásra fognak kerülni.

[1] Ezt az alkalmazást Darvas Dániellel együtt készítettem



7.3. ábra. Modellvasút szimulációjához vizualizációs rendszer

7.5. Mérési eredmények

Dolgozatomban többször hivatkoztam arra, hogy a beágyazott rendszerekhez készített alkalmazások esetén kiemelt figyelmet kell fordítani a korlátozott erőforrásokra. Az automatikus vezérlő felműszerezés célja, hogy csak a követelményeknek ellenőrzéséhez ténylegesen szükséges instrumentáló kódokat illesszük be a vezérlő forráskódjába, ezáltal is minimalizálva a rendszeren okozott többletterhelést.

A monitorozás hatásait mértem mind alkalmazásméretben, mind futásidőben. Minden esetben a korábban bemutatott *mbed* platformot használtam a mérésekhez. Két *benchmark* modellt és egy valós modellt vizsgáltam mindkét esetben. Az első benchmark modell esetén csak gyors vezérlőfunkciókat használtam az állapotokban, illetve digitális kimenetek értékét állítottam be. Ebben az esetben az állapotátmenetek azonnal megtörténhetnek, semmilyen várakozásra nincs szükség az állapotokban. Ez nyilván nem egy valós helyzet, hiszen a rendszereknek mindig valamilyen számításokat kell végeznie, illetve külső jelekre várakoznia kell, de ez tekinthető a legrosszabb esetnek a felműszerezés szempontjából, hiszen egy ilyen rendszeren jelentős többletterhelést tud okozni. A második benchmark modell egy olyan modell, amely néhány állapotból és állapotátmenetből áll, illetve minden állapotban a soros porton küld egy rövid üzenetet, ezzel szimulálva azt, hogy az állapotokban valamilyen számítást illetve kommunikációt végez. A harmadik vizsgált modell a modellvasút rendszer modellje volt.

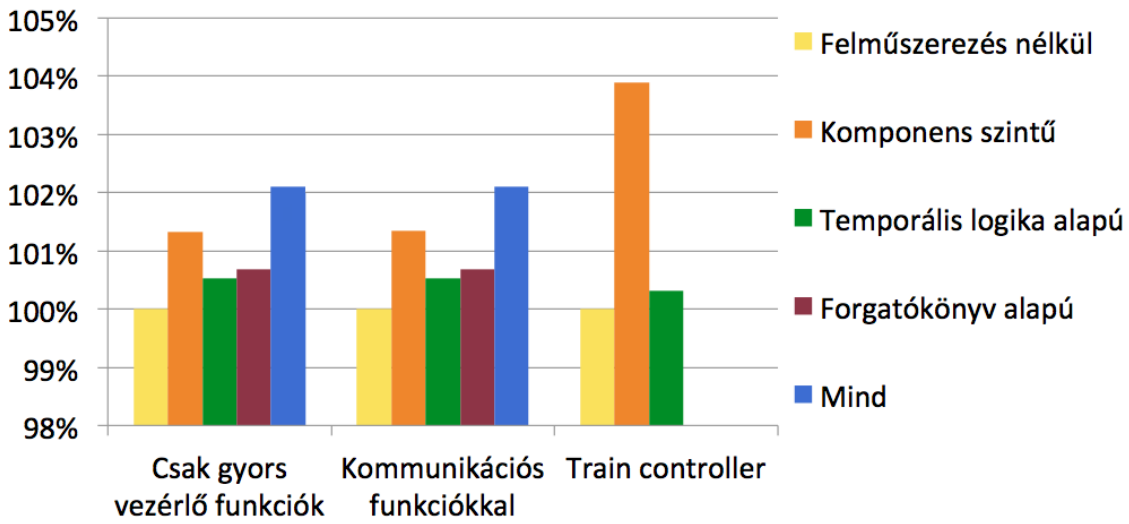
A méréseket elvégeztem felműszerezés nélkül, komponens szintű felműszerezéssel, temporális logikai felműszerezéssel illetve forgatókönyv alapú felműszerezéssel. Ez alól kivételt képez a modellvasút, ahol forgatókönyv alapú méréseket még nem végeztem.

7.5.1. Kódméret növekedések

A kódméretet minden esetben a lefordított alkalmazásnál mértem bájt mértékegységben. Az eredményeket a 7.1 táblázatban és a 7.4. ábrán mutatom be.

7.1. táblázat. Kódméret mérések *mbed* platformon

	Csak gyors vezérlő-funkciók	Kommunikáció és számítások is	Modellvasút
Felműszerezés nélkül	90832 bájt	90912 bájt	403724 bájt
Komponens szintű	92044 bájt	92132 bájt	419424 bájt
Temporális logika alapú	91308 bájt	91396 bájt	405012 bájt
Forgatókönyv alapú	91452 bájt	91534 bájt	n.a.



7.4. ábra. Kódméret mérések *mbed* platformon

A mérések alapján az tapasztalható, hogy számottevő kódméret növekedést nem okoz a felműszerezés – minden esetben 5% alatti növekedés tapasztalható.

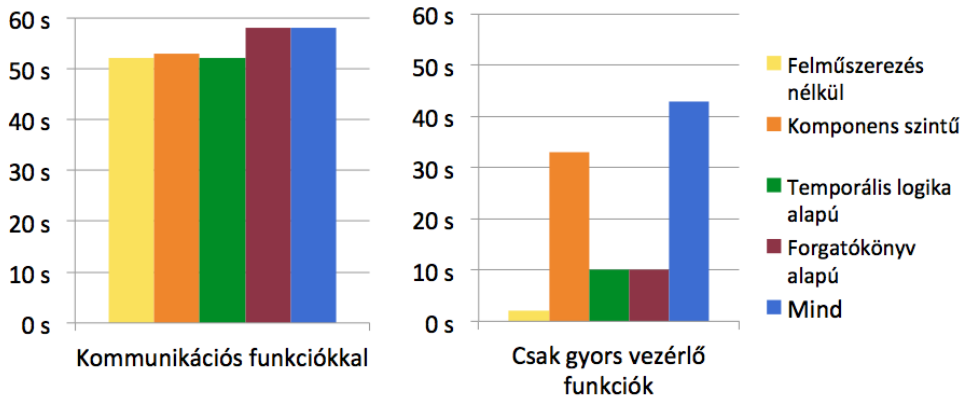
7.5.2. Futásidőbeli változások

A mérések során egy adott számú állapotváltást mértem meg és eredményeimet a 7.2. táblázat, illetve a 7.5. ábra mutatja. Mivel az első modell illetve a második modell között igen jelentős időkülönbségek tapasztalhatóak már felműszerezés nélkül, így az első esetben 500.000 állapotváltást mértem le, míg a második esetben csak 50.000 állapotváltás idejét mértem. A modellvasút mérése során azt tapasztaltam, hogy a rendszeren nem jelentkezik semmilyen többletterhelés. Ennek az oka az, hogy a rendszer az idő jelentős hányadában a fizikai rendszerre (a vonat mozgásából adódó jelre) várakozik, függetlenül attól, hogy

fel van-e műszerezve. Emellett a felműszerezés által generált többletműveletek elvégzésére fordított idő elhanyagolható. Az eredmények minden esetben másodpercben vannak feltűntetve.

7.2. táblázat. Futásidő mérése *mbed* platformon

	Csak gyors vezérlő-funkciók	Kommunikáció és számítások is
Felműszerezés nélkül	2 másodperc	52 másodperc
Komponens szintű	33 másodperc	53 másodperc
Temporális logika alapú	10 másodperc	52 másodperc
Forgatókönyv alapú	10 másodperc	52 másodperc



7.5. ábra. Futásidő mérése *mbed* platformon

Ebben a mérésben nagyon jelentős különbségek adódtak az első illetve második benchmark modell között. Ahogy említettem korábban, az első modell nem egy valós vezérlés megvalósítva, hanem a lehető legnagyobb futásidő növekedést mutatja. Ebben az esetben a komponens szintű monitorozás nagyon jelentős, 16,5-szeres növekedést okozott. A második modell, amely egy valós rendszert szimulál, már sokkal kisebb növekedést mutat, hiszen a komponens szintű monitorozás esetén is 2 % körüli többlet jelent meg.

8. fejezet

Összefoglalás és jövőbeli munkák

Dolgozatom végén szeretném munkámat összefoglalni és értékelni. Dolgozatomban egy hierarchikus monitor szintézis eszközt mutattam be, amely időzített automata modellek felhasználásával képes beágyazott rendszerek számára komponens, illetve rendszer szintű monitorokat szintetizálni. Írásomban bemutattam a megoldásom technológiai hátterét, pontosan definiáltam a felhasznált formalizmusokat. Ezután beszámoltam a területet érintő irodalomkutatásomról illetve a kidolgozott hierarchikus monitorozási koncepcióról. Az ezt követő fejezetekben részletesen bemutattam a különböző hierarchia szinteken végezhető monitor szintézis lépéseit.

Komponens szinten vezérlési folyam szintű monitorok generálhatóak a rendszer időzített automata modelljéből. Rendszer szinten monitorok szintetizálhatóak a rendszer TCTL temporális logikai követelményeiből, illetve a rendszer követelményeit leíró LSC forgatókönyvekből is. A TCTL monitor esetén pontosan meghatároztam a kiértékelő modul elfogadó automatáit, míg az LSC monitor szintézis esetén a szintézist végző algoritmusok pszeudokódjait is megadtam.

Munkám újdonságai az alábbi pontokban foglalom össze:

- A futásidőbeli verifikációhoz kidolgozott monitor megoldások már a tesztelési fázisban is felhasználhatóak, hiszen lefutások halmazát képesek kezelni. Ez annak köszönhető, hogy a klasszikus LTL formalizmus helyett a CTL formalizmust használtam a temporális logikán alapuló monitorok előállításához.

- Cél-orientált elfogadó automatákat definiáltam minden TCTL temporális operátorhoz, amelyek a temporális logikai kifejezéseken alapuló monitorok kiértékelő moduljának működését írják le formálisan.

- Definiáltam a monitor szintézis algoritmusát a forgatókönyv alapú Live Sequence Chart formalizmussal leírt követelményekből is, amellyel sok esetben jóval szemléletesebb és a mérnöki gondolkozáshoz közelebbi leírást eredményeznek, mint a temporális logikai nyelvek.

- A monitor szintézis illetve a monitorozott komponensek forráskódjának szintézise összehangoltan, egy modell alapú tervezői környezetben (az UPPAAL modellező környezet bővítéseként) történik. Ennek előnye, hogy a kódszintézis során az automatikus program felműszerezés olyan módon történhet, amely figyelembe veszi az ellenőrizendő követelményeket, így a felműszerezés optimalizálható. Ez csökkenti a kódméret növekedés illetve a futásidőbeli többletterhelést a rendszerben, amely beágyazott rendszerek esetén előnyös.

Munkám egyes részeit sikeresen publikáltam a nemzetközi SPLST'11 (*12th Symposium on Programming Languages and Software Tools, Észtország, Tallinn, 2011. október 5-7.*) tudományos konferencián [32], ahol a publikációt személyesen adtam elő.

A munkám a jövőben tovább folytatható. Az alábbi területeket emelném ki:

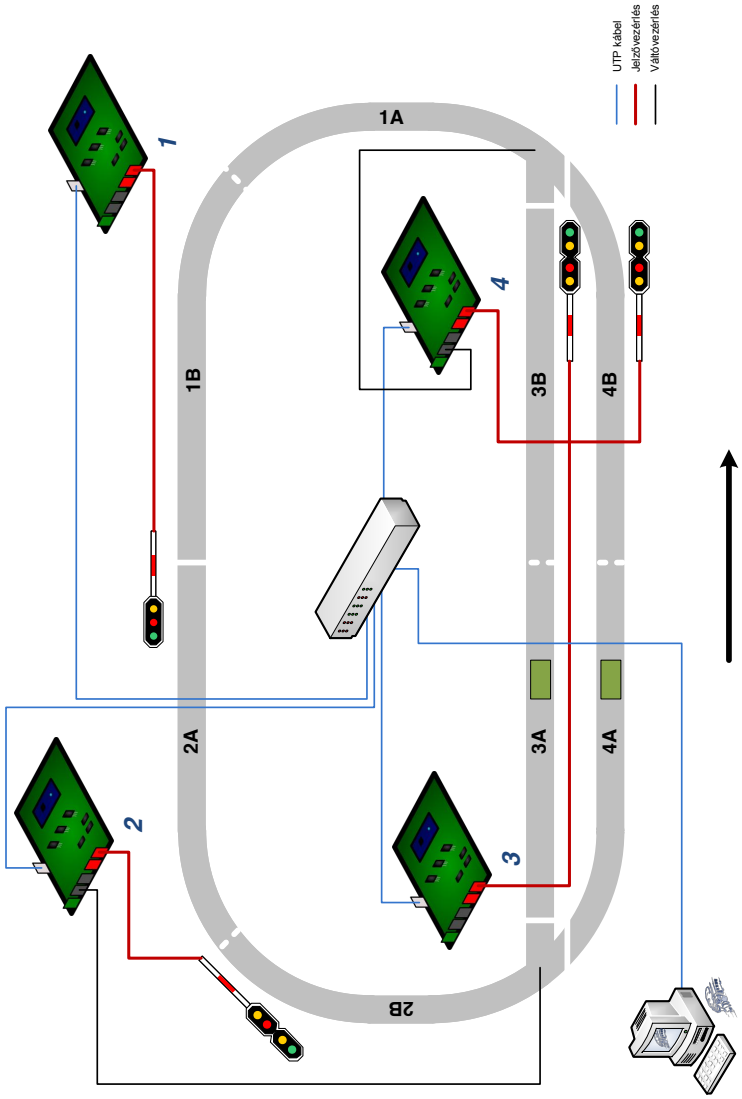
- További mérések végzése. A rendszer hibadetektáló képességét további mérésekkel kell alátámasztani, amelyhez valamilyen hibainjektáló technológiát kell alkalmazni. Eddig csak egyszerű tesztekkel (a modell illetve a forráskód kézi módosításával) történt a hibadetektálás ellenőrzése, ezt érdemes kiterjeszteni véletlen hibainjektálásra is.

- További követelményspecifikáló formalizmusokat is érdemes lenne megvizsgálni, illetve az LSC formalizmus kibővítése is hasznos lehetne, hiszen a formalizmus jelen formájában nem alkalmas például a kontextusfüggő alkalmazások követelményeinek leírására. Ez a célkitűzés szerepel a tanszék közreműködésével futó R3-COP (*Robust & Safe Mobile Co-operative Autonomous Systems*) projekt céljai között is.

A. függelék

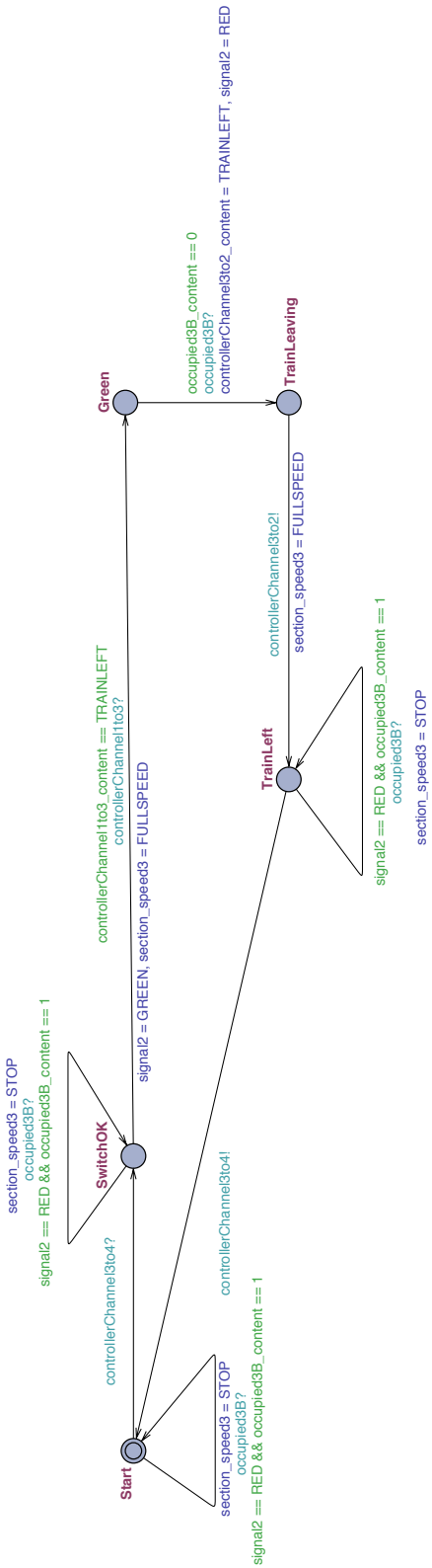
A modellvasút alkalmazás

A.1. Áttekintő alaprajz



Az ábrán látható a vasútmodell rendszer alaprajza. Az ábrán megjelenítem az elosztott vezérlőrendszer komponenseit illetve a vezérlőkhöz kötött perifériákat is.

A.2. Az egyik vezérlő időzített automata modellje



Az ábra a harmadik vezérlő időzített automata modelljét mutatja be UPPAAL-ban megtervezve.

B. függelék

Komponens szintű monitor forráskódja

A következőekben olvasható egy komponens szintű monitor forráskódja, amelyet a szintézis eszköz állított elő. A forráskód egyes részeit elhagytam.

```
1  #include "stdio.h"
2  #include "stdlib.h"
3  #include "clock_API.h"
4  #include <SDL_net/SDL_net.h>
5  #include <SDL/SDL_thread.h>
6  //Function declarations
7  void updateVariables(int ID, int data);
8    int checkInvariant0();
9    int checkInvariant1();
10 //...
11 void initialize();
12 int handleMessage (char* msg);
13 void updateVariables(int ID, int data);
14 void heartbeatChecker(void* unused);
15 int checkInvariantDispatcher(int id);
16 #define clock int
17 #define RESET 0
18 #define EDGE 1
19 #define LOCATION 2
20 #define HEARTBEAT 3
21 int monitoredDeviceID = 1;
22 //Defining data structure for storing reference information
23 typedef struct _Location {
24   int ID;
25   struct _Edge* edgeList;
26   int edgeListSize;
27 } Location;
28 typedef struct _Edge {
29   int ID;
30   Location* sourceLocation;
31   Location* targetLocation;
32 } Edge;
33 //Communication sockets
34 UDPsocket udpsocket;
35 UDPpacket *udppacket;
36 //Last message sequence number
37 int lastMessage=0;
38 //Last Heartbeat tick count
39 int heartbeatTicks=0;
40 //Location data
```

```
41 Location loc[5];
42 Location* currentLocation = NULL;
43 Edge* currentEdge = NULL;
44 //Device variables needed for timed invariant check
45 int  a = 0;
46 clock x;
47 //Initialize location data
48 void initialize() {
49   int i = 0;
50     loc[0].ID = 0;
51     loc[0].edgeListSize = 0;
52     i++;
53     loc[1].ID = 1;
54     loc[1].edgeListSize = 0;
55     i++;
56     //...
57     loc[0].edgeList = (Edge*)malloc(1 * sizeof(Edge));
58     loc[0].edgeListSize = 1;
59     loc[0].edgeList[0].ID = 0;
60     loc[0].edgeList[0].sourceLocation = &(loc[0]);
61     loc[0].edgeList[0].targetLocation = &(loc[3]);
62     loc[1].edgeList = (Edge*)malloc(2 * sizeof(Edge));
63     loc[1].edgeListSize = 2;
64     loc[1].edgeList[0].ID = 2;
65     loc[1].edgeList[0].sourceLocation = &(loc[1]);
66     loc[1].edgeList[0].targetLocation = &(loc[4]);
67     loc[1].edgeList[1].ID = 3;
68     loc[1].edgeList[1].sourceLocation = &(loc[1]);
69     loc[1].edgeList[1].targetLocation = &(loc[2]);
70     //...
71   SDL_CreateThread(heartbeatChecker, NULL);
72 }
73 extern int main (int argc, char * argv[]) {
74   initialize();
75   if (SDLNet_Init() < 0) {
76       INITIALIZATION_ERROR(SDLNet_GetError());
77       return -1;
78     }
79     //Communcation platform-level service initialization
80     //...
81     while (1) {
82       if (SDLNet_UDP_Recv(udpsocket, udppacket)) {
83         if (handleMessage(udppacket->data) == 0) {
84           VERIFICATIONERROR();
85           return 1;
86         }
87       }
88     }
89 }
90 //Timed invariant checker methods
91     int checkInvariant0() {
92       return (a < 5 && getClockValue(x) < 15);
93     }
94     int checkInvariant1() {
95       return 1;
96     }
97     //..
98 //Method for updating data of device variables
99 void updateVariables(int ID, int data) {
100   switch(ID) {
101       case 7:
102         a=data;
```

```
103         break;
104         case 6:
105           setClockValue(&x, (data));
106         break;
107         default:
108           UNKNOWNVARIABLE();
109         break;
110     }
111 }
112
113
114 //Function for handling message from client
115 int handleMessage (char* msg) {
116   int senderID = msg[0];
117   int msgSEQ = msg[4];
118   int posTYPE = msg[8];
119   int posID = msg[12];
120   if (senderID != monitoredDeviceID) {
121     //Wrong device id
122     WRONG_DEVICE_ID(senderID, monitoredDeviceID);
123     return 0;
124   }
125   if (posTYPE == RESET) {
126     RESET();
127     lastMessage = 1;
128     currentLocation = NULL;
129     currentEdge = NULL;
130     heartbeatTicks = SDL_GetTicks();
131     return 1;
132   }
133   if (msgSEQ != lastMessage + 1) {
134     //Wrong message sequence number.
135     SEQUENCE_NUMBER_ERROR(msgSEQ, lastMessage);
136     return 0;
137   }
138   lastMessage++;
139   if (lastMessage == 126) {
140     lastMessage = 0;
141   }
142    if (posTYPE == EDGE) {
143     if (currentLocation == NULL) {
144       NO_LOCATION_ERROR();
145       return 0;
146     }
147     int i;
148     for (i=0;i<currentLocation->edgeListSize;i++) {
149       if ((currentLocation->edgeList)[i].ID == posID) {
150         currentEdge = &((currentLocation->edgeList)[i]);
151         //Invariant assertion
152         int variableCount = msg[16];
153         int i;
154         for (i=0; i<variableCount;i++) {
155           int variableID = msg[20+(i*8)];
156           int variableContent = msg[24+(i*8)];
157           updateVariables(variableID, variableContent);
158         }
159         if (checkInvariantDispatcher(currentLocation->ID) == 0) {
160           INVARIANT_ERROR();
161           return 0;
162         }
163         return 1;
164       }
```

```
165       }
166       NO_EDGE_ERROR();
167       return 0;
168     } else if (posTYPE == LOCATION) {
169       if (currentLocation == NULL && currentEdge == NULL) { //First message
170         if (posID != 4) {
171           WRONG_INITIAL_LOCATION_ID_ERROR(posID);
172           return 0;
173         }
174         currentLocation = &(loc[4]);
175         return 1;
176       }
177       if (currentEdge == NULL) {
178         NO_EDGE_ERROR();
179         return 0;
180       }
181       if (currentEdge->targetLocation->ID != posID) {
182         WRONG_LOCATION_ID(posID);
183         return 0;
184       }
185       currentLocation = currentEdge->targetLocation;
186       currentEdge = NULL;
187       return 1;
188     } else if (posTYPE == HEARTBEAT) {
189       heartbeatTicks = SDL_GetTicks();
190       return 1;
191     }
192     UNKNOWN_MESSAGE_ERROR();
193     return 0;
194 
195 }
196 void heartbeatChecker(void* unused) {
197     while (1) {
198       SDL_Delay(4000);
199       if (SDL_GetTicks() - 4000 > heartbeatTicks) {
200         HEARTBEAT_ERROR();
201         exit(1);
202       }
203     }
204 }
205 int checkInvariantDispatcher(int id) {
206     switch (id) {
207         case 0:
208           return checkInvariant0();
209           break;
210         //...
211     }
212 }
```

Ábrák jegyzéke

Irodalomjegyzék

[1] nuSMV modellellenőrző eszköz. `http://nusmv.fbk.eu/` [Elérve: 2011.10.11.].

[2] PetriDotNet keretrendszer. `http://www.inf.mit.bme.hu/research/tools/petridotnet` [Elérve: 2011.10.11.].

[3] SPIN modellellenőrző eszköz. `http://spinroot.com/` [Elérve: 2011.10.11.].

[4] UPPAAL Documentation. `http://www.it.uu.se/research/group/darts/uppaal/documentation.shtml` [Elérve: 2011.10.11.].

[5] UPPAAL model checking tool. `http://www.uppaal.org` [Elérve: 2011.10.11.].

[6] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical computer science*, 1994.

[7] Rajeev Alur, Gerard Holzmann, and Doron Peled. An analyzer for message sequence charts. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 35–48. Springer Berlin / Heidelberg, 1996.

[8] Oliver Arafat, Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification revisited. Technical report, Technical University of Munich, 2005.

[9] Sandie Balaguer. Specification of properties using Live Sequence Charts. Technical report, Aalborg University, Denmark, 2009.

[10] A. Bauer, M. Leucker, and C. Schallhart. Model-based runtime analysis of distributed reactive systems. In *Software Engineering Conference, 2006. Australian*, page 10 pp., April 2006.

[11] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[12] Hanene Ben-Abdallah and Stefan Leue. Expressing and analyzing timing constraints in message sequence chart specifications. Technical report, Department of Electrical and Computer Engineering, University of Waterloo, 1997.

[13] Johan Bengtsson. Timed automata: Semantics, algorithms and tools. *Lectures on Concurrency and Petri Nets*, January 2004.

[14] K. Bhargavan, S. Chandra, P. McCann, and C.A. Gunter. What packets may come: Automata for network monitoring. *SIGPLAN Notices*, 35(3):209–219, 2001.

[15] UK Air Investigations Branch. Report on the incident to Airbus A340-642, registration G-VATL en-route from Hong Kong to London Heathrow on 8 february 2005. *Report 4*, 2007.

[16] Edmund Clarke. The Birth of Model Checking. *25 Years of Model Checking*, 2008.

[17] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln anf Narcisco Martí-Oliet, José Meseguer, and José Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[18] Pierre Combes, David Harel, and Hillel Kugler. Modeling and verification of a telecommunication application using Live Sequence Charts and the Play-Engine tool. *Software and Systems Modeling*, 7:157–175, 2008.

[19] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001.

[20] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *Software Engineering, IEEE Transactions on*, 30(12):859–872, December 2004.

[21] Paul S. Dodd and Cinya V. Ravishankar. Monitoring and debugging distributed real-time programs. *Software – Practice and Experience*, 22(10):863–877, October 1992.

[22] E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of computer and system sciences*, 30(1):1–24, February 1985.

[23] Alwyn Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, July 2010. Available at `http://ntrs.nasa.gov/search.jsp?R=278742&id=3&as=false&or=false&qs=Ns%3DArchiveName%257c0%26N%3D4294643047`.

[24] Weiming Gu, Greg Eisenhauer, and Karsten Schwan. Falcon : On-line Monitoring for Steering Parallel Programs. *Concurrency: Practice and Experienc*, 6(June):1–33, 1998.

[25] David Harel and Rami Marelly. *Come, Let's Play: Scenario-based Programming using LSCs and the Play-Engine*, volume 1. Springer–Verlag, 2003.

[26] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2), March 2004.

[27] International Telecommunication Union Telecommunication Standardization Sector (ITU-T). Z.120 Message Sequence Chart (Edition 5.0), 2011.

[28] P. Graubmann J. Grabowski and E. Rudolph. Towards a Petri net based semantics definition for message sequence charts. *SDL'93: Using Objects*, 1993.

[29] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-mac: a run-time assurance tool for Java program. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[30] Jochen Klose and Hartmut Wittke. An automata based interpretation of live sequence charts. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 512–527. Springer Berlin, 2001.

[31] Kim G. Larsen, Shuhao Li, Brian Nielsen, and Saulius Pusinskas. Verifying real-time systems against scenario-based requirements. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 676–691, Berlin, Heidelberg, 2009. Springer-Verlag.

[32] I. Majzik and G. Horányi. Automated code synthesis for run-time verification of distributed embedded systems. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools*, 2011.

[33] M. Mansouri-Samani and M. Sloman. Monitoring distributed systems (a survey). Technical report, Imperial College of Science and Technology and Medicine, 1992.

[34] András Pataricza, editor. *Formális módszerek az informatikában.* TypoTex, 2005.

[35] Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo, and Grigore Rosu. Hardware runtime monitoring for dependable COTS-based real-time embedded systems. *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 481–491, 2008.

[36] Gergely Pintér and István Majzik. Runtime Verification of Statechart Implementations. *Lecture Notes in Computer Science Architecting Dependable Systems III*, 3545:148–172, 2005.

[37] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[38] John Rushby. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA, 1995.

[39] John Rushby. Runtime certification. In Martin Leucker, editor, *Runtime Verification*, volume 5289 of *Lecture Notes in Computer Science*, pages 21–35. Springer Berlin / Heidelberg, 2008.

[40] Horányi Gergő és Jeszenszky Balázs. Elosztott beágyazott rendszerek formális modellek alapján történő fejlesztése paraméterezhető kódgenerálás segítségével. *BME TDK konferencia, Szoftver szekció*, 2010. `http://tdk2010.aut.bme.hu/Files/TDK2010/Elosztott-beagyazott-rendszerek.pdf`.

[41] Sriram Sankar and Manas Mandal. Concurrent runtime monitoring of formally specified programs. *Computer*, 26:32–41, March 1993.

[42] J.J.P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *Software Engineering, IEEE Transactions on*, 16(8):897 –916, aug 1990.

[43] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41:19:1–19:36, October 2009.