Írta:

ÉSIK ZOLTÁN
GOMBÁS ÉVA
NÉMETH L. ZOLTÁN

HARDVER- ÉS SZOFTVERRENDSZEREK VERIFIKÁCIÓJA

Egyetemi tananyag

2011

TÁMOGATÁS:
Készült a TÁMOP-4.1.2-08/1/A-2009-0008 számú, „Tananyagfejlesztés mérnök informatikus, programtervező informatikus és gazdaságinformatikus képzésekhez” című projekt keretében.

ISBN 978-963-279-497-6

KÉSZÜLT: a Typotex Kiadó gondozásában
FELELŐS VEZETŐ: Votisky Zsuzsa
AZ ELEKTRONIKUS KIADÁST ELŐKÉSZÍTETTE: Juhász Lehel

KULCSSZAVAK:
verifikáció, modell-ellenőrzés, átmeneti rendszerek, időzített automaták, temporális logikák, bináris döntési diagramok, modell-ellenőrző szoftverek, SPIN, UPPAAL.

ÖSSZEFOGLALÁS:
Napjaink informatikai rendszerei egyre komplexebbek, és rohamosan bővül azoknak az alkalmazásoknak sora, melyek formális verifikációt igényelnek a hagyományos szimuláció és tesztelés mellett. A jegyzet az egyik legfontosabb verifikációs módszer, a modell-ellenőrzés elméletébe és gyakorlatába kíván bevezetést nyújtani. Először áttekintjük a modell-ellenőrzés feladatát, megvalósítását, a módszer előnyeit és korlátait. Ezután konkurens és időzített rendszerek átmeneti rendszerekkel történő modellezését mutatjuk be, majd a rendszerek dinamikus viselkedésének leírása kerül tárgyalásra temporális logikák segítségével. Ezt követően az explicit és szimbolikus modell-ellenőrzés legfontosabb algoritmusait mutatjuk be. Végül a SPIN és az UPPAAL modell-ellenőrző szoftver rendszerrel ismertetjük meg az olvasót.

Tartalomjegyzék

1. fejezet

Verifikáció

1.1. A verifikáció szükségessége

A hardver és szoftver rendszerek ellenőrzésének, szakszóval verifikációjának szükségességét leginkább két közvetlenül tapasztalható, eléggé nyilvánvaló és egyre inkább érvényben levő folyamat megfigyelésével támaszthatjuk alá:

1. Egyre növekvő mértékben függünk napjaink számítógépes rendszereitől.

2. Ezen rendszerek komplexitása folyamatosan és drasztikusan nő.

Valóban, mindannyiunk életére direkt vagy indirekt módon hatással vannak az információs és kommunikációs rendszerek. Napjainkban számos *beágyazott rendszerrel* kerülünk kapcsolatba napi szinten. Ezek olyan speciális célú számítógépek, melyeket valamilyen konkrét feladat ellátására terveztek. Ezek vesznek minket körül az autókban, telefonokban, audiovizuális eszközökben, orvosi eszközökben, és még sorolhatnánk.

Az informatika nélkülözhetetlen a közlekedésben, iparban, energiatermelésben, vállalatirányításban, de egyre fontosabbak az elektronikus szolgáltatások az üzleti és banki világban, kereskedelemben, munkavállalásban, oktatásban is. Az első pont alátámasztása szempontjából különös figyelmet érdemelnek az úgynevezett *biztonságkritikus rendszerek*. Ezek olyan rendszerek, melyek meghibásodása emberek súlyos sérülését, sőt halálát okozhatja, vagy környezeti katasztrófát idézhet elő. Ilyen rendszereket találunk az atomerőművekben, vegyi üzemekben, létfenntartó orvosi berendezésekben, a közlekedésirányítási vagy riasztási rendszerekben.

Arról sem nehéz meggyőződni, hogy az informatikai rendszerek komplexitása eddig soha nem látott mértékben nő. Míg jó harminc évvel ezelőtt elsőként a Volkswagen 1600-as típusú autókban egyetlen mikroprocesszor volt, mely az üzemanyag-befecskendezést vezérelte, addig ma egy modern kocsiban több mint száz mikroszámítógépet találunk. Ezek együttes gyártási költsége meghaladja a járműhöz felhasznált acél árát.

Ma már korántsem képes egyetlen mérnök átlátni egy-egy nagyobb rendszer működésének részleteit. Ezt a komplexitást csak tetézi az egyre általánosabbá váló vezetékes és vezeték nélküli hálózati megoldások alkalmazása. Ezekben a rendszerekben az időben és térben egymástól eltérő tevékenységek komplex és sokszor előre nehezen kiszámítható módon hatnak

egymásra. A komplexitás a hibalehetőségek számának növekedését is okozza, így növekszik a hibafelfedéshez szükséges verifikáció komplexitása is.

A felsorolt két tényből közvetlenül adódik, hogy *a megbízhatóság kulcsfontosságú kérdés az információs és kommunikációs rendszerek tervezésében.*

Sajnos, nem nehéz olyan eseteket találni, amikor informatikai rendszerek meghibásodása katasztrofális következményekhez vezetett.

Az egyik klasszikus példa az Ariane-5 rakéta esete, melynek felrobbanását az első tesztrepüléskor, 1996 június 4-én, 37 másodperccel a start után, szoftverhiba okozta. Egy másik, sokkal szomorúbb baleset a Therac-25 sugárterápiás készülék hibája. 1985 és 87 között hat ember halt meg, mert a készüléket vezérlő szoftver hibája miatt az előírtnál körülbelül százszor nagyobb röntgensugárzást kaptak.

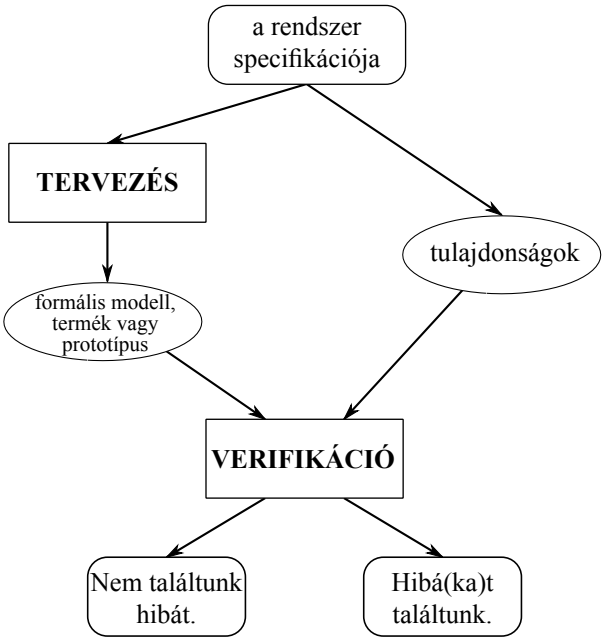
De nem feltétlenül csak az emberéletek veszélyeztetése győzhet meg bennünket a verifikáció szükségességéről. Szintén sokak által ismert az Intel egyik Pentium processzorának lebegőpontos osztási (FDIV) hibája. A hibát, csak több helytelenül működő processzorszéria piacra dobása után, 1994 májusában fedezték fel, először a cégen belül. Kezdetben a cég megpróbálta titokban tartani a hibát, majd miután egy matematikus professzor is észrevette, és 1994 októberében nyilvánosságra hozta, azzal mentegetőztek, hogy a hiba nem súlyos, és a legtöbb felhasználót nem érinti. Mégis, elsősorban a média és az egyre növekvő tiltakozás hatására, 1994 decemberében az Intel kénytelen volt bejelenteni a hibás processzorok ingyenes cseréjét. Ez a cégnek körülbelül 475 millió dollárjába került.

Ezek a példák és a [37] weboldalon felsorolt több mint 100 hasonló „szoftver horror történet” arra mutatnak rá, hogy a verifikáció nélkülözhetetlen biztonságkritikus és költségkritikus alkalmazásoknál.

1.2. Verifikációs módszerek

A hardver és szoftver rendszerek ellenőrzésében két különböző megközelítést létezik: a *verifikáció* és a *validáció*. Ezek segítenek annak elérésében, hogy helyes terméket helyes módon fejleszthessünk. A rendszer validációja a külső körülményeknek és a felhasználói elvárásoknak való megfelelést ellenőrzi. A verifikáció pedig azt vizsgálja, hogy az adott fejlesztési fázis eredménye teljesíti-e azokat a feltételeket, melyeket vele szemben a fejlesztési fázis kezdetén megfogalmaztunk. Másként fogalmazva a verifikáció azt kérdezi, hogy a terveknek megfelelően, *„jól építjük-e a rendszert?”* Ezzel szemben a validáció azt vizsgálja, hogy a fejlesztés összhangban van-e a rendszerrel szemben támasztott nem formális tervezési elgondolásokkal, azaz azt kérdezi, egyáltalán *„jó rendszert építünk-e?”* [8, 25].

Egyszerűen fogalmazva, a verifikáció során arról akarunk meggyőződni, hogy a tervezett vagy megvalósított rendszerünk bizonyos elvárt tulajdonságokkal valóban rendelkezik. A tulajdonságok, melyeket verifikálunk, lehetnek egészen egyszerűek, mint például a *holtpontmentesség*, mely azt fejezi ki, hogy a rendszer nem juthat olyan állapotba, melyből nincs továbblépés. Más tulajdonságok a rendszer biztonságos működésének feltételeit írják elő, például egy lift esetében kiköthetjük, hogy sohasem mozoghat nyitott ajtóval. Ezeket a követelményeket a rendszer *specifikációja* tartalmazza. A specifikáció pontos és szabatos leírása annak, hogyan kell és hogyan nem szabad a rendszernek viselkednie. Ez minden verifikációs



1.1. ábra. A verifikáció általános menete

tevékenység alapja. Ha a rendszer viselkedése csak egyetlen ponton is eltér a specifikációtól, azt hibának tekintjük, míg a verifikáció szempontjából a rendszer helyes, amennyiben viselkedése a specifikáció minden követelményét teljesíti. Ebből adódik, hogy a verifikáció sohasem a rendszer abszolút helyességét igazolja, hanem mindig relatív, csak a specifikációnak való megfelelést képes biztosítani.

Az 1.1 ábrán a verifikáció menetének vázlatát láthatjuk. A rendszer specifikációjából egyrészt a tervezés során egy formális modell, termék vagy prototípus, másrészt a megkövetelt tulajdonságok leírása származik. Ezek alkotják a verifikációs eljárás bemenetét, mely vagy sikerrel jár, vagy hibát, azaz a specifikációtól való eltérést mutat ki.

A verifikáció lehet elsődleges vagy utólagos. *Elsődleges verifikáció* esetén a rendszer formális modelljét ellenőrizzük, melyet a későbbi tervezési fázisokhoz vagy a gyártáshoz használunk fel. *Utólagos verifikáció* esetén ezzel szemben a prototípust vagy a kész terméket vizsgáljuk.

Most röviden tekintsük át a hardver és szoftver rendszerekben jelenleg alkalmazott verifikációs technikákat. Ehhez forrásul az [5] könyvet és a [8] tanulmányt használtuk.

1.3. Szoftver verifikáció

Könnyen látható általános szabály, hogy a szoftvertervezés során minél korábban sikerül egy hibát felfedezni, annál jobb. Valóban, egy olyan hiba javítási költsége, melyet csak az üzemeltetés szakaszában fedeznek fel, körülbelül 500-szor nagyobb, mintha a hiba felismerése még a tervezés fázisában történt volna. Ezért a verifikációnak el kell kezdődnie már a tervezéssel együtt, és végig kell kísérnie a szoftverfejlesztés egész menetét.

Szoftver rendszerek esetén a leggyakrabban alkalmazott hagyományos verifikációs technikák a *szakértői felülvizsgálat* (peer reviewing) és a *tesztelés*.

A szakértői felülvizsgálat azt jelenti, hogy a szoftvert alapos elemzésnek vetik alá, melyet lehetőleg olyan kompetens szoftverfejlesztők végeznek, akik az adott termék tervezésében és a fejlesztésében nem vettek részt. A vizsgálók a kapott kódot futtatás nélkül, teljesen statikusan értelmezik, elemzik és bírálják. Annak ellenére, hogy manuális módszerről van szó, a szakértői felülvizsgálat a gyakorlatban meglehetősen hatékony. Tapasztalati felmérések azt mutatják, hogy így a hibák 31-93%-át sikerül felfedezni, 60% körüli átlaggal. Így nem meglepő, hogy ennek a vizsgálatnak valamilyen formáját a legtöbb szoftverfejlesztési projektben alkalmazzák. Ugyanakkor a gyakorlat azt is mutatja, hogy a nehezen megfogható hibákon, például az algoritmusok hiányosságain vagy a konkurens végrehajtásból adódó speciális problémákon, a hasonló emberi megközelítés miatt a vizsgálók könnyen átsiklanak.

A *szoftvertesztelés* minden szoftvertervezési projekt igen jelentős része. A fejlesztés teljes költségének tipikusan 30-50%-át fordítják erre. A szakértői felülvizsgálattal szemben, amely statikus technika, a tesztelés dinamikus módszer, mely a kód minél többféleképpeni futtatásán alapszik. Tehát az elkészült (általában lefordított) szoftvert veszik alapul és a tesztelés céljára gondosan kiválasztott bemenetekkel, ún. tesztesetekkel látják el. Ezeket igyekeznek úgy megválasztani, hogy velük kikényszerítsék, hogy a szoftver futása során bizonyos utasítássorozatokon menjen át. Természetesen a futás során kapott eredményt a rendszer dokumentációjában rögzített specifikációval vetik össze. Míg a tesztesetek generálása és végrehajtása, ha nem is mindig és teljesen, de nagyrészt automatizálható, addig az eredményeknek a specifikációval való összevetését többnyire emberek végzik. A tesztelés legnagyobb előnye, hogy mindenféle szoftverre alkalmazható: klasszikus üzleti alkalmazásokra, fordítóprogramokra, operációs rendszerekre egyaránt. Ugyanakkor az összes lehetséges futás tesztelése lehetetlen, gyakorlatban a végrehajtási utaknak csak kis hányadát tudjuk ellenőrizni. Ezért a tesztelés sohasem teljes. Ebből adódik Edsger W. Dijkstra klasszikus mondása, mely szerint *a programtesztelés mindig csak a programhibák jelenlétét tudja kimutatni, sohasem a hiányukat* [17]. Egy másik probléma a teszteléssel annak meghatározása, hogy mikor hagyhatjuk abba. A gyakorlatban igen nehéz, ha nem lehetetlen, megállapítani, hogy a tesztelésben mekkora ráfordítással érhetjük el a *hibasűrűség* (a hibák száma osztva a programsorok számával) adott szint alá szorítását.

A tanulmányok szerint a szakértői felülvizsgálat és a tesztelés különböző típusú hibákat fedez fel a fejlesztés különböző szakaszaiban. Ezért gyakran együtt használják őket. Természetesen a szoftverminőség növelése érdekében ezeket a verifikációs technikákat kiegészítik a fejlesztés eredményességét segítő eszközök. Ilyenek a rendszerfejlesztési és specifikációs módszertanok, mint például az UML modellező nyelv és a hozzá kapcsolódó eszközök, valamint a verziókezelő és konfiguráció felügyelő rendszerek.

Ezen kívül a szoftverfejlesztési projektek 5-10 százalékában alkalmaznak *formális módszereket*. A formális módszerek közül a statikus és automatikus szoftver verifikációs módszereket mutatja be a [29] tanulmány. A cikkben ismertetett három technika: az absztrakt statikus analízis, a modell-ellenőrzés és a korlátos modell-ellenőrzés. Mivel a model ellenőrzés jegyzetünk központi témája, ezért a fentiek közül itt most csak az absztrakt statikus analízist ismertetjük.

Az *absztrakt statikus analízis* olyan módszerek összessége, melyek a program futtatása

nélkül szolgáltatnak információt annak viselkedéséről. Ennek egyik alapja a program változóinak *absztrakt tartományok* (például számértékek helyett intervallumok vagy kongruencia osztályok) feletti ún. *absztrakt interpretációja*. Ezzel a módszerrel egyszerű tulajdonságok ellenőrizhetők. Ilyenek például a változók korlátellenőrzése, aritmetikai és puffer túlcsordulási hibák valamint érvénytelen mutatók kiszűrése, a programozó által a változók értékeire tett egyszerű feltételezések ellenőrzése. Előnye, hogy nagyméretű szoftverekre is alkalmazható, erőforrás igényét tekintve igen hatékony technika. Hátránya, hogy hamis riasztásokat is adhat, nem szolgáltat ellenpéldát, és csak viszonylag egyszerű tulajdonságok ellenőrzésére korlátozódik. Az absztrakt statikus analízist megvalósító modern eszközök például az Astrée [30], a CodeSonar [35], a Coverity [36] és a PolySpace [43].

1.4. Hardver verifikáció

Hardver rendszerek esetében a hibák időben való felfedezése és kijavítása életbevágó. Egyrészt, mind a hardver berendezések előállítási költségei, mind a velük szemben támasztott minőségi elvárások igen magasak. Másrészt, míg a szoftver hibák javítására a hibajavítások (patchek) és frissítések (update-ek) használatát mára már a felhasználók elfogadták, addig hardver termékeknél a piacra dobás utáni javítás igen körülményes, és általában csak a termék újragyártásával és a cseretermék vevőkhöz való eljuttatásával oldható meg, aminek igen magas költségei lehetnek. Mint említettük, a hibás Pentium processzorok cseréje az Intelnek körülbelül 475 millió dollárjába került.

A leggyakrabban használt hagyományos hardver verifikációs technikák a strukturális elemzés, az emuláció és a szimuláció.

A *strukturális elemzés* (structural analysis) több különböző specifikus technikát foglal magába, ezek közé sorolható a szintézis, az időzítés elemzés és az ekvivalencia ellenőrzés. Ezeket a módszereket itt nem ismertetjük.

Az *emuláció* egyfajta tesztelés. Ennek során először egy átkonfigurálható, általános célú hardvert, az *emulátort*, úgy konfigurálunk, hogy viselkedése a tervezett hardver áramkört utánozza, majd azt alapos tesztelésnek vetjük alá. Ahogy a szoftvertesztelésben, úgy itt is, az emuláció egyenértékű azzal, hogy az emulátorban megvalósított áramkört különböző bemeneteknek tesszük ki, majd a rájuk adott válaszokat összevetjük a hardver komponens specifikációjában leírtakkal. Ahhoz, hogy az áramkört teljes mértékben teszteljük, a rendszer összes lehetséges állapotában az összes lehetséges bemenetet figyelembe kell vennünk. Ez a gyakorlatban általában kivitelezhetetlen, ezért a tesztesetek számát lényegesen csökkentenünk kell, ami felderítetlen hibákhoz vezethet.

A *szimuláció* során a vizsgálandó áramkör szoftveres modelljét készítjük el, majd annak működését a *szimulátor* programmal vizsgáljuk. A modell megadására jellemzően valamilyen hardver leíró nyelvet használunk. Ilyen nyelv például a PSL [12], a Verilog [11] és a VHDL [10], mindhármat az IEEE (Institute of Electrical and Electronic Engineering, USA) szabványosította. A teszteseteket vagy a felhasználó adja meg, vagy automatikusan állítjuk elő, például véletlenszámgenerátor segítségével. Bármilyen eltérés a szimulátor kimenete és a specifikációban rögzített viselkedés között hibát jelent. A szimuláció olyan, mint a szoftvertesztelés, csak nem programokra, hanem modellekre alkalmazzuk. Ebből adódik, hogy ugyan-

azokkal a korlátokkal küzd: ahhoz, hogy 100%-ig megbizonyosodjunk a hibamentességről, jóval több esetet kellene megvizsgálnunk, mint amennyire a gyakorlatban képesek vagyunk.

A gyakorlatban a szimuláció a legnépszerűbb hardver verifikációs technika, és ezt több szinten is alkalmazzák. Például regiszterátviteli szinten, logikai kapu szinten és tranzisztor szinten. Ám ezeken a hibakeresési módszereken túl még mindig szükség van a kész termék *hardver tesztelésére* is, hiszen a gyártási hibák még tökéletes terv esetén is előfordulhatnak. A hardver elemek verifikációjában a *formális módszerek* szintén jelentős szerepet kapnak, és használatuk bizonyos területeken ma már mindennapi gyakorlattá vált. A továbbiakban ezekkel foglalkozunk.

1.5. Modell-ellenőrzés

Eddig röviden áttekintettük a hardver és szoftver rendszerek esetében hagyományosnak mondható verifikációs módszereket. Ezek mellett egyre inkább előtérbe kerülnek az úgynevezett *formális módszerek*. Ezek rendszerek tervezésében és verifikációjában alkalmazott, matematikai elméleteken alapuló módszerek, melyek a matematika szigorával és precizitásával segítenek a megbízhatóság megteremtésében.

Ide sorolhatók a különböző specifikációs nyelvek (mint például a B, Z, VDM és az Alloy), specifikációs formalizmusok (mint például az állapottérképek, az adatfolyam hálók és az absztrakt állapotgépek), a programozási nyelvek szemantikájának formális leírása, különböző számítási modellek (mint például a Petri hálók és a processzus algebrák), az automatikus tételbizonyítás, a programhelyesség bizonyítás (mint például a Floyd–Hoare kalkulus), a modell-ellenőrzés, a modell transzformáció, valamint az absztrakciós és finomítási technikák. Ezen módszerek egy része inkább a tervezés és fejlesztés folyamatát segíti, míg mások kifejezetten a verifikációt támogatják. Az utólagos vizsgálatok azt mutatják, hogy formális módszerek alkalmazásával felfedezhetők lettek volna azok a rejtett hibák, melyek az Ariane-5 rakéta felrobbanásához vagy a Therac-25 sugárterápiás készülék tragédiájához vezettek.

Mivel a fenti módszereknek a vázlatos ismertetése is meghaladja ezen jegyzet kereteit, a továbbiakban csak a fő témakörünket adó modell-ellenőrzéssel foglalkozunk részletesen. A formális rendszerek áttekintéséhez és megismeréséhez a [25] magyar nyelvű jegyzetet vagy a [2] angol nyelvű tanulmányt ajánljuk.

A *modell-ellenőrzés* olyan formális módszer, mely a vizsgálandó rendszer egy modelljéről és annak elvárt működését tartalmazó specifikációjáról a rendszermodell állapotainak (más szóval állapotterének) szisztematikus bejárásával dönti el, hogy a rendszermodell a specifikációt teljesíti-e vagy sem. Amennyiben a rendszermodell a specifikációt nem teljesíti, a modell-ellenőrzés ellenpéldát ad a specifikációt sértő működésre.

A modell-ellenőrzés a nyolcvanas évek elején indult útjára, két egymástól függetlenül dolgozó kutató páros, E. M. Clarke és E. A. Emerson [15], valamint J.-P. Queille és J. Sifakis [28] úttörő munkája nyomán. A modell-ellenőrzés hatékony, ipari környezetben is alkalmazható verifikációs technikává fejlesztéséért Clarke, Emerson és Sifakis 2007-ben megkapták az igen rangos, sokak által csak számítástechnikai Nobel-díjként emlegetett Turing-díjat. Ezt az elismerést az ACM (Association for Computing Machinery) évenként olyan személyeknek ítéli oda, akik kiemelkedően járultak hozzá a számítástechnika tudományának fejlődéséhez.

Magát a modell-ellenőrzés kifejezést először Clarke és Emerson használta. A tudományág első 25 éves történetét és fejlődését összegzi a [18] kötet.

A állapottér reprezentációja szerint megkülönböztetünk explicit modell-ellenőrzést és szimbolikus modell-ellenőrzést.

Explicit modell-ellenőrzés esetén minden állapotot explicit módon tárolunk a memóriában, az állapotokat közvetlenül indexeljük, és gráfalgoritmusokat alkalmazunk az állapottér bejárására. Explicit modell-ellenőrzést alkalmaz például a Java PathFinder [39], a CMC [34], a Zing [52] vagy a később ismertetésre kerülő SPIN [48] modell-ellenőrző program.

A *szimbolikus modell-ellenőrzéskor* ezzel szemben nem állapotokat, hanem állapothalmazokat tárolunk, és az ellenőrzés fixpont számítással történik. Az állapothalmazokat valamilyen szimbolikus reprezentáció segítségével ábrázoljuk. Ilyenek szimbolikus reprezentációt nyújtanak a bináris döntési diagrammok (BDD-k) véges halmazok vagy az automaták végtelen halmazok esetében. Ez a tárolási mód az explicit modell-ellenőrzésnél nagyságrendekkel hatékonyabb lehet, amennyiben az adott modell állapotterének sikerül tömör reprezentációját találnunk. Szimbolikus reprezentációt alkalmaz például a SAL [45], a NuSMV [42] és a szintén később ismertetésre kerülő UPPAAL [50] modell-ellenőrző.

1.6. A modell-ellenőrzés menete

A következőkben tekintsük át a modell-ellenőrzés alkalmazásának lépéseit. Az alábbi fázisokat különböztethetjük meg:

- *modellezési fázis*,
- *futtatási fázis*,
- *elemzési fázis*.

Ezen fázisok végrehajtásán felül fontos, hogy a modell-ellenőrzés egésze gondosan tervezett és megfelelően dokumentált legyen. Erről a *verifikáció szervezése* gondoskodik. Most röviden a fázisokat tekintjük át majd a verifikáció szervezéséről is ejtünk pár szót.

1.6.1. A modellezési fázis

A modell-ellenőrzésnek két bemenete van. Egyrészt szükségünk van a vizsgálni kívánt rendszer modelljére, másrészt a specifikációra, az ellenőrizni kívánt tulajdonságok leírására. Mindkét bemenetet pontos, egyértelmű módon kell megadnunk. Míg a rendszer modellje azt fogalmazza meg, hogy az *hogyan viselkedik*, addig a specifikáció azt írja le, hogy *mit kell* és *mit nem szabad* a rendszernek tennie.

A rendszer modelljét általában véges átmeneti rendszerekkel adjuk meg, melyek a jegyzet következő fejezetének központi témái. Ezek tulajdonképpen a véges automaták egy változatának tekinthetők. Az átmeneti rendszerek állapotokból és állapotok közti átmenetekből állnak. Az állapotok a rendszer pillanatnyi jellemzőit írják le, ilyenek például egy szoftver rendszer esetén a változók aktuális értéke vagy értékének valamely tartománya és a következő végrehajtandó utasítás címe (több folyamat esetén a végrehajtandó utasítások címei), a pufferek

értéke, stb. Az átmenetek pedig azt határozzák meg, hogyan változhat, léphet a rendszer egyik állapotból a másikba. A valós rendszerek leírása általában valamilyen modell leíró nyelven történik. Hardver rendszerek esetén ilyen a PSL, a VHDL és a Verilog hardver leíró nyelv. Szoftverek esetében pedig a C vagy Java nyelvhez hasonló, de a modellezést támogató modellező nyelveket használunk. Ilyen például az ötödik fejezetben részletesen is ismertetendő *Promela* (Process Meta Language), melyet a SPIN modell-ellenőrző használ.

Akár hardver, akár szoftver rendszerről van szó, a modell méretének kézben tartása a verifikáció sarkalatos pontja. Ehhez mindenekelőtt az *absztrakció* mértékének helyes megtalálása szükséges. Egy túl részletes, valósághű modell könnyen kezelhetetlen méretű állapottérhez vezethet, a túlságosan egyszerűsített pedig a verifikáció lényegét veszélyeztetheti. A szakirodalomban többször előfordul az a kijelentés, hogy a helyes absztrakció megtalálása igazából művészet. Különösen igaz ez konkurens rendszerek esetében, melyekre a modell-ellenőrzést előszeretettel alkalmazzák.

Automatikus absztrakció generálásra ad lehetőséget a *predikátum absztrakció ellenpélda alapú absztrakciófinomítással* (Counterexample-guided abstraction refinement, CEGAR), leírását lásd például [29]-ben. Ezt sikerrel használó eszközök a BLAST [32], a SLAM [47] és a MAGIC [41] modell-ellenőrzők.

A modellezés megvalósítása általában nem egy lélegzetvételre történik, hanem mielőtt a tényleges modell-ellenőrző futtatását elkezdenénk, szimulációkat végzünk a készülő modellel. Ez segíti a modell fejlesztését és velük kiküszöbölhetjük az egyszerű kezdeti modellezési hibákat még azelőtt, hogy az idő- és költségigényes utólagos verifikációra sor kerülne.

Ahhoz, hogy a verifikáció pontos és megbízható legyen, szükséges a vizsgálandó tulajdonságok precíz és egyértelmű megfogalmazása is. Ez valamilyen *tulajdonságokat leíró nyelven* történik. Ezek közül a legismertebbek a *temporális logikák*. Ezek az ítéletkalkulus olyan logikai műveletekkel való kiterjesztései, melyekkel a rendszer valós időbeli vagy logikai időbeli viselkedésére hivatkozhatunk. A verifikáció során a matematikai logika szóhasználatával élve azt kell ellenőriznünk, hogy a rendszer leírása (a véges átmeneti rendszer) *modellje-e* az elvárt tulajdonságokat megadó temporális logikai formuláknak. Innen származik a módszer *„modell-ellenőrzés”* elnevezése, nem pedig abból, hogy a rendszer modelljével dolgozunk, ami más verifikációs módszerekre is igaz, például a modell alapú tesztelésre vagy a modell alapú szimulációra.

A temporális logikák számos érdekes és fontos tulajdonságot tudnak kifejezni a vizsgálandó rendszerrel kapcsolatban. Például: *funkcionális helyességet*: tényleg azt csinálja a rendszer, amit tőle elvárunk?; *elérhetőséget*: például, lehet-e, hogy a rendszer holtpont állapotba kerül?; *biztonsági feltételeket*: „biztos, hogy valamilyen rossz esemény sohasem következhet be?”; *elevenségi tulajdonságokat*: „valami jó előbb-utóbb biztosan bekövetkezik?”; és *valós idejű tulajdonságokat*: a rendszer a megadott időben hajt-e végre egy adott tevékenységet?

A gyakorlatban sokszor komoly problémát jelent annak megítélése, hogy vajon a formalizált probléma (a modell + a specifikáció) valóban kielégítő leírása-e a aktuális verifikációs problémának, vagy esetleg attól eltér és nem is azt vizsgáljuk, amit célul tűztünk ki. Ezt a már említett validációs probléma. A tekintett rendszer komplexitása és a kiindulási, sokszor nem formális specifikáció pontatlansága miatt ezt gyakran nehéz pontosan megválaszolni. Mindenesetre a validáció nem keverendő a verifikációval. Modell alapú megközelítésben a verifikáció annak ellenőrzése, hogy a formalizált rendszer (a modell) megfelel-e azoknak a formalizált

követelményeknek (a specifikációnak), melyeket előzőleg elfogadtunk. Ezzel szemben a validáció azt vizsgálja, hogy a formális modell és specifikáció összhangban van-e a rendszerrel szemben támasztott nem formális tervezési elgondolásokkal.

1.6.2. A futtatási fázis

Amennyiben mind a modell, mind a specifikáció megadása megtörtént, akkor kezdődhet a szűkebb értelemben vett modell-ellenőrzés, vagyis a modell-ellenőrző program futtatása. Általában a specifikációt több formulával adjuk meg, és ezek ellenőrzése egyesével történik. A program a specifikációban megadott tulajdonságot a modell összes állapotában ellenőrzi. Amennyiben a tulajdonság nem teljesül a modellben, a modell-ellenőrző a tulajdonságot sértő ellenpéldát generál. Ezt a modell-ellenőrző rendszerhez tartozó szimulátorban tudjuk „visszajátszani”, elemezni, és így lehetőségünk van a hiba okának felderítésére.

Általában a valós rendszerek verifikációjához szükséges modellek mérete igen nagy. Ezért komoly korlát a modell-ellenőrző programok használatakor az a maximális állapotszám, melyet a rendelkezésre álló erőforrásokkal (pl. processzorok száma, memória mérete) még kezelni tudunk. Általában a memória mérete jelenti az erősebb megszorítást. Napjaink modell-ellenőrző szoftverei $10^8 - 10^{10}$ nagyságú állapottér teljes bejárására képesek, de speciális problémák esetében ügyes algoritmusok és testreszabott adatszerkezetek alkalmazásával nagyobb, akár 10^{20} (vagy volt már rá példa, hogy 10^{470}) állapotszámú rendszerek is verifikálhatók.

A modell-ellenőrzőnek számos kezdeti beállítást és direktívát adhatunk meg, melyekkel az állapottér bejárását, az alkalmazott redukciós technikákat, az ellenpéldák generálását, stb. vezérelhetjük. Ezek az ellenőrzés idő- és memóriaigényét is nagyban befolyásolhatják.

1.6.3. Az elemzési fázis

A futtatásnak alapvetően három lehetséges eredménye lehet: a specifikációban megadott tulajdonság a modellben vagy teljesül vagy nem, illetve előfordulhat még, hogy a vizsgált modell állapotainak bejárásához szükséges tár meghaladja a rendelkezésre álló fizikai memória méretét.

Ha egy tulajdonság teljesül a modellben, akkor továbbléphetünk a következő tulajdonság, pontosabban logikai formula ellenőrzésére. Ha a specifikáció minden formuláját ellenőriztük, a modell-ellenőrzés sikerrel zárult, a vizsgált modell az összes specifikációban rögzített tulajdonsággal rendelkezik.

Ha a vizsgált tulajdonság nem teljesül, a negatív eredménynek több különböző oka lehet. Lehetséges, hogy *modellezési hibával* állunk szemben. Ez azt jelenti, hogy a hiba elemzésénél azt vesszük észre, hogy nem a tervezett rendszerünk viselkedése a hibás, hanem a modell nem tükrözi hűen a rendszer viselkedését, és ezért nem felel meg a modell a specifikációnak. Ekkor a modellt kell javítanuk, majd a javított modellen újra végrehajtani a verifikációt. Fontos, hogy amennyiben bizonyos tulajdonságokat már ellenőriztünk, azokat a módosított modellben is verifikálnunk kell, hiszen a modell változtatása maga után vonhatja a már ellenőrzött tulajdonságok meghiúsulását.

Ha a hiba elemzése azt mutatja, hogy nincs indokolatlan különbség a rendszer és a modell között, akkor vagy *tervezési hibával* vagy *tulajdonság hibával* állunk szemben. Tervezési hi-

ba esetén a tervezett rendszert és természetesen vele együtt annak modelljét kell javítanunk. Az ilyen hibák felfedezése a modell-ellenőrzés alapvető feladata. De még az is lehetséges, hogy a hiba tanulmányozása során arra jutunk, hogy nem a rendszer viselkedésével van baj, hanem a formális specifikáció formulájával megfogalmazott tulajdonság nem felel meg annak a nem formális elvárásnak, amit a rendszertől meg kell követelnünk. Ilyenkor természetesen a formulát kell kijavítani. Az, hogy a javítás megfelelő-e vagy sem, már a validáció témakörébe tartozik. Ezután újra futtatnunk kell a modell-ellenőrzőt a módosított formulával. Mivel a javítás során a modell nem változott, a tulajdonság hibák nem érintik a korábban már sikeresen ellenőrzött tulajdonságok helyességét. A verifikáció akkor ér véget, ha sikerül az összes kívánt tulajdonság fennállását az ellenőrzéssel bizonyítanunk.

Sajnos, sokszor a modell túl nagy ahhoz, hogy a rendelkezésre álló erőforrásokkal kezelni tudjuk. A valós rendszerek állapotszáma könnyen több nagyságrenddel nagyobb lehet, mint amit a jelenleg elérhető memória kapacitással tárolni tudunk, és nem valószínű, hogy ez a jövőben megváltozik. Ha a memória a modell-ellenőrzéshez kevésnek bizonyul, több úton haladhatunk tovább. Az egyik, hogy megpróbálunk olyan technikákat alkalmazni, melyek a modell struktúrájában rejlő szabályosságokat ki tudják használni. Példa erre az állapottér szimbolikus technikákkal, például bináris döntési diagramokkal való reprezentálása, vagy a részleges rendezési redukció alkalmazása. Ha ez nem vezet eredményre, a rendszer modelljének további absztrakciójával csökkenthetjük annak méretét. Fontos, hogy az absztrakciónak csak a vizsgálandó tulajdonság teljesülését/nem teljesülését kell megőriznie. Ezért gyakran kaphatunk jóval kisebb modellt, ha csak egy tulajdonságot kell figyelembe vennünk. Így, ha szükséges, minden tulajdonság ellenőrzéséhez külön modellt készíthetünk, mégpedig az ellenőrizendő tulajdonságnak megfelelő, testreszabott absztrakció választásával.

A harmadik megközelítés, melyet túl nagy állapottér esetén alkalmazhatunk, a verifikáció precizitásának a feladása. A *valószínűségi ellenőrzéssel* lemondunk az állapottér teljes bejárásáról. Ha ezt jól meghatározott keretek közt tesszük, az állapottér lefedettségére nézve kis (gyakran elhanyagolható) áldozat árán a verifikációt a gyakorlatban kivitelezni tudjuk. Valószínűségi modell-ellenőrzést valósít meg például a PRISM [44] és a KRONOS [40] modell-ellenőrző.

Egy másik lehetőség, hogy az állapotteret csak korlátos mélységig vizsgáljuk, ez vezet a *korlátos modell-ellenőrzéshez*. Ebben az esetben legcélszerűbb, ha az állapotokat Boole változókkal írjuk le, és az ellenőrzést az ítéletformulák kielégíthetőségének problémájára (SAT-ra) vezetjük vissza. A SAT probléma kapott példányának megoldására általános vagy kifejezetten korlátos modell-ellenőrzés céljára kifejlesztett SAT megoldó programot használhatunk. Ezzel a technikával sokszor sikerrel tudunk az állapotrobbanás problémájával megküzdeni, és a módszer kiválóan alkalmas a *felszíni hibák* (shallow bugs) kiszűrésére. Ugyanakkor mély, többszörösen egymásba ágyazott ciklikus programok esetén nem teljes a módszer, azaz csak a vizsgált korlátos mélységig tudja a hibamentességet garantálni. Ezt a technikát alkalmazza többek között a CBMC [33], az F-Soft [38] és a SATURN [46], melyek C programok verifikációjára alkalmasak.

1.6.4. A verifikáció szervezése

Az egész modell-ellenőrzés folyamatának jól szervezettnek, strukturáltnak és előre tervezettnek kell lennie. A modell-ellenőrzés ipari alkalmazásai arra mutatnak rá, hogy a verzió- és konfigurációmenedzsment különösen fontos. A verifikáció folyamán számos, a rendszer különböző komponenseit leíró modellt készítünk. Ezeknek a modelleknek az esetenként eltérő absztrakciós szintekből és a többszöri javításokból adódóan különböző változatai lesznek. Hasonló a helyzet a specifikációt megfogalmazó formulákkal. Ezen kívül számon kell még tartanunk számos verifikációs paramétert (például a modell-ellenőrző szoftver beállításait) és az eredményeket (az ellenpéldák leírásait, a futási statisztikákat, stb.) Nem nehéz belátni, hogy ezeket az információkat gondosan rendezni, karbantartani és dokumentálni kell annak érdekében, hogy a gyakorlatban kézben tarthassuk a modell-ellenőrzés menetét, és biztosítsuk az ellenőrzés reprodukálhatóságát.

1.7. A modell-ellenőrzés előnyei és hátrányai

Ebben a részben [5] alapján ismertetjük a modell-ellenőrzés előnyeit és hátrányait más verifikációs technikákkal szemben.

1.7.1. A modell-ellenőrzés előnyei

- A modell-ellenőrzés *általános* verifikációs módszer, mely széles körben alkalmazható. Egyaránt alkalmas hardver elemek, szoftver termékek és beágyazott rendszerek ellenőrzésére.

- A modell-ellenőrzés támogatja a *parciális verifikációt*, azaz a vizsgált tulajdonságok egyenként ellenőrizhetők, megengedve, hogy először a leglényegesebb tulajdonságokra koncentráljunk. Nem szükséges, hogy a teljes, minden esetre kiterjedő specifikáció rendelkezésünkre álljon.

- A modell-ellenőrzés *alkalmas olyan hibák felderítésére is, melyek csak igen ritkán, nagyon speciális körülmények között jelentkeznek*. Ezért a modell-ellenőrzés különösen jól kiegészíti a szimulációval és a teszteléssel végrehajtott verifikációt, melyek ezzel szemben elsősorban a gyakori hibákat találják meg.

- A modell-ellenőrzés *diagnosztikus információkat* (tipikusan ellenpéldákat) is szolgáltat, amennyiben a specifikáció valamely tulajdonsága nem teljesül, ezzel segítve a hibakeresést.

- Potenciálisan *gombnyomásra működő* („push-button”) technológia, használatához sem nagyfokú szakértői tapasztalat, sem felhasználói interaktív beavatkozás nem szükséges, szemben például a tételbizonyításon alapuló módszerekkel.

- A modell-ellenőrzés *egyre növekvő érdeklődést élvez az ipari partnerek részéről*. Számos hardver cég már kialakította belső verifikációs laboratóriumát. Egyre nagyobb

számban jelennek meg modell-ellenőrzési tapasztalatot megkövetelő állásajánlatok és modell-ellenőrzést (is) nyújtó kereskedelmi szoftverek.

- *Könnyen integrálható már létező hardver-szoftver fejlesztési ciklusokba*. Tanulási görbéje nem túl meredek, a gyakorlati tanulmányok azt mutatják, hogy használatával a fejlesztés egészére nézve időt tudunk megtakarítani.

- *Szilárd matematikai elméleteken alapul*, mint például a gráfelméleti algoritmusok, az adatszerkezetek elmélete, és a matematikai logika.

1.7.2. A modell-ellenőrzés hátrányai

- A modell-ellenőrzés sokkal *inkább vezérlés-intenzív, mintsem adat-intenzív rendszerekre alkalmazható*, mert az utóbbiakban általában végtelen, vagy igen nagy az adatok értelmezési tartománya, így a vizsgálandó állapottér mérete is.

- A modell-ellenőrzés alkalmazhatósága *eldönthetetlenségi eredményekbe ütközhet* végtelen állapotszámú rendszerek vagy absztrakt adattípusok vizsgálata esetén, mivel ezek verifikációja eldönthetetlen vagy csak félig eldönthető logikák használatát követelheti meg.

- Az ellenőrzés legtöbbször a rendszer modelljére és nem magára a valós rendszerre (a termékre vagy a prototípusra) vonatkozik, így az ellenőrzés eredménye is *legfeljebb annyira lehet pontos, amennyire a rendszer modellje az*. Ezért kiegészítő ellenőrzésre mindig szükség van, hardver termékeknél a gyártási hibák, szoftvereknél pedig a kódolás során keletkezett hibák kiszűrése érdekében.

- A módszer *csak a vizsgálat során felírt tulajdonságokat ellenőrzi*, melyek teljességére és helyességére nincs garancia. A nem vizsgált tulajdonságok teljesülését így nem tudjuk megítélni.

- A modell-ellenőrzés gyakorlati alkalmazásának legnagyobb gátja *az állapotszám robbanás problémája*, azaz, hogy a rendszer pontos modellezéséhez szükséges állapotok száma könnyen meghaladhatja a rendelkezésre álló memória méretéből adódó korlátot. Annak ellenére, hogy ennek leküzdésére több igen hatékony módszer született, a realisztikus modellek még mindig túlságosan nagyok lehetnek ahhoz, hogy elférjenek a memóriában.

- A modell-ellenőrzés használata *bizonyos szakértelmet követel*. Erre különösen szükség van a megfelelő absztrakció megtalálásához, mely még kezelhető méretű, de ugyanakkor valósághű modellekhez vezet, és a követelmények formalizálásához, mely logikai formalizmusok ismeretét igényli.

- A verifikáció *megvalósítása modell-ellenőrző szoftverrel történik, mely maga is hibás lehet*, így a végeredmény helyessége nem garantált. Igaz, bizonyos fejlett modell-ellenőrző eljárások egyes részeinek helyességét már sikerült formálisan, tételbizonyítók segítségével igazolni, megoldva ezzel (legalábbis részben) a problémát.

- A tételbizonyítástól eltérően, a modell-ellenőrzés *nem engedi meg az általánosítások igazolását*, ezért tetszőleges sok komponenssel rendelkező, vagy parametrizált rendszerek nem ellenőrizhetők.

2. fejezet

Konkurens rendszerek modellezése

2.1. Átmeneti rendszerek

Ahogy az előző fejezetben láttuk, a verifikáció a tervezett vagy vizsgálni kívánt rendszer modelljére épül. Ehhez először el kell készítenünk az alkalmas modellt. Ez egyrészt alapvető jelentőségű az egész verifikáció sikere szempontjából, másrészt igen nehéz feladat. Ugyanis, ha túl egyszerű modellt készítünk, lényeges részleteket veszíthetünk el, és a verifikáció nem szolgáltat releváns adatokat a rendszer helyességéről. Ha pedig túl részletes modellt készítünk, könnyen előfordulhat, hogy a modell komplexitása és nagysága meghiúsítja a verifikáció rendelkezésre álló hardveren történő kivitelezését.

Ebben a fejezetben az átmeneti rendszerekkel ismerkedünk meg, melyeket széles körben használnak modellezésre. Maga az átmeneti rendszer kifejezés azt hangsúlyozza, hogy ezek olyan formális rendszerek, melyek *állapotokból* és köztük definiált *átmenetekből* állnak, nem pedig nyelvek felismerésére alkalmazott gépek. Az átmenetek közt megkülönböztethetünk egyrészt a modellezett folyamat által végrehajtott *akciókat* (actions), másrészt külső *eseményeket* (events).

Az átmeneteket kiegészítő információt hordozó címkékkel is elláthatjuk, ekkor kapjuk az ún. *címkézett átmeneti rendszereket*. Ez nagyban segíti a modellek értelmezését.

Paraméterezett átmeneti rendszerek esetében mind az állapotok, mind az átmenetek bizonyos (általában véges halmazból kikerülő) paraméterekkel rendelkezhetnek. Így például megjelölhetünk kezdő- és végállapotokat, illetve további tulajdonságokkal ruházhatjuk fel őket, például mind az állapotokat, mind az átmeneteket kritikus szekciókba sorolhatjuk. Hasonlóképpen különböztethetjük meg a bemenetként és kimenetként értelmezendő átmeneteket.

Ez a fejezet az alábbi könyvek alapján készült: [4], [1], [13], [5] és [25].

2.1.1. Egyszerű átmeneti rendszerek

Definíció. Átmeneti rendszer alatt egy $\mathcal{A} = (S, T, \alpha, \beta)$ négyest értünk, ahol S az állapotok véges vagy végtelen halmaza, T az átmenetek véges vagy végtelen halmaza, α és β pedig két T-ből S-be képező függvény, melyek tetszőleges T-beli t átmenethez annak $\alpha(t)$ kezdő- és $\beta(t)$ végpontját adják meg.

A továbbiakban az átmeneti rendszert *egyszerű átmeneti rendszerként* is említjük, de ezt

csak akkor fogjuk használni, ha a címkézett és paraméterezett átmeneti rendszerektől való különbségüket hangsúlyozni szeretnénk.

Példa. Modellezzünk egy italautomatát, mely először rögzített összegű pénz bedobására, majd ez után egy gomb megnyomására vár. Két gombja van. Ha a pénz bedobása után a K gombot nyomják meg, az automata kávét ad, ha pedig a T gombot nyomják meg, teát szolgáltat. Ezután az automata alapállapotba kerül, és újabb pénz bedobására vár. Az említett italautomatát a következő $\mathcal{A}_1 = (S_1, T_1, \alpha_1, \beta_1)$ átmeneti rendszerrel modellezhetjük: Az állapothalmaz $S_1 = \{s_1, s_2, s_3, s_4\}$, melynek jelentése lehet:

- s_1: az automata pénz bedobására várakozik;
- s_2: az automata valamelyik gomb megnyomására vár;
- s_3: a K gombot nyomták meg, így az automata kávét fog adni;
- s_4: a T gombot nyomták meg, így az automata teát fog adni.

Az átmenetek legyenek az alábbiak, $T_1 = \{t_1, t_2, t_3, t_4\}$, melyekre

$$\alpha_1(t_1) = s_1,\ \alpha_1(t_2) = s_2,\ \alpha_1(t_3) = s_2,\ \alpha_1(t_4) = s_3,\ \alpha_1(t_5) = s_4,$$
$$\beta_1(t_1) = s_2,\ \beta_1(t_2) = s_3,\ \beta_1(t_3) = s_4,\ \beta_1(t_4) = s_1,\ \beta_1(t_5) = s_1.$$

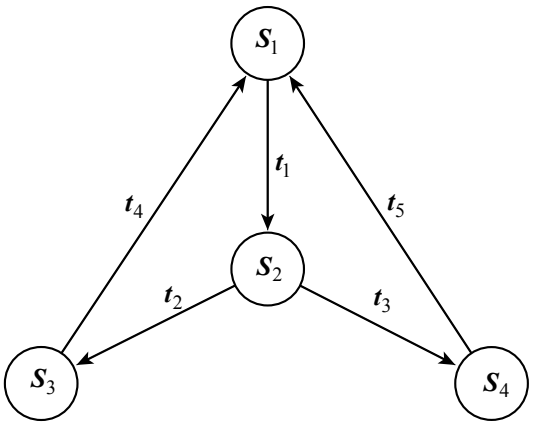
Ezeknek a következő szemléletes jelentést tulajdoníthatjuk:

- t_1: pénz bedobása;
- t_2: a K gomb megnyomása;
- t_3: a T gomb megnyomása;
- t_4: az automata kávét ad;
- t_5: az automata teát ad.

Az átmeneti rendszereket grafikusan is ábrázolhatjuk, mint az a 2.1 ábrán látható. Szokásos módon az állapotokat körök jelölik, az átmeneteket pedig a kezdőpontjukból a végpontjukba mutató nyilak. A grafikus ábrázolás segítségével könnyen áttekinthetjük a modellezett rendszert és annak lehetséges viselkedését.

2.1.2. Utak átmeneti rendszerekben

Definíció. Az $\mathcal{A} = (S, T, \alpha, \beta)$ átmeneti rendszerben $n > 0$ hosszú *útnak* nevezzünk egy $t_1, \ldots, t_n$ sorozatot, amennyiben $\beta(t_i) = \alpha(t_{i+1})$ teljesül, minden i-re, $1 \leq i \leq n$ esetén. Az utakat a továbbiakban pusztán az átmenetek leírásával, tehát vesszők nélkül fogjuk jelölni, így: $t_1 t_2 \ldots t_n$. Ennek az útnak a *kezdőpontját* $\alpha(t_1 \ldots t_n) = \alpha(t_1)$, *végpontját* pedig $\beta(t_1 \ldots t_n) = \beta(t_n)$-ként definiáljuk. Tehát az út kezdőpontja a kiinduló élének a kezdőpontja, az út végpontja pedig az utolsó élének végpontja. *Végtelen úton* olyan $t_1, \ldots, t_n, \ldots$ végtelen sorozatot értünk, melyre $\beta(t_i) = \alpha(t_{i+1})$ minden $i \geq 1$-re fennáll. A végtelen utak jelölése szintén vesszők



2.1. ábra. Az $\mathcal{A}_1$ átmeneti rendszer

nélkül történik: $t_1 t_2 \ldots$. Végtelen utaknak a végpontját nem, csak *kezdőpontját* értelmezzük, ami $\alpha(t_1 \ldots) = \alpha(t_1)$. Az $\mathcal{A} = (S, T, \alpha, \beta)$ átmeneti rendszerben a véges illetve végtelen *utak halmazát* T^+, illetve T^ω fogja jelölni.

A későbbi fejezetekben néha egy olyan $s_0, s_1, \ldots$ véges vagy végtelen állapot sorozatot is útnak nevezünk, melyben s_i-ből s_{i+1}-be vezet átmenet minden $i = 0, 1, \ldots$ esetén.

Példa. Az előző példánkban az $\mathcal{A}_1$ átmeneti rendszerben $c_1 = t_4 t_1 t_3 t_5 t_1$ utat alkot, melynek kezdőpontja $\alpha(c_1)=s_3$, végpontja pedig $\beta(c_1)=s_2$. Megjegyezzük, hogy a gráfokban a szokásos módon értelmezett útfogalommal ellentétben átmeneti rendszerekben egy átmenet (mint a példában t_1) többször is szerepelhet egy úton.

Definíció. Legyen $c=t_1 \ldots t_n$ egy véges, $c'=t'_1 \ldots$ pedig egy véges vagy végtelen út. Amennyiben a c út végpontja megegyezik a c' út kezdőpontjával, azaz $\beta(c) = \alpha(c')$, akkor c és c' *összefűzését* vagy *konkatenációját* így értelmezzük:

$$c \cdot c' = t_1 \ldots t_n t'_1 \ldots$$

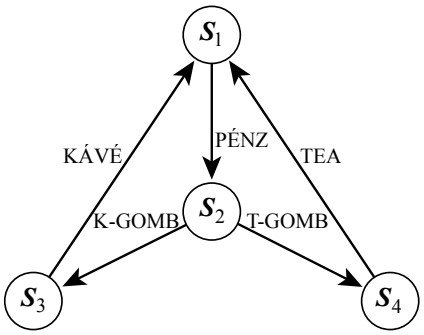
Amennyiben a $\beta(c) = \alpha(c')$ feltétel nem teljesül, c és c' összefűzését nem definiáljuk.

Definíció. Egy átmeneti rendszer minden s állapotára jelölje ε_s az s-ből s-be vezető *üres utat*. Azaz $\alpha(\varepsilon_s) = \beta(\varepsilon_s) = s$, és $c \cdot \varepsilon_s = c$, illetve $\varepsilon_s \cdot c' = c'$, minden olyan c és c' útra, melyre az összefűzés értelmezve van.

2.1.3. Címkézett átmeneti rendszerek

Definíció. Legyen A egy véges vagy végtelen nemüres halmaz, melyet a továbbiakban *ábécéként* is említünk. Az $\mathcal{A} = (S, T, \alpha, \beta, \lambda)$ ötöst A feletti *címkézett átmeneti rendszernek* nevezzük, ha (S, T, α, β) egyszerű átmeneti rendszer, λ pedig $T \to A$ leképezés, mely minden átmenethez egy *címkét* rendel az A halmazból. Egy ilyen címkét gyakran *akciónak* vagy *eseménynek* is nevezünk majd.

Példa. Lássuk el az $\mathcal{A}_1$ egyszerű átmeneti rendszert címkézéssel. Ehhez válasszuk címkehalmaznak az A_1 = {PÉNZ, K-GOMB, T-GOMB, TEA, KÁVÉ} ötelemű halmazt, továbbá legyen a $\lambda_1 : T \to A_1$ címkézés a következő: $\lambda_1(t_1)$ = PÉNZ, $\lambda_1(t_2)$ = K-GOMB, $\lambda_1(t_3)$ = = T-GOMB, $\lambda_1(t_4)$ = KÁVÉ, $\lambda_1(t_5)$ = TEA. Az így kapott $\mathcal{A}'_1 = (S_1, T_1, \alpha_1, \beta_1, \lambda_1)$ címkézett



2.2. ábra. Az $\mathcal{A}'_1$ címkézett átmeneti rendszer

átmeneti rendszert a 2.2 ábrán láthatjuk. A címkézés több területen is hasznos. Egyrészt, ha azonos címkét adunk azoknak az átmenetnek, melyek a verifikáció szempontjából hasonló szerepet töltenek be, akkor ezekre az átmenetekre egyszerre tudunk hivatkozni. Például, ha a "TEA" és "KÁVÉ" helyett az "ITAL" címkét használjuk, akkor egyszerűbb megfogalmazni azt a specifikációt, hogy az automata nem ad italt pénz bedobása nélkül. Másrészt – és ez jóval fontosabb –, majd lehetőségünk lesz különböző átmeneti rendszerek átmenetei között a címkézés segítségével kapcsolatokat létrehozni, így például bizonyos átmenetek szinkronizációját megkövetelni.

A továbbiakban feltételezzük, hogy címkézett átmeneti rendszerekben nem lehet két olyan átmenet, melyeknek kezdőpontja, végpontja és címkéje is azonos. Másként fogalmazva, ez a három adat egyértelműen meghatározza az átmenetet. Ezt kihasználva, azt a t átmenetet, melyre $\alpha(t) = s$ és $\beta(t) = s'$, továbbá $\lambda(t) = a$ teljesül röviden a

$$t : s \mapsto a \to s'$$

jelöléssel fogjuk megadni.

Definíció. Amennyiben $c = t_1 t_2 \ldots$ egy véges vagy végtelen út egy $\mathcal{A} = (S, T, \alpha, \beta, \lambda)$ címkézett átmeneti rendszerben, a c út *nyomát* az alábbiak szerint értelmezzük:

$$\mathrm{trace}(c) = \lambda(t_1)\lambda(t_2)\ldots$$

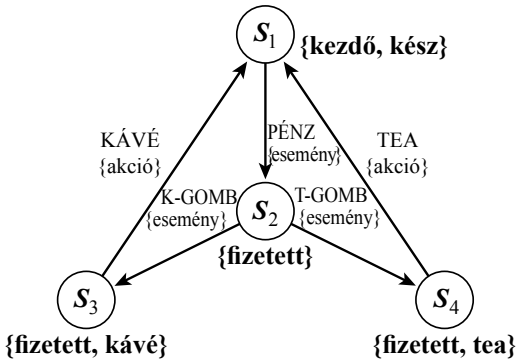
Példa. A $c_1 = t_4 t_1 t_3 t_5 t_1$ út nyoma az $\mathcal{A}'_1$ címkézett átmeneti rendszerben,

$$\mathrm{trace}(c_1) = \text{KÁVÉ PÉNZ T-GOMB TEA PÉNZ.}$$

2.1.4. Paraméterezett átmeneti rendszerek

A címkézett átmeneti rendszereknél még általánosabb a paraméterezett átmeneti rendszerek fogalma, mely a következő:

Definíció. Legyen $\mathcal{X} = \{x_1, \ldots, x_n\}$ és $\mathcal{Y} = \{y_1, \ldots, y_m\}$ két véges halmaz. A továbbiakban $\mathcal{X}$ elemeit *állapotparamétereknek*, $\mathcal{Y}$ elemeit pedig *átmenetparamétereknek* hívjuk. Az $\mathcal{A} = (S, T, \alpha, \beta, S_{x_1}, \ldots, S_{x_n}, T_{y_1}, \ldots, T_{y_m})$ rendszert $(\mathcal{X}, \mathcal{Y})$*-paraméterezett átmeneti rendszernek* nevezzük, ha $\mathcal{A} = (S, T, \alpha, \beta)$ egyszerű átmeneti rendszer, továbbá $S_{x_i} \subseteq S$ és $T_{y_j} \subseteq T$ teljesül minden $1 \leq i \leq n$, illetve $1 \leq j \leq m$-re.



2.3. ábra. Példa paraméterezett átmeneti rendszerre

A paraméterezés elsősorban arra jó, hogy a verifikációhoz tulajdonsággal lássuk el az állapotokat és átmeneteket. Ha egy $x \in \mathcal{X}$ paraméterrel az s állapot rendelkezik, azaz $s \in S_x$, akkor ezt úgy értelmezhetjük, hogy s rendelkezik az x tulajdonsággal. Így például a *kezdő* paraméterrel jelölhetjük meg a rendszer lehetséges kezdőállapotait, melyeket az $S_{kezdő}$ halmazba gyűjtünk össze. Amennyiben a tulajdonság nem egy egyszerű igen/nem válasszal, hanem mondjuk egy k változó 1, 2 vagy 3 értékével írható le, akkor erre az $x_{k=1}$, $x_{k=2}$ és $x_{k=3}$ paramétereket használhatjuk. Egy egyszerű példát láthatunk a 2.3 ábrán. Az állapotparaméterek: $\mathcal{X} = \{$**kezdő, kész, fizetett, kávé, tea**$\}$, az átmenetparaméterek pedig: $\mathcal{Y} = \{$akció, esemény$\}$.

Az ábrán még az előző példából örökölt címkéket is feltüntettük, noha ez definíció szerint nem részei paraméterezett átmeneti rendszernek. Mivel ehhez példához hasonlóan a címkézés nyújtotta előnyökről nem szeretnénk lemondani, a továbbiakban bemutatott átmeneti rendszerek gyakran egyszerre címkézett *és* paraméterezett átmeneti rendszerek lesznek.

Könnyen látható, hogy az egyszerű átmeneti rendszereknél általánosabb a címkézett átmeneti rendszerek és még általánosabb a paraméterezett átmeneti rendszerek fogalma. Valóban, minden $\mathcal{A} = (S, T, \alpha, \beta)$ egyszerű átmeneti rendszerből közvetlenül származtathatjuk azt az $\mathcal{A}'$ címkézett átmeneti rendszert, melynek címkehalmaza T, és minden átmenet önmagával van címkézve. Azaz $\mathcal{A}' = (S, T, \alpha, \beta, \lambda)$, ahol $\lambda : T \to T$ az identikus leképezés.

Hasonló módon minden, valamely $A = \{a_1, \ldots, a_n\}$ halmazzal címkézett, $\mathcal{A}' = (S, T, \alpha, \beta, \lambda)$ átmeneti rendszer olyan $\mathcal{A}'' = (S, T, \alpha, \beta, T_{a_1}, \ldots, T_{a_n})$ $(\emptyset, A)$-paraméterezett átmeneti rendszernek tekinthető, melyben minden átmenet a saját címkéjének megfelelő paraméterrel van ellátva, azaz $T_{a_i} = \{t \in T \mid \lambda(t) = a_i\}$.

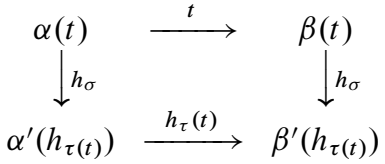
2.2. Átmeneti rendszerek homomorfizmusai

Definíció. Legyen $\mathcal{A} = (S, T, \alpha, \beta)$ és $\mathcal{A}' = (S', T', \alpha', \beta')$ két átmeneti rendszer. $\mathcal{A}$-ból $\mathcal{A}'$-be ható *homomorfizmus* alatt egy olyan $h = (h_\sigma, h_\tau)$ leképezéspárt értünk, melyre:

$$h_\sigma : S \to S',$$
$$h_\tau : T \to T',$$

és minden T-beli t átmenetre teljesül, hogy

$$\alpha'(h_\tau(t)) = h_\sigma(\alpha(t)), \text{ és}$$
$$\beta'(h_\tau(t)) = h_\sigma(\beta(t)).$$



2.4. ábra. A $h : \mathcal{A} \to \mathcal{A}'$ homomorfizmus szemléltetése

A homomorfizmus fogalmát a 2.4 ábra szemlélteti. A $h : \mathcal{A} \to \mathcal{A}'$ homomorfizmus úgy felelteti meg az $\mathcal{A}$ átmeneti rendszer állapotait és átmeneteit az $\mathcal{A}'$ átmeneti rendszer bizonyos állapotainak és átmeneteinek, hogy ha $\mathcal{A}$-ban van átmenet két állapot között, akkor $\mathcal{A}'$-ben is kell, hogy legyen átmenet ezen két állapot képe között, és az egyik ilyen átmenetet kell a homomorfizmusnak az eredeti átmenethez rendelnie. A homomorfizmus lényeges tulajdonsága, hogy mindaz a viselkedés, ami egy átmeneti rendszerben megfigyelhető, megtapasztalható annak homomorf képében is. Visszafele ez nem feltétlenül igaz.

A címkézett átmeneti rendszerek közötti homomorfizmusnak ezen felül még tiszteletben kell tartania az átmenetek címkézését is. Ez alatt azt értjük, hogy az egymásnak megfeleltetett átmenetek címkéje azonos kell, hogy legyen.

Definíció. Legyen $\mathcal{A} = (S, T, \alpha, \beta, \lambda)$ és $\mathcal{A}' = (S', T', \alpha', \beta', \lambda')$ azonos A címkehalmaz feletti, két címkézett átmeneti rendszer. $\mathcal{A}$-ból $\mathcal{A}'$-be ható *címkézett átmeneti rendszer homomorfizmus* alatt egy olyan $h = (h_\sigma, h_\tau)$ leképezéspárt értünk, mely homomorfizmus (S, T, α, β) és $(S', T', \alpha', \beta')$ között, továbbá minden T-beli t átmenetre teljesíti a

$$\lambda'(h_\tau(t)) = \lambda(t)$$

feltételt.

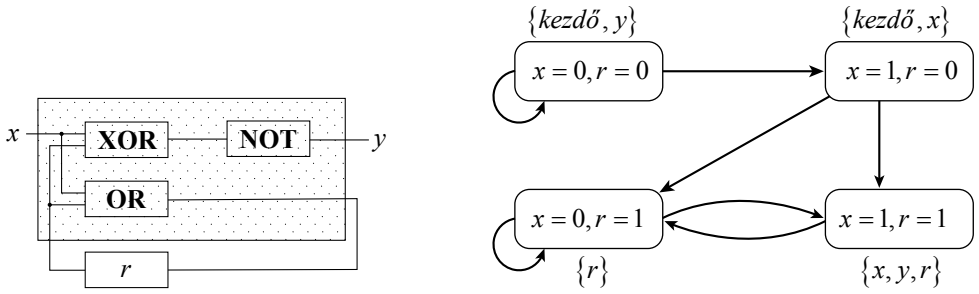
Paraméterezett átmeneti rendszerek esetében a homomorfizmus fogalma az alábbiak szerint alakul:

Definíció. Ha $\mathcal{A} = (S, T, \alpha, \beta, S_{x_1}, \ldots, S_{x_n}, T_{y_1}, \ldots, T_{y_m})$ valamint $\mathcal{A}' = (S', T', \alpha', \beta', S'_{x_1}, \ldots, S'_{x_n}, T'_{y_1}, \ldots, T'_{y_m})$ azonos $(\mathcal{X}, \mathcal{Y})$ paraméterhalmaz feletti paraméterezett átmeneti rendszerek, akkor $\mathcal{A}$-ból $\mathcal{A}'$-be ható *paraméterezett átmeneti rendszer homomorfizmus* alatt egy olyan $h = (h_\sigma, h_\tau)$ leképezéspárt értünk, mely homomorfizmus (S, T, α, β) és $(S', T', \alpha', \beta')$ között, továbbá teljesíti az alábbi két tulajdonságot:

$$\forall s \in S, \forall x \in \mathcal{X} : \left(s \in S_x \iff h_\sigma(s) \in S'_x\right)$$
$$\forall t \in T, \forall y \in \mathcal{Y} : \left(t \in T_y \iff h_\tau(t) \in T'_y\right).$$

Ez azt jelenti, hogy akkor és csak akkor rendelkezik valamely állapotparaméterrel vagy átmenetparaméterrel egy állapot, illetve átmenet, ha a képe is rendelkezik a kérdéses paraméterrel.

Egyszerű vagy címkézett átmeneti rendszerek közötti *izomorfizmus* alatt olyan homomorfizmust értünk, mely bijektív (azaz kölcsönösen egyértelmű) leképezés. Két átmeneti rendszert *izomorfnak* nevezünk, ha létezik köztük izomorfizmus. Mivel az izomorf átmeneti rendszerek egymástól csak az állapotok és átmenetek elnevezésében különböznek, nincs okunk rá, hogy köztük különbséget tegyünk.



2.5. ábra. Egy sorrendi hálózat és paraméterezett átmeneti rendszer modellje

2.3. Példák átmeneti rendszerekre

Ebben a fejezetben néhány példával illusztráljuk, hogy az átmeneti rendszerek természetes módon alkalmazhatók különböző hardver és szoftver rendszerek modellezésére. Először egyszerű elemeket tekintünk, majd később lesz szó arról, hogyan építhetünk fel ezekből az elemekből komplex rendszereket.

2.3.1. Szekvenciális áramkörök

Tekintsük a 2.5 ábra bal oldalán látható egyszerű szekvenciális (más szóval sorrendi) hálózatot. A hálózat változói a következők:

- x: bemeneti változó,
- y: kimeneti változó, és
- r: regiszter.

Mivel sorrendi áramkörről van szó, a kimeneti y változó nem csak az x bemenettől, hanem az r regiszter értékétől is függ. A függést most a

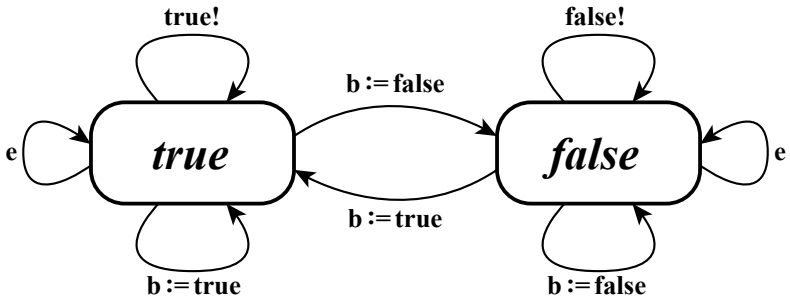
$$\lambda_y = \neg(x \oplus r)$$

képlet írja le, ahol $\oplus$ a XOR, más néven *kizáró vagy* függvényt jelöli. Az r regiszter értéke, melyet majd y következő értékének meghatározásánál használunk, az alábbi függvény szerint változik:

$$\delta_r = x \vee r.$$

Vegyük észre, hogy amennyiben az r regiszterbe 1 érték kerül, a regiszter mindvégig megtartja ezt az értékét.

Tegyük fel, hogy kezdetben r értéke 0, az x bemeneti változó értékét azonban nem ismerjük. Mivel sorrendi hálózatról van szó, a rendszer egy állapotát az x és r aktuális értékével írhatjuk le. Az állapotokat jelöljük úgy, hogy a változók értékét leíró egyenleteket szögletes zárójelbe tesszük. Így a rendszernek két lehetséges kezdőállapota van $[x = 0, r = 0]$ és $[x = 1, r = 0]$. Az átmeneteket, pontosabban bármely állapot rákövetkező állapotát meghatározza egyrészt az x bemenet következő értéke (erre a rendszernek nincs befolyása, mind az $x = 0$ és



2.6. ábra. A **b** *Boole változó $\mathcal{B}$ címkézett átmeneti rendszer modellje*

$x = 1$ lehetőségeket figyelembe kell venni), másrészt az r regiszter δ_r által kiszámított értéke. Például az $[x = 1, r = 0]$ állapotból egy átmenettel az $[x = 0, r = 1]$ vagy az $[x = 1, r = 1]$ állapotba juthatunk. Az első eset akkor következik be, ha a következő input bit értéke 0, a második akkor, ha a következő input bit 1. Az r regiszter következő értéke mindkét esetben x és r előző értékéből számolandó, ami $1 \vee 0 = 1$-et ad eredményül.

Mivel minden átmenet azonos szerepet tölt be, nincs értelme az átmenetek címkézésének. Az állapotokat viszont érdemes paraméterekkel ellátni például a következő módon. Minden állapothoz rendeljük hozzá paraméterként azoknak a változóknak a nevét, melyek az állapotban igaz értéket vesznek föl. Ezen felül vegyük még fel a *kezdő* paramétert a kezdőállapotok szokásos jelölésére. Így az állapotparaméterek halmaza $\mathcal{X} = \{kezdő, x, y, r\}$ lesz, és például $S_y = \{[x = 0, r = 0], [x = 1, r = 1]\}$, mert a kimeneti y változó $\lambda_y = \neg(x \oplus r)$ szerinti értéke csak ebben a két állapotban igaz. A teljes paraméterezett átmeneti rendszer a 2.5 ábra jobb oldalán látható.

Könnyen ellenőrizhető, hogy ezzel a módszerrel tetszőleges (közömbös bitek nélküli) sorrendi hálózat modellezhető.

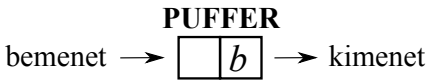
2.3.2. Boole változók

Egy Boole típusú változó mindig a logikai igaz/hamis értékek egyikét veszi fel. Tehát úgy tekinthetjük, mint egy olyan rendszert, melynek két állapota van: az egyik az igaz, a másik pedig a hamis érték tárolását képviseli. Ennek megfelelően nevezzük a **b** Boole változót modellező $\mathcal{B}$ átmeneti rendszer állapotait *true*-nak és *false*-nak, mint ahogy az a 2.6 ábrán is látható.

A változók értékét értékadásokkal módosíthatjuk. Értelemszerűen a **b:=true** utasítás a **b** változó értékét igazra, a **b:=false** utasítás pedig hamisra állítja. Így az alábbi átmeneteket kapjuk:

$$t_1 : true \mapsto \textbf{b:=true} \to true,$$
$$t_2 : true \mapsto \textbf{b:=false} \to false,$$
$$t_3 : false \mapsto \textbf{b:=true} \to true,$$
$$t_4 : false \mapsto \textbf{b:=false} \to false.$$

Nyilvánvaló, hogy ezen felül még olvasni vagy tesztelni is szeretnénk a változó értékét. Az ezekhez a műveletekhez tartozó átmenetek a változó állapotát nem fogják megváltoztatni.



2.7. ábra. Egy két betű tárolására alkalmas puffer

Ezért a hozzájuk tartozó átmeneteknek ugyanaz lesz a kezdő- és végpontjuk. A változó olvasásakor kétféle esemény következhet be. Az egyik, amikor a változó *true* állapotban van és az igaz értéket kapjuk olvasáskor; a másik, amikor a változó *false* állapotban található és ekkor nyilván hamis értéket ad vissza. Jelöljük rendre ezt a két eseményt a **true!** és **false!** címkével. Ezeket tehát úgy értelmezhetjük, hogy **true!** bekövetkezésekor az igaz, míg **false!** bekövetkezésekor a hamis értéket olvastuk a változóból. Ezeket az eseményeket az alábbi átmenetek jelenítik meg:

$$t_5 : true \mapsto \textbf{true!} \rightarrow true,$$
$$t_6 : false \mapsto \textbf{false!} \rightarrow false,$$

Végül, mivel a később bevezetésre kerülő szinkronizált szorzat definíciójához szükségünk lesz rá, vezessük be az **e**-vel címkézett *üres eseményt*. Az üres esemény semmit nem csinál, és nem változtathatja a rendszer állapotát sem, minden állapotból önmagába vezet. Mivel most két állapot van, két átmenetet kell felvennünk:

$$t_7 : true \mapsto \mathbf{e} \rightarrow true,$$
$$t_8 : false \mapsto \mathbf{e} \rightarrow false,$$

A **b** Boole változót modellező teljes átmeneti rendszer a 2.6 ábrán látható.

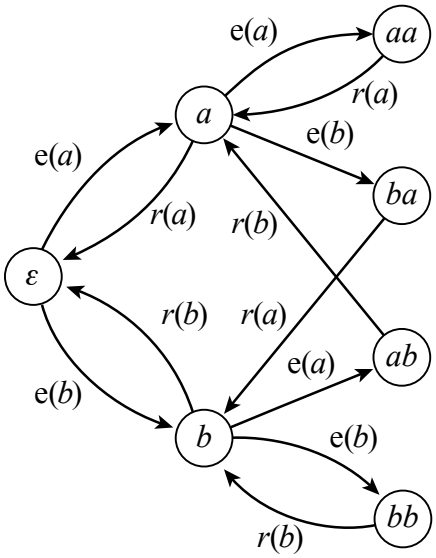
Már csak a kezdőállapot meghatározása maradt hátra. Ezt megtehetjük egy *kezdő* paraméter bevezetésével, mely a változó kezdeti értékét jelöli ki. Ha például azt tételezzük fel, hogy kezdetben **b** értéke hamis, akkor ezt az $S_{kezdő} = \{false\}$ választással tehetjük meg.

2.3.3. Korlátos pufferek

Tekintsünk az egyszerűség kedvéért egy olyan puffert, mely maximum két szimbólum tárolására alkalmas, melyek az 'a' illetve 'b' betűk közül kerülnek ki. Egy ilyen puffer működését mutatja a 2.7 ábra. A puffernek az alábbi hétféle állapota lehet. Zárójelben adjuk meg az állapotok nevét, melyet majd a pufferhez rendelt átmeneti rendszerben használunk.

- A puffer lehet üres (ε);
- tartalmazhat egyetlen 'a' (a), illetve 'b' betűt (b);
- egy korábban érkezett 'a' betű után tárolhat még egy 'a' (aa), illetve 'b' betűt (ba); végül,
- egy korábban érkezett 'b' betű után raktározhat még egy 'a' (ab), illetve 'b' betűt (bb).

Mint láthatjuk, az állapotok jelölésénél azzal a megállapodással éltünk, hogy a korábban érkezett betűk az állapotot reprezentáló szó jobb oldalára, azaz a szó végére, míg a később



2.8. ábra. A puffer címkézett átmeneti rendszer modellje,
e(x) *az* **enter(x)**, **r(x)** *a* **remove(x)** *címke rövidítése*

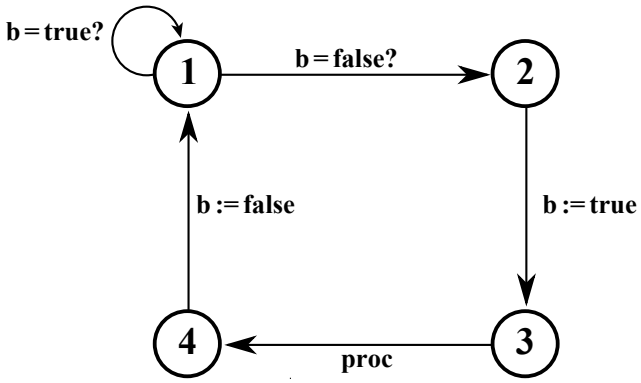
érkezett betűk a szó bal oldalára, azaz a szó elejére kerültek. Mindez csupán megállapodás, jelölhetnénk a puffer állapotait a szavak megfordításaival is. Ha a puffer üres, akkor ezt az állapotot ε, jelöli, amit az üres szóra is szoktunk alkalmazni.

A pufferen a következő műveleteket végezhetjük:

- Amennyiben a puffer nincs tele, egy újabb betűt helyezhetünk bele. Jelölje ennek a címkéjét **enter(**x**)**, ahol $x \in \{a, b\}$ a pufferbe írt betű.
- Amennyiben a puffer nem üres, kivehetünk belőle egy betűt, de mindig csak a legrégebben behelyezettet. Ennek címkéje legyen **remove(**x**)**. Természetesen itt $x \in \{a, b\}$ a pufferből eltávolított betű.

Ezek alapján az alábbi átmeneteket kapjuk:

$$\begin{aligned}
\varepsilon &\mapsto \textbf{enter}(\mathrm{a}) \to a,\\
\varepsilon &\mapsto \textbf{enter}(\mathrm{b}) \to b,\\
a &\mapsto \textbf{enter}(\mathrm{a}) \to aa,\\
a &\mapsto \textbf{enter}(\mathrm{b}) \to ba,\\
b &\mapsto \textbf{enter}(\mathrm{a}) \to ab,\\
b &\mapsto \textbf{enter}(\mathrm{b}) \to bb,\\
a &\mapsto \textbf{remove}(\mathrm{a}) \to \varepsilon,\\
b &\mapsto \textbf{remove}(\mathrm{b}) \to \varepsilon,\\
aa &\mapsto \textbf{remove}(\mathrm{a}) \to a,\\
ab &\mapsto \textbf{remove}(\mathrm{b}) \to a,\\
ba &\mapsto \textbf{remove}(\mathrm{a}) \to b,\\
bb &\mapsto \textbf{remove}(\mathrm{b}) \to b.
\end{aligned}$$



2.9. ábra. A programból származó $\mathcal{P}$ címkézett átmeneti rendszer

Az így kapott átmeneti rendszert láthatjuk a 2.8 ábrán. Mint korábban, minden állapotból önmagába vezető, üres, **e**-vel címkézett átmeneteket is felvethetnénk. Példánkat könnyen általánosíthatnánk tetszőleges $n \geq 1$ méretű és $k \geq 1$ betű tárolására alkalmas puffer modellezésére. Ekkor az átmeneti rendszernek $\frac{k^{n+1}-1}{k-1}$ állapota lenne.

2.3.4. Szekvenciális programok

Tekintsük a következő szekvenciális programrészletet:

```
while true do
   1: if not b then
      begin
         2: b:=true;
         3: proc;
         4: b:=false;
      end
```

A program végtelen ciklusban működik. Az **1.** utasítás teszteli, hogy a **b** Boole változó igaz-e. Amennyiben **b** igaz, nem történik semmi, amennyiben **b** hamis, a **2-4.** utasítások kerülnek végrehajtásra. Ezután mindkét esetben az **1.** utasítással újra kezdődik a ciklus.

A program vezérlési szerkezetét a 2.9 ábrán látható $\mathcal{P}$ átmeneti rendszerrel reprezentálhatjuk. $\mathcal{P}$-nek négy állapota van: **1, 2, 3, 4**. Ezeket tekinthetjük az utasításszámláló aktuális értékének, mely mindig azt mutatja, hogy melyik utasításnál áll a programszámláló. Kétfajta átmenet van: egyrészt az értékadó utasítások (**b:=true** és **b:=false**), illetve a **3.** utasításban szereplő **proc** eljáráshívás, másrészt a **b** változó lekérdezését végző **b=true?** és **b=false?** tesztelések. Így a rendszer átmenetei:

$$t_1 : 1 \mapsto \textbf{b:=true?} \to 1,$$
$$t_2 : 1 \mapsto \textbf{b:=false?} \to 2,$$
$$t_3 : 2 \mapsto \textbf{b:=true} \to 3,$$
$$t_4 : 3 \mapsto \textbf{proc} \to 4,$$

$$t_5 \ : 4 \mapsto \textbf{b:=false} \to 1.$$

Nyilvánvalóan a rendszerben a kezdőállapot az **1** állapot.

2.3.5. A Peterson algoritmus

Gray L. Peterson 1981-ben publikált kölcsönös kizárást megvalósító algoritmusa jól ismert a konkurens programozás területén. Feladata, hogy két egyszerre futó processzus (más szóval folyamat) biztonságosan férhessen hozzá valamely közös erőforráshoz (pl. fájlhoz, vagy memóriacímhez), melyet egyszerre csak egyikük birtokolhat. Például joggal várjuk el, hogy, miközben az egyik processzus az említett erőforráshoz hozzáfér, addig a másik processzus azt ne változtathassa meg. Ennek szokásos módszere az, hogy a közös erőforrást kezelő kódrészleteket ún. *kritikus szekciókba* foglaljuk, a többi kódrészlet pedig *nemkritikus szekciókba* kerül. A kölcsönös kizárást megvalósító algoritmusokkal azt kívánjuk elérni, hogy egy időben legfeljebb egy processzus tartózkodhasson a kritikus szekciójában.

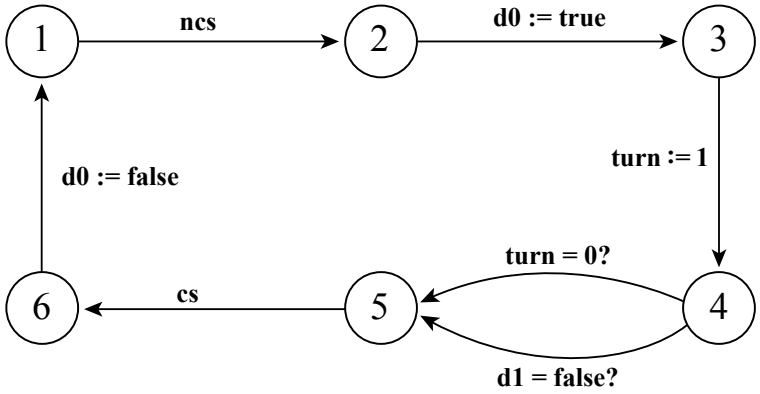
A Peterson algoritmus ezt a célt három globális változó segítségével valósítja meg. Jelöljük a két processzust P_0-lal és P_1-gyel. A **d0**, illetve **d1** Boole változó azt fogja jelezni, hogy a P_0, illetve a P_1 processzus szeretne-e a kritikus szekciójába lépni. Egész pontosan, mielőtt a P_0 processzus a kritikus szekciójába lépne, köteles a **d0** változót igazra állítani, majd miután a kritikus szekciót elhagyta, köteles **d0**-t hamisra visszaállítani. Hasonlóan állítja P_1 a **d1** változót. A harmadik, **turn** változó annak eldöntésére szolgál, melyik processzus kapja meg a kritikus szekcióhoz való hozzáférés jogát abban az esetben, ha **d0** és **d1** értéke is igaz. Ekkor az a processzus kap elsőbbséget, amelynek indexét a **turn** változó tartalmazza. Miután valamelyik processzus a kritikus szekcióba lépés igényét bejelentette, köteles ezt a változót a másik processzus indexére állítani. Ez biztosítja azt, hogy a processzusok fair módon férjenek hozzá a kritikus szekciójukhoz.

Végül, a **wait_until**(*feltétel*) utasítás tevékeny várakozást jelent mindaddig, míg a feltétel nem teljesül. Amennyiben a feltétel teljesül, a vezérlés tovább lép a következő utasításra, amennyiben nem, a következő végrehajtandó lépés újra a feltétel tesztelését jelenti. Ez alapján P_0 az alábbi kódot futtatja:

```
while true do
begin
   1: {non-critical section}
   2: d0:=true;
   3: turn:=1;
   4: wait_until(d1=false or turn=0);
   5: {critical section}
   6: d0:=false;
end
```

P_1 kódja szimmetrikus, csupán a **0**-kat és **1**-eket kell felcserélnünk P_0 kódjában:

```
while true do
begin
   1: {non-critical section}
```



2.10. ábra. A Petterson algoritmus P_0 processzusát modellező címkézett átmeneti rendszer

```
   2: d1:=true;
   3: turn:=0;
   4: wait_until(d0=false or turn=1);
   5: {critical section}
   6: d1:=false;
end
```

Példánkban a P_i processzust a $\mathcal{P}_i$ átmeneti rendszer fogja modellezni. Ezekben az **ncs**-sel címkézett átmenet a nem-kritikus szekció, **cs** pedig a kritikus szekció végrehajtását fogja jelenteni. A változók értékadását a szokásos módon **d0 :=true**, **d0 :=false**, **turn :=0** és **turn :=1** fogja megvalósítani, míg lekérdezésüket a **d0=true ?**, **d0=false ?**, **turn=0 ?** és **turn=1 ?** fogja reprezentálni. Hasonlóan járunk el a **d1** változó esetében.

Így $\mathcal{P}_0$ esetében az alábbi átmeneteket kapjuk:

$$
\begin{aligned}
t_1 &: 1 \mapsto \textbf{ncs} \to 2,\\
t_2 &: 2 \mapsto \textbf{d0:=true} \to 3,\\
t_3 &: 3 \mapsto \textbf{turn:=1} \to 4,\\
t_4 &: 4 \mapsto \textbf{d1=false ?} \to 5,\\
t_5 &: 4 \mapsto \textbf{turn=0 ?} \to 5,\\
t_6 &: 5 \mapsto \textbf{cs} \to 6,\\
t_7 &: 6 \mapsto \textbf{d0:=false} \to 1.
\end{aligned}
$$

Az így kapott $\mathcal{P}_0$ címkézett átmeneti rendszert, mely a P_0 processzust modellezi, a 2.10 ábrán láthatjuk.

Természetesen a $\mathcal{P}_1$ átmeneti rendszer a **0** és **1** indexek és értékek felcserélésével adódik:

$$
\begin{aligned}
t'_1 &: 1 \mapsto \textbf{ncs} \to 2,\\
t'_2 &: 2 \mapsto \textbf{d1:=true} \to 3,\\
t'_3 &: 3 \mapsto \textbf{turn:=0} \to 4,\\
t'_4 &: 4 \mapsto \textbf{d0=false ?} \to 5,
\end{aligned}
$$

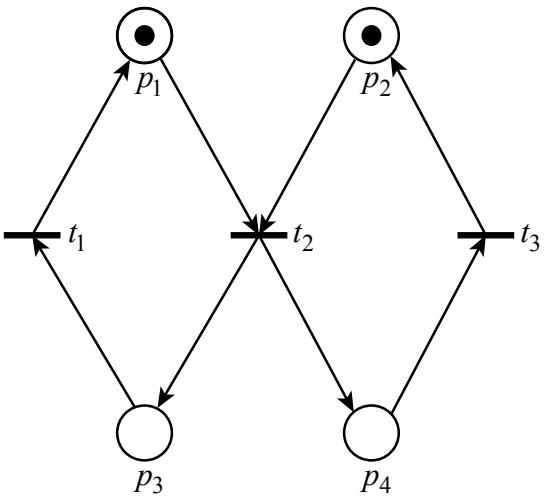
$$t'_5 : 4 \mapsto \textbf{turn=1?} \to 5,$$
$$t'_6 : 5 \mapsto \textbf{cs} \to 6,$$
$$t'_7 : 6 \mapsto \textbf{d1:=false} \to 1.$$

Az algoritmusban használt három változót, **d0**-t, **d1**-et és **turn**-t, a 2.3.2 fejezetben bemutatott, illetve **turn** esetén egy ahhoz nagyon hasonló átmeneti rendszerrel modellezhetjük.
Animált ábra. Az 1. animált ábra a Peterson algoritmus működését mutatja be a $\mathcal{P}_0$ és $\mathcal{P}_1$ átmeneti rendszerek segítségével.

2.4. Petri hálókból származtatott átmeneti rendszerek

A *Petri hálók* az egyik jól ismert és széles körben használt formalizmus konkurens rendszerek modellezésére. Ebben a fejezetben anélkül, hogy a részletekbe bocsátkoznánk, megmutatjuk, miért tekinthetjük a Petri hálókat is átmeneti rendszereknek.
Definíció. *Petri háló* alatt egy $(P, T, Pre, Post)$ négyest értünk, ahol

- P a *helyek* véges halmaza,
- T a *tranzíciók* (átmenetek) véges halmaza,
- Pre és $Post$ rendre a tranzíciók *őseit*, illetve *utódait* megadó: $T \to \mathcal{P}(P)$ függvények. (Itt $\mathcal{P}(P)$ a P halmaz hatványhalmazát, vagyis részhalmazainak a halmazát jelöli.)



2.11. ábra. Példa Petri hálóra

A Petri hálók szokásos grafikus ábrázolását mutatja a 2.11 ábra. A helyeket körök, a tranzíciókat rövid vastag vonalak, a Pre és $Post$ függvényeket pedig nyilak jelölik. Amennyiben egy p hely egy t tranzíció őse, azaz $p \in Pre(t)$, akkor ezt egy p-ből t be mutató nyíllal jelöljük. Amennyiben a t tranzíciónak utódja a p' hely, azaz $p' \in Post(t)$, ezt egy t-ből p'-be mutató nyíl ábrázolja.

Egy $(P, T, Pre, Post)$ Petri háló *token eloszlásán* egy $m : P \to \mathbb{N}$ függvényt értünk. Azt mondjuk, hogy $m(p) = k$ esetén az m token eloszlásban a p helyen k darab *token* szerepel. A grafikus ábrázolásban a helyeken található tokeneket tömör pontokkal (•) szemléltetjük.

Azt mondjuk, hogy egy m token eloszlásban egy t tranzíció *tüzelhető*, ha t minden ősén van legalább egy token, azaz

$$\forall p \in Pre(t) : m(p) > 0$$

teljesül. Amennyiben a t tranzíció tüzelhető m-ben, t *tüzelése* azt az m' új token eloszlást eredményezi, melyre

$$\forall p \in P\text{-re:} \quad m'(p) = m(p) - pre(p,t) + post(t,p), \text{ ahol}$$

$$pre(p,t) = \begin{cases} 1, & \text{ha } p \in Pre(t), \\ 0, & \text{különben;} \end{cases} \qquad post(t,p) = \begin{cases} 1, & \text{ha } p \in Post(t), \\ 0, & \text{különben.} \end{cases}$$

Például a 2.11 ábrán a t_2 tranzíció tüzelhető, és t_2 tüzelése azt az új token eloszlást eredményezi, melyben p_1-ről és p_2-ről lekerül egy-egy token, p_3-on és p_4-en pedig megjelenik egy-egy token.

Az mondjuk, hogy egy m token eloszlás *elérhető* a kezdő m_0 token eloszlásból, ha megadható olyan $m_0, m_1, \ldots, m_n = m$ sorozat, melyben minden m_{i+1} a megelőző m_i-ből valamely tranzíció tüzelésével kapható.

Ez alapján könnyen látható, hogy minden m_0 kezdő token eloszlással rendelkező Petri hálóhoz hozzárendelhetünk egy címkézett átmeneti rendszert a következő módon. Az állapotok legyenek az m_0-ból elérhető token eloszlások, az átmenetek pedig a tranzíciók tüzelésének feleljenek meg. Precízen fogalmazva, pontosan akkor vegyük fel az $m \mapsto t \to m'$ átmenetet, ha m-ben a t tranzíció tüzelhető, és tüzelése az m' token eloszlást eredményezi. Természetesen így csak akkor kapunk véges átmeneti rendszert, ha az m_0-ból elérhető token eloszlások száma véges. Ez biztosan teljesül, ha a Petri hálóban minden helyen korlátozzuk az ott elhelyezhető tokenek számát. Ez a tüzelési szabály módosulását is maga után vonja. Az ilyen Petri hálókat a szakirodalomban véges kapacitású Petri hálóknak nevezik.

Példa. Határozzuk meg a 2.11 ábrán található Petri hálóhoz társított címkézett átmeneti rendszert, ha az m_0 kezdeti token eloszlás

$$m_0(p_1) = 1, \quad m_0(p_2) = 1, \quad m_0(p_3) = 0, \quad m_0(p_4) = 0.$$

Közvetlen számolással adódik, hogy m_0-ból még az alábbiak az elérhető token eloszlások:

$$\begin{array}{llll} m_1(p_1) = 0, & m_1(p_2) = 0, & m_1(p_3) = 1, & m_1(p_4) = 1; \\ m_2(p_1) = 1, & m_2(p_2) = 0, & m_2(p_3) = 0, & m_2(p_4) = 1; \\ m_3(p_1) = 0, & m_3(p_2) = 1, & m_3(p_3) = 1, & m_3(p_4) = 0. \end{array}$$

Így a Petri háló t_1, t_2 és t_3 tranzícióinak tüzelése szerint az alábbi átmeneteket kapjuk:

$$m_0 \mapsto t_2 \to m_1,$$
$$m_1 \mapsto t_1 \to m_2,$$

$$m_1 \mapsto t_3 \to m_3,$$
$$m_2 \mapsto t_3 \to m_0,$$
$$m_3 \mapsto t_1 \to m_0.$$

Megjegyzés. Petri hálók esetében a t' és t'' tranzíciókat *függetlennek* nevezzük, ha

$$Pre(t') \cap \big(Pre(t'') \cup Post(t'')\big) = \emptyset \text{ és}$$
$$Pre(t'') \cap \big(Pre(t') \cup Post(t')\big) = \emptyset$$

teljesül. Ekkor, mivel t' és t'' ősei és utódai egymástól diszjunktak, az egyikük tüzelése a másik tüzelhetőségét nem befolyásolja. Így, ha mondjuk m-ben mind t', mind t'' tüzelhető, akkor végrehajtásuk sorrendje a végső token eloszlás szempontjából mindegy. Azaz, ha t' és t'' független tranzíciók, és a Petri hálóból származtatott átmeneti rendszerben $m \mapsto t' \to m'$, és $m \mapsto t'' \to m''$, valamint $m' \mapsto t'' \to m_1$ átmenet, akkor szükségképpen $m'' \mapsto t' \to m_1$ is teljesül. Mivel a t' és t'' közvetlenül egymás után tetszőleges sorrendben tüzelve m-ből az m_1-be vezet, úgy is gondolhatjuk, hogy t' és t'' egyszerre tüzelhetők, melynek az $m \mapsto t', t'' \to m_1$ átmenetet feleltethetjük meg. Ebben az esetben a Petri hálók T tranzícióhalmaza helyett annak hatványhalmazát $\mathcal{P}(T)$-t használjuk az átmeneti rendszer címkehalmazaként.

2.5. Átmeneti rendszerek szabad szorzata

Mint az eddigiekben láthattuk, az átmeneti rendszerek segítségével le tudjuk írni a modellezendő rendszerek komponenseit, legyenek azok akár önálló processzusok, akár más komponensek, mint például pufferek vagy közös változók. Ugyanakkor nyilvánvaló, hogy ahhoz, hogy a rendszer egészének viselkedését modellezzük, szükséges még a komponensek közötti kölcsönhatások, interakciók megadása is. Erre a célra a *szinkronizált szorzat* szolgál, mely segítségével a komponenseket leíró átmeneti rendszerekből az egész rendszerre vonatkozó globális átmeneti rendszer származtatható. A szinkronizált szorzat segítségével fogjuk a komponensek között megengedett és nem megengedett interakciókat definiálni, de mielőtt ezt megtennénk, az átmeneti rendszerek *szabad szorzatát* definiáljuk, melyben a komponensek közötti interakciót nem határozzuk meg. A szabad szorzatra ezért úgy tekinthetünk, mintha a rendszer komponensei egymástól teljesen függetlenül működnének, és köztük semmilyen interakció nem lenne.

Definíció. Legyenek $\mathcal{A}_i = (S_i, T_i, \alpha_i, \beta_i)$, $i = 1,2,\ldots,n$, átmeneti rendszerek. Az $\mathcal{A}_i$-k *szabad szorzata* alatt azt az $\mathcal{A}_1 \times \mathcal{A}_2 \times \ldots \times \mathcal{A}_n = (S, T, \alpha, \beta)$ átmeneti rendszert értjük, melyre

$$S = S_1 \times S_2 \times \ldots \times S_n,$$
$$T = T_1 \times T_2 \times \ldots \times T_n,$$
$$\alpha(t_1, \ldots, t_n) = (\alpha_1(t_1), \ldots, \alpha_n(t_n)),$$
$$\beta(t_1, \ldots, t_n) = (\beta_1(t_1), \ldots, \beta_n(t_n)).$$

Továbbá, amennyiben minden i-re, $\mathcal{A}_i$ valamely A_i ábécé feletti címkézett átmeneti rendszer, rendre $\lambda_i : T_i \to A_i$ címkefüggvényekkel, akkor a fentieken túl $\mathcal{A}_1 \times \ldots \times \mathcal{A}_n$ az $A_1 \times \ldots \times A_n$

ábécé feletti címkézett átmeneti rendszer, melynek $\lambda : T \to A_1 \times \ldots \times A_n$ címkefüggvényére

$$\lambda(t_1, \ldots, t_n) = \langle \lambda_1(t_1), \ldots, \lambda_n(t_n) \rangle$$

teljesül.

Megjegyzés. A $\langle \lambda_1(t_1), \ldots, \lambda_n(t_n) \rangle$ jelölés ugyanúgy az $A_1 \times \ldots \times A_n$ Descartes-szorzat egy elemét jelöli, mint az eddig használt $(\lambda_1(t_1), \ldots, \lambda_n(t_n))$ írásmód. Címkehalmazok esetében azért használjuk a $\langle$ és $\rangle$ zárójelpárt a kerek zárójelek helyett, hogy a később bevezetésre kerülő szinkronizációs vektorok olvashatóságát megkönnyítsük.

Szabad szorzat esetén az $\mathcal{A}$ rendszer $(s_1, \ldots, s_n)$ globális állapota azt fejezi ki, hogy az $\mathcal{A}_i$ komponens az s_i állapotban van, ahol $i = 1, \ldots, n$. Ekkor minden $\mathcal{A}_i$ egymástól függetlenül hajthatja végre az s_i állapotban a t_i átmenetet, mely, mondjuk, az s_i' állapotba viszi az $\mathcal{A}_i$ komponenst. Ezek az átmenetek együttesen az $\mathcal{A}$ rendszerben a $t = (t_1, \ldots, t_n)$ globális átmenetet eredményezik, mely értelemszerűen az $s = (s_1, \ldots, s_n)$ állapotból az $s' = (s_1', \ldots, s_n')$ állapotba viszi $\mathcal{A}$-t. A t átmenetet *globális átmenetnek* nevezzük. Címkézett átmeneti rendszerek esetén a t globális átmenet $\lambda(t)$ címkéjére, mint *globális akcióra* fogunk hivatkozni.

2.6. Szinkron és aszinkron rendszerek

Mint láthattuk, a szabad szorzat azt feltételezi, hogy a benne szereplő komponensek egymástól függetlenül, de egy időben hajtják végre átmeneteiket. Azt is képzelhetjük, hogy van egy globális óra, melynek minden ütésére minden komponens egy-egy átmenet végrehajtásába kezd, és biztos, hogy az átmenet végrehajtása befejeződik még a következő óraütés előtt. Az mondjuk, hogy az ilyen rendszerek *szinkron* módban működnek.

Bizonyos esetekben viszont azt szeretnénk, hogy ne kelljen mindig minden komponensnek állapotot váltania, hanem egy-egy komponens tetszőleges ideig maradhasson egy állapotban, míg a rendszer más komponensei tevékenykednek. Ha a komponensek egymáshoz viszonyított sebességéről semmilyen feltételezést nem teszünk, vagy, másként fogalmazva, megengedjük, hogy az egyik komponens egy lépésének végrehajtása előtt egy másik komponens tetszőlegesen sok lépést megtegyen, akkor *aszinkron* működésről beszélünk.

Fontos észrevétel, hogy az aszinkron működés modellezhető szinkron módban is a következő egyszerű technikával. Ha egy aszinkron rendszert kívánunk szinkron módban modellezni, akkor minden komponens minden állapotából önmagába vegyünk fel egy speciális **e** címkével ellátott *üres eseményt* vagy *üres akciót*. Mint korábban már utaltunk rá, ennek végrehajtása az adott állapotban való tétlen várakozást modellezi. Igazából lényegtelen, hogy ezt a várakozást kifejező átmenetet milyen speciális címkével jelöljük, hiszen az átmenetekhez rendelt címkéknek mindaddig nincs jelentőségük, míg nem kívánunk rájuk a specifikációt leíró logikai formulákban hivatkozni.

Például, amennyiben a Peterson algoritmus két processzusát egyetlen processzor hajtja végre, akkor nyilvánvaló, hogy minden pillanatban csak az egyik processzus lehet aktív, a másiknak inaktívnak kell maradnia. Ezért a processzusokhoz rendelt átmeneti rendszerek minden állapotában meg kell engednünk az **e** átmenetet is.

Közvetlenül belátható, hogy amennyiben kikötjük, hogy az így kibővített átmeneti rendszerben mindig pontosan egy komponens hajt végre nem **e** átmenetet, a többi pedig **e** átmenet

végrehajtására kényszerül, akkor a kibővített rendszer szinkron működésével éppen az eredeti rendszer aszinkron működését modellezzük.

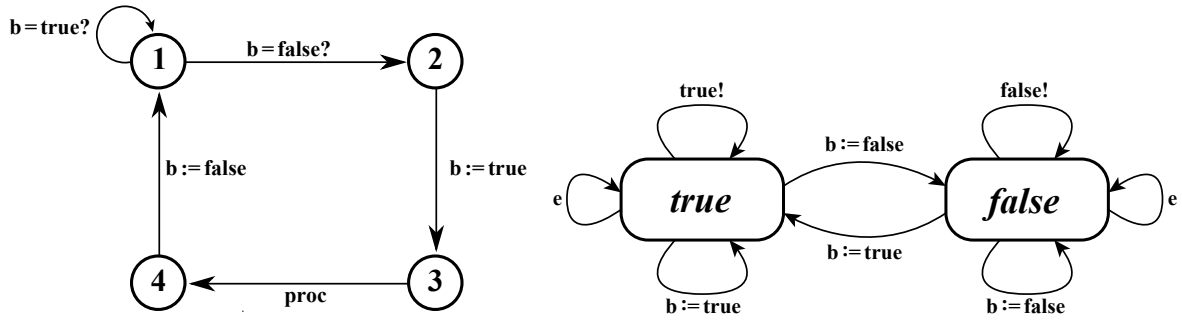
Ezért, a továbbiakban csak a szinkron működést tárgyaljuk, hiszen ezzel a várakozó átmenetek révén az aszinkron esetet is vizsgálhatjuk. Sőt kevert, szinkron-aszinkron rendszereket is modellezhetünk, melyekben nem mindig, hanem csak bizonyos állapotokban engedélyezünk tétlen várakozást. A szinkron interpretáció további előnyeit fogjuk látni a következő fejezetben, amikor a különböző komponensek közötti interakciókat vizsgáljuk.

2.7. Átmeneti rendszerek szinkronizált szorzata

Nyilvánvaló, hogy mihelyst a rendszer komponensek egymással kölcsönhatásba, interakcióba lépnek, a szabad szorzat bizonyos globális átmeneteit nem kívánjuk megengedni, mert az interakció szinkronizációs és kommunikációs megszorításokkal jár. Például elvárjuk, hogy ha az egyik processzus megváltoztatja egy változó értékét, akkor a változót reprezentáló komponensnek ezzel együtt végre kelljen hajtania egy olyan átmenetet, mely a változó értékének megváltozását fejezi ki. Az interakciók megszorításait is figyelembe vevő átmeneti rendszer mindig a komponensek szabad szorzatának részrendszere lesz. A kommunikációs és szinkronizációs megszorításokat a *szinkronizált szorzat* segítségével fogjuk kifejezni, de mielőtt ezt megtennénk, lássunk egy példát.

2.7.1. Szinkronizációs vektorok

Emlékeztetőül: a 2.3.4 fejezetben látott szekvenciális programhoz tartozó, valamint a programban szereplő (a 2.3.2 fejezetben bemutatott) **b** Boole változót reprezentáló címkézett átmeneti rendszereket a 2.12 ábrán láthatjuk. Korábban ezt a két átmeneti rendszert $\mathcal{P}$-vel, illetve $\mathcal{B}$-vel jelöltük.



2.12. ábra. $\mathcal{P}$ és $\mathcal{B}$, a szekvenciális program és a benne szereplő **b** *változó átmeneti rendszer modellje*

Mivel a $\mathcal{P}$-nek 4 állapota és 5 átmenete, $\mathcal{B}$-nek pedig 2 állapota és 8 átmenete van, a definíció értelmében $\mathcal{P}$ és $\mathcal{B}$ szabad szorzatában 8 állapotnak és 40 átmenetnek kell szerepelnie. Például a **(4, true)** globális állapotban végrehajtható a $\langle$**b** := **false**, **b** := **true**$\rangle$ globális akció, mely az **(1, true)** állapotba vezet.

Ugyanakkor nyilvánvaló, hogy a fenti $\langle$**b** := **false**, **b** := **true**$\rangle$ globális akciónak a valóságban semmi értelme, ha a $\mathcal{P}$ program a **b** := **false** értékadást hajtja végre, akkor ezzel egyidőben

a $\mathcal{B}$ komponensnek, mely a **b** változót reprezentálja, szintén egy **b** := **false**-vel címkézett átmenetet kell végrehajtania. Hasonló a helyzet fordítva: ha példánkban a **b** változóra csak a $\mathcal{P}$ program van hatással, akkor a $\mathcal{B}$ komponensben egy **b** := **false**-vel címkézett átmenet csak a $\mathcal{P}$ program egy **b** := **false** utasításából származhat. Ezért a szabad szorzat **b** := **false** címkét tartalmazó átmenetei közül $\langle$**b** := **false**, **b** := **false**$\rangle$ az egyetlen, amit megengedhetünk.

Hasonlók a megszorítások, ha a $\mathcal{P}$ program ezután a változó értékét teszteli, mégpedig a **b** = **true**? vagy a **b** = **false**? utasítással. Emlékeztetőül, a **b** = **false**? utasítás egyben azt is jelenti, hogy a program nem csak teszteli a **b** értékét, hanem azt is tudjuk, hogy az érték hamis (**false**) volt, és az utasítás után is az maradt. Ennek az a következménye, hogy $\mathcal{P}$ a 2-es állapotba, nem pedig az 1-es állapotba kerül, mint abban az esetben, ha a változó igaz (**true**) értékű, és ezért a teszt során a **b** = **true**? átmenet hajtódik végre. Ha tehát $\mathcal{P}$ a **b** = **false**? átmenetet végzi, akkor ezzel együtt $\mathcal{B}$-ben a **false**! átmenetnek kell lezajlani. Emlékeztetünk rá, hogy a 2.3.2 fejezetben a $\mathcal{B}$-ben a **false**! átmenet éppen azt jelentette, hogy kiolvastuk a változó értékét és azt hamisnak találtuk. Hasonlóan a $\mathcal{P}$ program **b** = **true**? átmenetéhez $\mathcal{B}$-ben a **true**! átmenetnek kell társulni.

Végül, ha $\mathcal{P}$ a **proc** átmenetet hajtja végre, akkor feltevésünk szerint ennek végrehajtásakor a processzus nincs kapcsolatban a **b** változóval. Ezt úgy fejezhetjük ki, hogy **proc** végrehajtásakor $\mathcal{B}$ nem csinál semmit, azaz az üres **e** átmenetet hajtja végre. Összefoglalva tehát a $\mathcal{P} \times \mathcal{B}$ szorzatban a megengedett globális átmenetek pontosan az alábbiak:

$$
\begin{array}{lcl}
\langle\ \textbf{b} := \textbf{true} & , & \textbf{b} := \textbf{true}\ \rangle, \\
\langle\ \textbf{b} := \textbf{false} & , & \textbf{b} := \textbf{false}\ \rangle, \\
\langle\ \textbf{b} = \textbf{true}? & , & \textbf{true}!\ \rangle, \\
\langle\ \textbf{b} = \textbf{false}? & , & \textbf{false}!\ \rangle, \\
\langle\ \textbf{proc} & , & \textbf{e}\ \rangle.
\end{array}
$$

A fenti felsorolás elemeit a továbbiakban *szinkronizációs vektoroknak* nevezzük.

2.7.2. A szinkronizált szorzat definíciója

Az előző példa jól mutatja, hogyan tudjuk a rész átmeneti rendszerek közötti szinkronizációs megszorításokat formalizálni. Ha az $\mathcal{A}_1, \ldots, \mathcal{A}_n$ részrendszerek rendre az A_1, …, A_n ábécé elemeivel címkézettek, akkor az $A_1 \times \ldots \times A_n$ Descartes-szorzat tetszőleges részhalmazát *szinkronizációs megszorításoknak*, e részhalmaz elemeit pedig *szinkronizációs vektoroknak* hívjuk. Az imént látott öt átmenetpár a $\mathcal{P} \times \mathcal{B}$ rendszer szinkronizációs vektorai. Nyilvánvaló, hogy a szinkronizációs vektorok így nem mások, mint a szabad szorzat bizonyos globális akciói. Amennyiben a szabad szorzatban csak a szinkronizációs vektorok által engedélyezett globális átmeneteket tartjuk meg, az adott szinkronizációs megszorításokhoz tartozó *szinkronizált szorzathoz* jutunk. A formális definíció az alábbi:

Definíció. Legyen $\mathcal{A}_i$ az A_i ábécé feletti címkézett átmeneti rendszer, minden $i = 1, \ldots, n$-re. Továbbá legyen $I \subseteq A_1 \times \ldots \times A_n$ tetszőleges szinkronizációs megszorítás, azaz szinkronizációs vektorok halmaza. Ekkor az $\mathcal{A}_i$-k, I *szerinti szinkronizált szorzatán* azt az $\langle \mathcal{A}_1, \ldots, A_n, I \rangle$ átmeneti rendszert értjük, mely az $\mathcal{A}_1 \times \ldots \times \mathcal{A}_n$ szabad szorzat azon rész átmeneti rendszere, mely csak azokat a $(t_1, \ldots, t_n)$ globális átmeneteket tartalmazza, melyek címkéjére $\langle \lambda_1(t_1), \ldots, \lambda_n(t_n) \rangle \in I$ teljesül.

Példa. Ha az előző példában felsorolt szinkronizációs vektorokkal képezzük $\mathcal{P}$ és $\mathcal{B}$ szinkronizált szorzatát, az alábbi átmeneti rendszert kapjuk:

$$\begin{array}{lrll}
(1, \textbf{true}) & \mapsto \langle\, \textbf{b} = \textbf{true}?, \textbf{true}!\,\rangle \rightarrow & (1, \textbf{true}), & (1)\\
(2, \textbf{true}) & \mapsto \langle\, \textbf{b} := \textbf{true}, \textbf{b} := \textbf{true}\,\rangle \rightarrow & (3, \textbf{true}), & (2)\\
(3, \textbf{true}) & \mapsto \langle\, \textbf{proc}, \textbf{e}\,\rangle \rightarrow & (4, \textbf{true}), & (3)\\
(4, \textbf{true}) & \mapsto \langle\, \textbf{b} := \textbf{false}, \textbf{b} := \textbf{false}\,\rangle \rightarrow & (1, \textbf{false}), & (4)\\
(1, \textbf{false}) & \mapsto \langle\, \textbf{b} = \textbf{false}?, \textbf{false}!\,\rangle \rightarrow & (2, \textbf{false}), & (5)\\
(2, \textbf{false}) & \mapsto \langle\, \textbf{b} := \textbf{true}, \textbf{b} := \textbf{true}\,\rangle \rightarrow & (3, \textbf{true}), & (6)\\
(3, \textbf{false}) & \mapsto \langle\, \textbf{proc}, \textbf{e}\,\rangle \rightarrow & (4, \textbf{false}), & (7)\\
(4, \textbf{false}) & \mapsto \langle\, \textbf{b} := \textbf{false}, \textbf{b} := \textbf{false}\,\rangle \rightarrow & (1, \textbf{false}). & (8)
\end{array}$$

Feltételezhetjük, hogy a rendszer az (1, **false**)-ból indul, azaz kezdetben az első program utasítást hajtjuk végre, és a **b** változó értéke pedig **false**. Ekkor az (1), (2), (7) és (8) átmenetek nem érhetők el, azaz sohasem kerülhetnek végrehajtásra, ezért elhagyhatók. Így egy olyan átmeneti rendszert kapunk, melynek 4 állapota egymást követve változik az alábbi átmenetek szerint:

$$\begin{array}{lrl}
(1, \textbf{false}) & \mapsto \langle\, \textbf{b} = \textbf{false}?, \textbf{false}!\,\rangle \rightarrow & (2, \textbf{false}),\\
(2, \textbf{false}) & \mapsto \langle\, \textbf{b} := \textbf{true}, \textbf{b} := \textbf{true}\,\rangle \rightarrow & (3, \textbf{true}),\\
(3, \textbf{true}) & \mapsto \langle\, \textbf{proc}, \textbf{e}\,\rangle \rightarrow & (4, \textbf{true}),\\
(4, \textbf{true}) & \mapsto \langle\, \textbf{b} := \textbf{false}, \textbf{b} := \textbf{false}\,\rangle \rightarrow & (1, \textbf{false}).
\end{array}$$

Ez megfelel annak, hogy, mivel egyetlen determinisztikus programról van szó, a végrehajtás során mindig egyértelmű mind a következő utasítás, mind a változó értéke.

Amennyiben viszont a szinkronizált szorzatban nem az (1, **false**), hanem az (1, **true**) globális állapot lenne a kezdőállapot, akkor csak ez az egyetlen állapot lenne elérhető, és a rendszert az egyetlen

$$(1, \textbf{true}) \mapsto \langle\, \textbf{b} = \textbf{true}?, \textbf{true}!\,\rangle \rightarrow (1, \textbf{true})$$

átmenet írná le.

2.8. Paraméterezett átmeneti rendszerek szinkronizált szorzata

2.8.1. Definíció

Tekintsünk k darab átmeneti rendszert, mégpedig legyen $\mathcal{A}_i = (S_i, T_i, \alpha_i, \beta_i, \lambda)$ az A_i ábécé feletti címkézett átmeneti rendszer, minden $i = 1, \ldots, k$ esetén. Legyen továbbá $I \subseteq A_1 \times \ldots \times \times A_k$ szinkronizációs megszorítások halmaza és jelölje $\mathcal{B} = (S, T, \alpha, \beta, \lambda)$ az A_i-k I szerinti szinkronizált szorzatát. Az előző fejezetben ismertetett definíció szerint $\mathcal{B}$ az I halmaz feletti címkézett átmeneti rendszer.

Ezen kívül tegyük fel, hogy minden $\mathcal{A}_i$ átmeneti rendszer paraméterezett átmeneti rendszer is, valamely $(\mathcal{X}_i, \mathcal{Y}_i)$ paraméterhalmazok felett, melyek nem feltétlenül azonosak minden i esetén. Ekkor $\mathcal{B}$-t $(\mathcal{X}, \mathcal{Y})$-paraméterezett átmeneti rendszernek is tekinthetjük, ahol $\mathcal{X}$, (illetve $\mathcal{Y}$) az $\mathcal{X}_i$ (illetve $\mathcal{Y}_i$) halmazok diszjunkt egyesítése, azaz

$$\mathcal{X} = \{(i, x) \mid 1 \leq i \leq k, x \in \mathcal{X}_i\}, \text{ és}$$

$$\mathcal{Y} = \{(i, y) \mid 1 \leq i \leq k, y \in \mathcal{Y}_i\}.$$

Továbbá,

$$S_{(i,x)} = \{(s_1, \ldots, s_i, \ldots s_k) \in S \mid s_i \in (S_i)_x\}, \text{ és}$$
$$T_{(i,y)} = \{(t_1, \ldots, t_i, \ldots t_k) \in T \mid t_i \in (T_i)_y\}$$

minden $i = 1, \ldots, k$, $x \in \mathcal{X}$ és $y \in \mathcal{Y}$ esetén.

Ez azt jelenti, hogy a szinkronizált szorzat megőrzi az összes részrendszer paraméterezését. Például azt, hogy az aktuális $s = (s_1, \ldots, s_k)$ globális állapotban az $\mathcal{A}_i$ részrendszer s_i állapota rendelkezik-e az $x \in \mathcal{X}_i$ paraméterrel az $s \in S_{(i,x)}$ feltétellel tudjuk kifejezni. Hasonló a helyzet átmenetparaméterek esetében. Így mindazok a tulajdonságok, melyek az $\mathcal{A}_i$ részrendszerekben kifejezhetők, felírhatók a szinkronizált szorzatban is.

2.8.2. Az alternáló bit protokoll

Az irodalomban az átmeneti rendszerek használatára az egyik klasszikus példa az *alternáló bit protokoll* (ABP, Alternating Bit Protocol), melynek már a definiálása is átmeneti rendszerek segítségével történt [6]. Az alábbiakban ezt ismertetjük.

A protokoll az adatkapcsolati rétegben (data link layer) dolgozik. Célja, hogy egy S (sender) adó egy R (receiver) vevőnek egy olyan csatornán keresztül is tudjon megbízhatóan üzeneteket továbbítani, melyben az üzenetek megsérülhetnek vagy elveszhetnek. A protokoll ezt úgy valósítja meg, hogy S minden üzenetéhez egy egybites sorszámot (sequence number), 0-t vagy 1-et csatol. Amennyiben az üzenet hibátlanul megérkezik R-hez, R egy nyugtát (acknowledgement, ACK) küld vissza S-nek a kapott üzenet sorszám bitjével együtt. Mivel a csatornában mind az üzenet, mind a nyugta megsérülhet vagy elveszhet, a protokoll szerint S mindaddig ismétli az adott üzenet és sorszám bit küldését, amíg egy várt vezérlő bitet tartalmazó nyugta nem érkezik hozzá. Ha ezt a nyugtát megkapta, a sorszám bitet átfordítja és annak megváltozott értékével küldi a következő üzenetet mindaddig, amíg a sorszám bitnek megfelelő nyugta nem érkezik hozzá. Az R vevő hasonlóan jár el. Egy üzenet sikeres vétele után mindaddig ismétli az üzenet nyugtázását (természetesen a vett üzenet sorszám bitjével), amíg egy ellentétes sorszámbittel ellátott hibátlan üzenet el nem érkezik hozzá. Ezután az új üzenetről kezd nyugtákat küldeni egészen a következő üzenet sikeres vételéig.

A könnyebb áttekinthetőség kedvéért, pontosabban, hogy a végső szinkronizált szorzat ne legyen túlságosan nagy, példánkban néhány egyszerűsítéssel élünk az eredeti változathoz képest. Ezek a következők:

- Az eredeti egyidőben kétirányú (full-duplex) összeköttetés helyett csak egyirányú csatornát modellezünk, így az egyik fél végig az adó, másik pedig a vevő szerepét tölti be.
- Feltesszük azt is, hogy az üzenettovábbítás azonnal megtörténik, szemben azzal a változattal, melyben az üzeneteket és nyugtákat egy pufferen keresztül továbbítjuk.
- Végül, az eredeti változat kétfajta hibát különböztetett meg: nem megfelelő ellenőrző bittel vett üzenetet, és egyéb átviteli hibát. Mivel a kétfajta hiba ugyanazt a viselkedést

idézi elő a vevő oldalon, mi nem fogjuk megkülönböztetni őket. Azaz feltesszük, hogy minden hibás üzenetküldés a nem megfelelő sorszám bit észlelésével érzékelhető.

Ezek alapján az adó akciói/eseményei az alábbiak:

em1 üzenet küldése 1-es bittel,
em0 üzenet küldése 0-ás bittel,
ra1 nyugta fogadása 1-es bittel,
ra0 nyugta fogadása 0-ás bittel.

Az **em** és **ra** címkék, az „emission of a message” (üzenet kibocsátása), valamint az „reception of an acknowledgement” (nyugta fogadása) kifejezések rövidítései.

A vevő processzus viselkedése hasonló:

rm1 üzenet fogadása 1-es bittel,
rm0 üzenet fogadása 0-ás bittel,
ea1 nyugta küldése 1-es bittel,
ea0 nyugta küldése 0-ás bittel.

Itt **rm** és **ea** a „reception of a message” (üzenet fogadása), valamint az „emission of an acknowledgement” (nyugta kibocsátása) kifejezések helyett állnak.

Vegyük észre, hogy az akciók listái nem tartalmazzák **e**-t, az üres eseményt. Ez azért van, mert feltesszük, hogy az üzenetek és nyugták küldése és fogadása egyszerre történik. Ezt azért tehetjük meg, mert a protokoll elemzése szempontjából csak az üzenetváltások lehetséges sorozatai érdekesek, így eltekinthetünk az üzenetváltások között eltelt időtartamok nyilvántartásától. Ekkor egyik kommunikáló fél sem lehet inaktív, minden pillanatban mindkét fél vagy adatküldést vagy fogadást végez, ezért a rendszert teljesen szinkron módon interpretálhatjuk.

Így az adót modellező átmeneti rendszer a következő:

$$
\begin{array}{lrlll}
t_1 & : & \mathbf{send}_0 & \mapsto \mathbf{em}0 \rightarrow & \mathbf{wait}_0, \\
t_2 & : & \mathbf{send}_1 & \mapsto \mathbf{em}1 \rightarrow & \mathbf{wait}_1, \\
t_3 & : & \mathbf{wait}_0 & \mapsto \mathbf{ra}0 \rightarrow & \mathbf{send}_1, \\
t_4 & : & \mathbf{wait}_0 & \mapsto \mathbf{ra}1 \rightarrow & \mathbf{resend}_0, \\
t_5 & : & \mathbf{wait}_1 & \mapsto \mathbf{ra}1 \rightarrow & \mathbf{send}_0, \\
t_6 & : & \mathbf{wait}_1 & \mapsto \mathbf{ra}0 \rightarrow & \mathbf{resend}_1, \\
t_7 & : & \mathbf{resend}_0 & \mapsto \mathbf{em}0 \rightarrow & \mathbf{wait}_0, \\
t_8 & : & \mathbf{resend}_1 & \mapsto \mathbf{em}1 \rightarrow & \mathbf{wait}_1.
\end{array}
$$

Egyetlen állapotparamétert definiálunk ($\mathcal{X}=\{initial\}$), mely a kezdőállapotokat tartalmazza:

$$S_{initial} = \{\mathbf{send}_0,\ \mathbf{send}_1\},$$

továbbá két átmenetparamétert ($\mathcal{Y} = \{emission, re-emission\}$), melyekre

$$
\begin{aligned}
T_{emission} &= \{t_1, t_2\}, \text{ az első alkalommal történő üzenetküldést,} \\
T_{re-emission} &= \{t_7, t_8\}, \text{ pedig az üzenetküldés megismétlését jelenti.}
\end{aligned}
$$

A vevő felet modellező átmeneti rendszer pedig a következő:

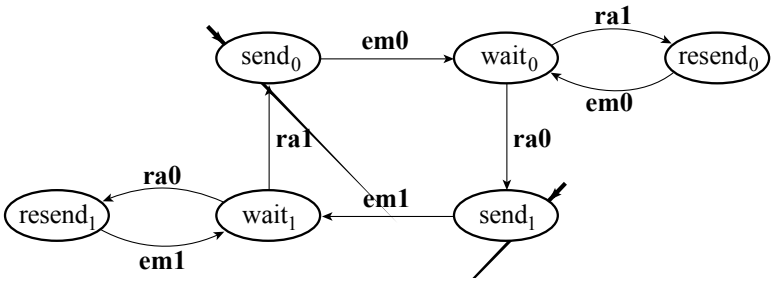
$$\begin{array}{rrcl} t_1' & : & \mathbf{wait}_0 & \mapsto \mathbf{rm}0 \to \quad \mathbf{send}_0, \\ t_2' & : & \mathbf{wait}_0 & \mapsto \mathbf{rm}1 \to \quad \mathbf{resend}_1, \\ t_3' & : & \mathbf{wait}_1 & \mapsto \mathbf{rm}1 \to \quad \mathbf{send}_1, \\ t_4' & : & \mathbf{wait}_1 & \mapsto \mathbf{rm}0 \to \quad \mathbf{resend}_0, \\ t_5' & : & \mathbf{send}_0 & \mapsto \mathbf{ea}0 \to \quad \mathbf{wait}_1, \\ t_6' & : & \mathbf{send}_1 & \mapsto \mathbf{ea}1 \to \quad \mathbf{wait}_0, \\ t_7' & : & \mathbf{resend}_0 & \mapsto \mathbf{ea}0 \to \quad \mathbf{wait}_1, \\ t_8' & : & \mathbf{resend}_1 & \mapsto \mathbf{ea}1 \to \quad \mathbf{wait}_0. \end{array}$$

Ebben az esetben az állapotparaméter ($\mathcal{X} = \{initial\}$) elemei a következők:

$$S_{initial} = \{\mathbf{wait}_0,\ \mathbf{wait}_1\}.$$

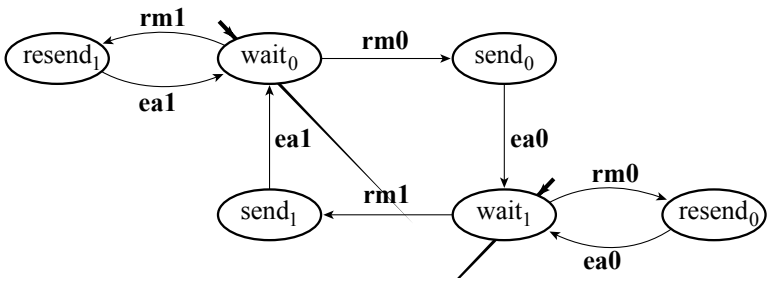
A két átmenetparaméter ($\mathcal{Y} = \{well\_received,\ ill\_received\}$) pedig az alábbi:

$$T_{well\_received} = \{t_1', t_3'\},\ \text{ami az üzenet sikeres fogadását jelzi,}$$
$$T_{ill\_received} = \{t_2', t_4'\},\ \text{ami a hibás üzenettovábbításra utal.}$$



2.13. ábra. Az adó átmeneti rendszer modellje

A két átmeneti rendszert a 2.13 és a 2.14 ábrán láthatjuk. Ez alapján könnyen végigkövethetjük a protokoll résztvevőinek működését.



2.14. ábra. A vevő átmeneti rendszer modellje

Nyilvánvaló, hogy a teljes rendszer az adó és a vevő átmeneti rendszerének szinkronizált szorzata lesz. A továbbiakban azt fogjuk bemutatni, hogy különböző szinkronizációs megszorítások választásával hogyan tudjuk a kommunikáció mindkét irányban vett hibátlan vagy tökéletlen voltát modellezni.

1. Ha mindkét irányban hibátlan a kommunikáció
Ha a csatorna mindkét irányban tökéletes, akkor mind az üzenetek, mind a nyugták hibátlanul, vagyis a vezérlő bit megváltozása nélkül érkeznek meg. Ebben az esetben adó minden **em0** üzenetküldésével szinkronban a vevő az **rm0** átmenetet hajtja végre, az adó **em1** üzenetküldéséhez pedig a vevő **rm1** átmenete társul. Hasonló a helyzet a nyugták továbbításánál, **ea0** az **ra0**-hoz, **ea1** pedig a **ra1**-hez társul. Így ennek a kommunikációs feltételnek az alábbi szinkronizációs vektorok felelnek meg:

$$\begin{aligned} I = \{ &\langle \mathbf{em}0 , \mathbf{rm}0 \rangle, \\ &\langle \mathbf{em}1 , \mathbf{rm}1 \rangle, \\ &\langle \mathbf{ra}0 , \mathbf{ea}0 \rangle, \\ &\langle \mathbf{ra}1 , \mathbf{ea}1 \rangle \}. \end{aligned}$$

2. Ha csak az vevőtől az adóig irányban hibátlan a kommunikáció
Ebben az esetben, amíg az adótól a vevőig ér az üzenet előfordulhat hiba, melyet az üzenet fogadásakor a megváltozott vezérlő bit jelez. Ezért a kommunikációs megszorítások előző listájához hozzá kell még adnunk az $\langle \mathbf{em}0, \mathbf{rm}1 \rangle$ és az $\langle \mathbf{em}1, \mathbf{rm}0 \rangle$ párokat, melyek a hibás üzenetküldést jelentik. Ezáltal az alábbi kommunikációs megszorításokat kapjuk:

$$\begin{aligned} I_e = \{ &\langle \mathbf{em}0 , \mathbf{rm}0 \rangle, \\ &\langle \mathbf{em}1 , \mathbf{rm}1 \rangle, \\ &\langle \mathbf{ra}0 , \mathbf{ea}0 \rangle, \\ &\langle \mathbf{ra}1 , \mathbf{ea}1 \rangle, \\ &\langle \mathbf{em}0 , \mathbf{rm}1 \rangle, \\ &\langle \mathbf{em}1 , \mathbf{rm}0 \rangle \}. \end{aligned}$$

3. Ha csak az adótól a vevőig irányban hibátlan a kommunikáció
Itt az előző esethez képest fordított a helyzet. Az adótól a vevőig hibátlan a kommunikáció, viszont az ellenkező irányban nem, vagyis a vevő által küldött nyugták vezérlő bitje változhat meg. Ezt úgy modellezhetjük, hogy az I halmazhoz az $\langle \mathbf{ra}0, \mathbf{ea}1 \rangle$ és az $\langle \mathbf{ra}1, \mathbf{ea}0 \rangle$ párokat adjuk hozzá. Ekkor a következő kommunikációs megszorításokhoz jutunk:

$$\begin{aligned} I_r = \{ &\langle \mathbf{em}0 , \mathbf{rm}0 \rangle, \\ &\langle \mathbf{em}1 , \mathbf{rm}1 \rangle, \\ &\langle \mathbf{ra}0 , \mathbf{ea}0 \rangle, \\ &\langle \mathbf{ra}1 , \mathbf{ea}1 \rangle, \\ &\langle \mathbf{ra}0 , \mathbf{ea}1 \rangle, \\ &\langle \mathbf{ra}1 , \mathbf{ea}0 \rangle \}. \end{aligned}$$

4. Ha mindkét irányban megbízhatatlan a kommunikáció
Ez az előző két eset egyesítése. Ha mind az üzenetek, mind a nyugták meghibásodhatnak,

ekkor az $I_{er} = I_e \cup I_r$ megszorítás vektorokkal tudjuk a rendszert modellezni:

$$\begin{aligned} I_e = \{ &\langle \mathbf{em}0 , \mathbf{rm}0 \rangle, \\ &\langle \mathbf{em}1 , \mathbf{rm}1 \rangle, \\ &\langle \ \mathbf{ra}0 \ , \ \mathbf{ea}0 \ \rangle, \\ &\langle \ \mathbf{ra}1 \ , \ \mathbf{ea}1 \ \rangle, \\ &\langle \mathbf{em}0 , \mathbf{rm}1 \rangle, \\ &\langle \mathbf{em}1 , \mathbf{rm}0 \rangle, \\ &\langle \ \mathbf{ra}0 \ , \ \mathbf{ea}1 \ \rangle, \\ &\langle \ \mathbf{ra}1 \ , \ \mathbf{ea}0 \ \rangle \}. \end{aligned}$$

A könnyebb áttekinthetőség végett érdemes a szinkronizált szorzat bizonyos átmenet paramétereit külön elneveznünk. Legyen

- **E**=$(1, emission)$, azaz azok az átmenetek tartozzanak ide, melyben az első (vagyis az adó) komponens egy $emission$ átmenetparaméterrel rendelkező átmenetet hajt végre, azaz éppen üzenetet küld;
- **R**=$(1, re-emission)$, azaz azok az átmenetek tartozzanak ide, melyben az első komponens újraküldi az előző üzenetet;
- **W**=$(2, well\_received)$, azaz a hibamentes üzenetküldések tartozzanak ide;
- **I**=$(2, ill\_received)$ pedig a hibás üzenetküldéseket gyűjtse össze.

Jelölje rendre az adó és a vevő folyamatot modellező átmeneti rendszerek azon szinkronizált szorzatát, melyet rendre az I, I_e, I_r, és I_{er} szinkronizációs vektorok választásával kapunk. Az $\mathcal{A}_{er}$ átmeneti rendszer látható a 2.15 ábrán.

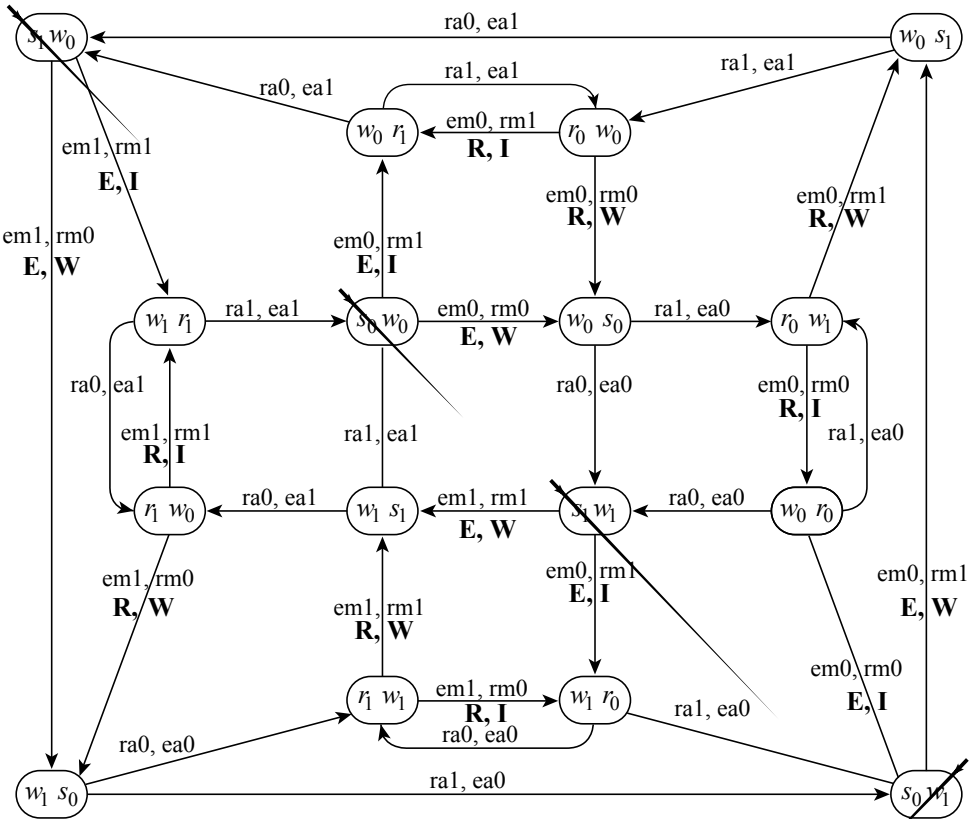
Animált ábra. A 2. animált ábra mind a négy, azaz az $\mathcal{A}$, $\mathcal{A}_e$, $\mathcal{A}_r$ és $\mathcal{A}_{er}$ átmeneti rendszert és azok egymáshoz való viszonyát szemlélteti az összes átmenet címkéjének, illetve csak az átmenetparamétereknek a feltüntetésével.

Talán ebből a példából is kitűnik, hogy az átmeneti rendszerek szinkronizált szorzatával számos rendszer modellezhető, és hogy a címkék és paraméterek hogyan segíthetnek a számunkra lényeges információk megkülönböztetésében.

2.9. Időzített rendszerek

Az eddig vizsgált konkurens rendszerekben a tevékenységek időtartamáról és a különböző helyeken végrehajtott folyamatok végrehajtásának sebességéről semmilyen feltételezéssel nem éltünk. Ez nagy szabadságot biztosít annak ellenőrzésére, hogy a vizsgált rendszerek különleges szituációkban is helyesen működnek-e.

Nem nehéz azonban példát találni olyan rendszerekre, ahol a tevékenységek gyors végrehajtása kritikus fontosságú. Tipikusan ilyenek a beágyazott rendszerek, például az autóban ütközéskor a légzsáknak milliszekundumokban meghatározott időhatárok között kell kinyílnia, sem túl korán, sem túl későn.



2.15. ábra. Az ABP protokoll modellje mindkét irányban megbízhatatlan csatornát feltételezve

2.9.1. Időzített átmeneti rendszerek

Ebben a fejezetben az időzített átmeneti rendszerekkel ismerkedünk meg. Az időzített automaták és az időzített automata hálózatok szemantikájának megadására fogjuk őket használni. Az időzített átmeneti rendszerek közül itt csak egy igen egyszerű változatot tárgyalunk. Nevezetesen csak annyit engedünk meg, hogy az eddigi akciókhoz kötődő átmeneteken túl szerepeljenek várakozó átmenetek is, melyek az idő múlását modellezik.

Definíció. Legyen A egy ábécé. A feletti (valós idejű) *időzített címkézett átmeneti rendszer* alatt olyan $\mathcal{A} = (S, T, \alpha, \beta, \lambda)$ címkézett átmeneti rendszert értünk, melynek címkehalmaza

$$A' = A \cup \{\varepsilon(d) \mid d \in \mathbb{R}_{\geq 0}\},$$

és melyre minden $s \in S$-re és $d \in \mathbb{R}_{\geq 0}$-ra létezik olyan $s' \in S$, melyre

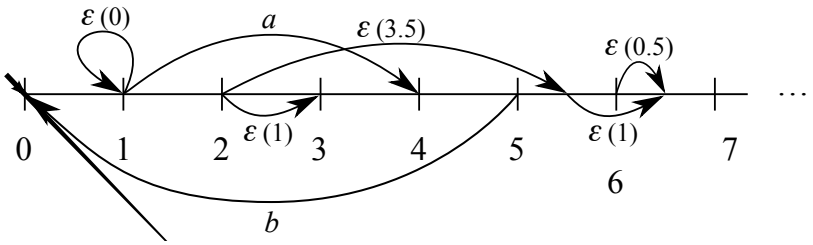
$$s \xrightarrow{\varepsilon(d)} s'$$

átmenet $\mathcal{A}'$-ben, továbbá teljesülnek az alábbi azonosságok:

(1) $\forall s \in S : s \xrightarrow{\varepsilon(0)} s$;

(2) $\forall s \in S, \forall d, d' \in \mathbb{R}_{\geq 0}$:
ha $s \xrightarrow{\varepsilon(d)} s'$ és $0 < d' < d$, akkor $\exists s'' \in S : s \xrightarrow{\varepsilon(d')} s'' \xrightarrow{\varepsilon(d-d')} s'$;

(3) $\forall s, s', s'' \in S, \forall d \in \mathbb{R}_{\geq 0} :$ ha $s \xrightarrow{\varepsilon(d)} s'$ és $s \xrightarrow{\varepsilon(d)} s''$, akkor $s' = s''$.



2.16. ábra. A példában definiált időzített átmeneti rendszer néhány átmenete

Mint láthatjuk, a feltételek közül (1) azt fogalmazza meg, hogy a nulla idejű várakozás nem változtathatja meg a rendszer állapotát, a (2) feltétel az idő additivitását fejezi ki, azaz egy d_1+ $+d_2$ idejű várakozás mindig felbontható egy d_1 idejű, majd egy d_2 idejű várakozás egymás utáni végrehajtására. Végül (3) azt jelenti, hogy a várakozó átmenetek determinisztikusak, azaz egy adott s állapotból adott d idejű várakozással mindig ugyanabba az s' állapotba kell, hogy jussunk.

Belátható, hogy a fenti azonosságoknak következménye az alábbi állítás:

$$\forall s, s', s'' \in S, \forall d, d' \in \mathbb{R}_{\geq 0}: \text{ ha } s \xrightarrow{\varepsilon(d)} s' \text{ és } s' \xrightarrow{\varepsilon(d')} s'', \text{ akkor } s \xrightarrow{\varepsilon(d+d')} s''.$$

Ha szükséges, az időzített átmeneti rendszereket az időzítés nélküli esethez hasonlóan átmenet- és állapotparaméterekkel is elláthatjuk. Például megkülönböztethetünk kezdőállapotokat, és mind az akciókat jelölő, mind a várakozó átmeneteknek paraméter értékeket adhatunk.

Példa. Lássunk egy egyszerű példát. Tekintsük azt a címkézett átmeneti rendszert, melyben az állapotok halmaza $S = \mathbb{R}_{\geq 0}$, az átmenetek pedig az alábbiak:

$$1 \xrightarrow{a} 4$$

$$5 \xrightarrow{b} 0$$

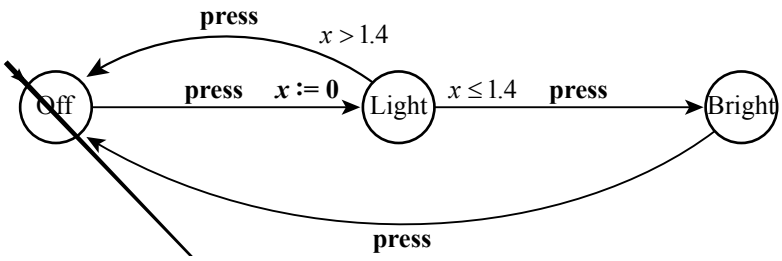
$$r \xrightarrow{\varepsilon(d)} r+d, \text{ minden } r, d \in \mathbb{R}_{\geq 0} \text{ esetén.}$$

A kezdőállapot legyen a 0.

A valós számokon vett összeadás tulajdonságaiból közvetlenül adódik, hogy ennek az átmeneti rendszernek az $\xrightarrow{\varepsilon(d)}$ várakozó átmenetei teljesítik az időzített átmeneti rendszerek definíciójának (1)–(3) feltételeit. A példabeli átmeneti rendszert szemlélteti a 2.16 ábra. Pontosabban, a rendszerben szereplő kontinuum sok átmenete közül nyilván csak néhányat tüntettünk fel. Egyetlen kezdőállapotot különböztettünk meg, a 0-t, melyet a *kezdő* állapotparaméterrel fejezhetünk ki: $S_{kezdő} = \{0\}$. A kezdőállapotot az ábrán egy rá mutató kis nyíl jelöli.

2.9.2. Időzített automaták

Az időzített automatákat valós idejű rendszerek modellezésére használhatjuk. A mára klasszikusnak mondható formalizmust R. Alur és D. L. Dill vezette be [3] a 90-es évek elején. Az



2.17. ábra. Az érintőkapcsolóval vezérelt lámpa időzített automata modellje

időzített automaták és a rájuk vonatkozó hatékony eldöntési algoritmusok adják a valós idejű modell-ellenőrzők, mint például a KRONOS [9, 40], vagy az ötödik fejezetben ismertetésre kerülő UPPAAL [13] működésének elméleti alapjait. A továbbiakban az [1] könyv alapján vezetjük be az időzített automaták fogalmát és azok szorzatát.

Az időzített automaták lényegében olyan nemdeterminisztikus automaták, melyeket véges sok valós idejű órával látunk el. A működés abban tér el a hagyományos automatáktól, hogy egyrészt az órák segítségével az átmenetek végrehajthatóságára feltételeket szabhatunk, azaz bizonyos átmenetek csak akkor hajthatók végre, ha az átmenethez tartozó úgynevezett *őrfeltételt* (guard condition) az órák aktuális állapota teljesíti. Másrészt, lehetőségünk van az átmenetek végrehajtásakor bizonyos órákat alaphelyzetbe állítani, más szóval újraindítani.

Először tekintsünk egy egyszerű példát. A 2.17 ábrán egy érintő kapcsolóval vezérelt lámpa modelljét láthatjuk, mely a következőképpen működik:

Ha a lámpa „Off” (kikapcsolt) állásban van, akkor a kapcsoló egyszeri megnyomásával (vagyis a **press** akcióval), „Light” (világít) állapotba hozhatjuk. A kapcsoló újabb megnyomásakor két dolog történhet. Ha a második gombnyomás az előzőhöz képest gyorsan, pontosabban 1.4 másodpercen belül történik, a lámpa „Bright” (világos) állapotba kerül, míg ha a gombnyomás később következik be, a rendszer „Off” állapotba kerül vissza. A „Bright” állapotból a kapcsoló érintésével mindig „Off” állapotba jut a rendszer.

Ezt a működést az automata átmenetein az őrfeltételek és az $x := 0$ értékadás biztosítja. Az „Off” állapotból a „Light”-ba vezető **press** címkéjének nincs őrfeltétele, ugyanakkor az átmenet végrehajtásakor az $x := 0$ értékadás, vagyis az x óra újraindítása is megtörténik. A „Light” állapotból két átmenet vezet. Mindkettőnek van őrfeltétele: az „Off” állapotba vezető átmenet csak $x > 1.4$ esetben, míg a „Bright” állapotba vivő csak $x \leq 1.4$ esetben hajtható végre. A „Bright”-ból „Off”-ba vezető átmenet nincs őrfeltételhez kötve, bármikor végrehajtható.

2.9.3. Az időzített automaták szintaxisa

Ahogy a példában is láthattuk, az időzített automaták definíciójához órákra, az órák segítségével megfogalmazott őrfeltételekre és az órák újraindítását kiváltó értékadásokra van szükség.

Rögzítsünk egy véges $C = \{x, y, \ldots\}$ halmazt, melynek elemeit óra neveknek, vagy egyszerűen csak *óráknak* nevezzük. Ezeket fogjuk az időzített automatákban használni.

Definíció. Órák egy C halmaza feletti *órafeltételek* (clock constraints) halmaza az a legszűkebb $\mathcal{B}(C)$ halmaz, melyre

- $x \bowtie n \in \mathcal{B}(C)$, minden $x \in C$, $\bowtie \in \{<, \leq, =, \geq, >\}$, $n \in \mathbb{N}$ esetén, és

- $g_1 \wedge g_2 \in \mathcal{B}(C)$, minden $g_1, g_2 \in \mathcal{B}(C)$ esetén.

Példa. Legyen $C = \{x,\ y,\ z\}$. Ekkor a következő kifejezések $\mathcal{B}(C)$-ben vannak:

$$x \geq 10,$$
$$x \leq 7 \wedge x > 2,$$
$$x > 5 \wedge y = 2 \wedge x \leq 1.$$

Példáinkban sokszor a matematikában szokásos jelölésmódot is használni fogjuk, így például $x \leq 7 \wedge x > 2$ helyett $2 < x \leq 7$-et írunk.

Feltesszük, hogy minden C-beli óra azonos ütemben jár, és mindegyik mindig az utolsó újraindítása óta eltelt időt tárolja. Az órák aktuális értékét formálisan a $v : C \to \mathbb{R}_{\geq 0}$ *óra értékelés* (clock valuation) függvény adja meg. Óra értékelés helyett egyszerűen csak értékelést is mondunk. Ha v értékelés, az $x \in C$ óra aktuális értékét $v(x)$ határozza meg. Ha például $C = \{x,\ y\}$, akkor $v(x) = 6.333$, $v(y) = 0.25$ egy lehetséges értékelés. Ezt a v értékelést röviden $[x = 6.333,\ y = 0.25]$-ként is felírhatjuk. Megjegyezzük, hogy az órák értékként akármilyen nemnegatív valós számot felvehetnek. Például $[x = \pi,\ y = \sqrt{2}]$ is megengedett értékelés.

A következőkben két műveletre lesz szükségünk, melyekkel az órák értékét megváltoztathatjuk: az egyik a *szünet* (delay), a másik az *újraindítás* (reset).

Legyen v egy értékelés, d pedig egy nemnegatív valós szám. Ekkor $v + d$ jelölje azt az értékelést, mely minden óra értékét d-vel megnöveli v-hez képest, azaz

$$(v + d)(x) = v(x) + d, \qquad \text{minden } x \in C \text{ órára.}$$

Legyen most $R \subseteq C$ órák halmaza, v pedig egy értékelés. Ekkor azt az értékelést, melyet úgy kapunk v-ből, hogy az R-ben szereplő órákat újraindítjuk $v[R \mapsto 0]$ fogja jelölni. Formálisan:

$$(v[R \mapsto 0])(x) = \begin{cases} 0, & \text{ha } x \in R, \\ v(x), & \text{különben;} \end{cases} \qquad \text{minden } x \in C \text{ órára.}$$

Megjegyzés. Amennyiben $R = \{x\}$ egyelemű halmaz, $v[\{x\} \mapsto 0]$ helyett $v[x \mapsto 0]$-t fogunk írni.

Miután az órafeltételeket és óra értékeléseket definiáltuk, azt kell megadnunk, hogy egy értékelés mikor elégít ki egy órafeltételt, vagyis mikor válik igazzá egy órafeltétel egy értékelés esetén.

Definíció. Legyen $g \in \mathcal{B}(C)$ egy órafeltétel, $v : C \to \mathbb{R}_{\geq 0}$ pedig egy értékelés. Azt, hogy v mikor *elégíti ki* g-t, (jelölése $v \models g$), g felépítése szerinti indukcióval az alábbi módon definiáljuk:

$$v \models x \bowtie n, \text{ akkor és csak akkor, ha } v(x) \bowtie n;$$
$$v \models g_1 \wedge g_2, \text{ akkor és csak akkor, ha } v \models g_1 \text{ és } v \models g_2.$$

Természetesen fentebb $x \in C$ óra, $n \in \mathbb{N}$, $g_1, g_2 \in \mathcal{B}(C)$ és $\bowtie \in \{<, \leq, =, \geq, >\}$. A definíció első sorában az „akkor és csak akkor” bal oldalán a $\bowtie$ jel egy tisztán szintaktikus szimbólumként szerepel, ezzel szemben a jobb oldalon a nemnegatív valós számok körében szokásos valamelyik rendezési relációként szerepel. Amennyiben $v \models g$ nem teljesül, a $v \not\models g$ jelölést használjuk.

Példa. Legyen $C = \{x,\ y\}$ és tekintsük a $v = [x = 1.2, y = 3.01]$ értékelést. Ekkor nem nehéz ellenőrizni, hogy

$$v \models x > 1 \wedge x \leq 2,$$
$$v \models x > 0 \wedge y \geq 3, \text{ és}$$
$$v \not\models y \geq 3 \wedge x \leq 1.$$

Definíció. Azt mondjuk, hogy két órafeltétel, g_1 és g_2 *ekvivalens*, ha ugyanazok az értékelések elégítik ki őket, azaz minden v értékelésre

$$v \models g_1 \iff v \models g_2$$

teljesül.

Az eddigi előkészületek után végre lehetőségünk nyílik az időzített automaták definiálására.

Definíció. Legyen C órák véges halmaza, A pedig egy véges vagy végtelen halmaz, az akciók halmaza. Ekkor A és C feletti *időzített automatán* (timed automaton) az

$$(L, \ell_0, E, I)$$

négyest értjük, ahol

- L a *helyek* (location) véges halmaza,
- $\ell_0 \in L$ a *kezdőhely* (initial location),
- $E \subseteq L \times \mathcal{B}(C) \times A \times \mathcal{P}(C) \times L$ az *átmenetek* vagy élek véges halmaza,
- $I : L \to \mathcal{B}(C)$ pedig a helyekhez *helyinvariánsokat* rendelő függvény.

A továbbiakban L elemeit az ℓ, ℓ' szimbólumokkal és ezek indexelt változataival fogjuk jelölni. Az (l, g, a, R, l') átmenetet $\ell \xrightarrow{g,a,R} \ell'$-ként írjuk, ebben ℓ-et az átmenet *forrás helyének* (source location), ℓ'-t a *cél helyének* (target location), a-t az átmenethez tartozó *akciónak*, g-t *őrfeltételnek* (guard), R-et pedig *óra újraindításoknak* (clock resets) fogjuk hívni.

Az időzített automatákat gyakran grafikusan is ábrázoljuk, mint ezt a 2.17 ábra esetén is tettük. Az ábrákon a helyeket körök, az átmeneteket pedig nyilak szemléltetik, a nyíl kezdetére helyezzük az őrfeltételt, közepére az akciót, végére pedig az újraindítandó órákat. Utóbbiban mindegyik órára külön az $x := 0$ típusú jelölést használjuk. A helyinvariánsokat, vagy a helyet jelölő körbe vagy közvetlenül a kör mellé balra alulra írjuk. Azokat a helyinvariánsokat, őrfeltételeket, melyek mindig igazak, illetve az üres óra újraindításokat nem jelöljük.

Példa. A 2.17 ábrán látható időzített automatát így adhatjuk meg formálisan:

- $C = \{x\}$,
- $L = \{\text{Off, Light, Bright}\}$,

- $\ell_0 = \text{Off}$,

$$E = \{\text{Off} \xrightarrow{\uparrow,\textbf{press},\{x\}} \text{Light},$$
$$\text{Light} \xrightarrow{x>14,\textbf{press},\emptyset} \text{Off},$$
$$\text{Light} \xrightarrow{x\leq 14,\textbf{press},\emptyset} \text{Bright},$$
$$\text{Bright} \xrightarrow{\uparrow,\textbf{press},\emptyset} \text{Off},$$

- $I(\text{Off}) = I(\text{Light}) = I(\text{Right}) = \uparrow$,

ahol $\uparrow$ egy azonosan igaz órafeltétel, például $\uparrow = (x \geq 0)$.
Megjegyzés. Vegyük észre, hogy a fenti definícióban az ábrától eltérően az 1.4 időkorlátot 14-gyel helyettesítettük, mert az órafeltételekben csak egész számok szerepelhetnek. Ez azonban nem jelent valódi megszorítást. Nyugodtan szerepeltethetnénk racionális számokat is az időzített automatákban, hiszen egyszerre csak véges sok átmenet szerepelhet egy automatában, és ha a nevezőjük legkisebb közös többszörösével végigszorozzuk őket, az eredetivel ekvivalens viselkedésű automatához jutunk. Valóban ez csak annyit jelent, hogy például nem 0.145, hanem, mondjuk, 145 szerepel az órafeltételekben, ezt pedig tekinthetjük úgy, mintha nem másodpercben, hanem milliszekundumokban mérnénk az időt.

2.9.4. Az időzített automaták szemantikája

Ebben a fejezetben az időzített automaták működését definiáljuk az előzőekben bevezetett időzített átmeneti rendszerek segítségével. Ahogy az eddigi példából is kitűnik, úgy gondoljuk, hogy az időzített automata az időzítés nélküli társaihoz hasonlóan a számítás során mindig egyetlen helyen tartózkodik. Csupán a hely ismerete azonban nem elég, nyilván kell tartanunk az órák aktuális értékét is, hogy az őrfeltételeknek megfelelő átmenetek közül választhassunk. Ezért az időzített automaták *számításainak állapotai* (vagy, ha jobban tetszik, pillanatfelvételei vagy konfigurációi) (ℓ, v) rendezett párok lesznek, ahol ℓ az automata aktuális helye, v pedig az automata óráinak aktuális értékelése. Ez magyarázza, miért helyeknek és nem állapotoknak hívjuk az időzített automaták helyeit. Egy dologra kell még tekintettel lennünk, minden (ℓ, v) állapotnak ki kell még elégíteni a $v \models I(\ell)$ feltételt, azaz az óra értékelésnek mindig ki kell elégítenie a helyhez tartozó invariánst.

Tekintsünk egy A akció- és C órahalmaz feletti $\mathcal{A} = (L, \ell_0, E, I)$ időzített automatát. A számítás kezdetén az automata az ℓ_0 kezdőhelyen tartózkodik és minden óra értéke 0.

Az automatának kétfajta átmenete van. Az egyik lehetőség az automata valamelyik éléhez tartozó *akció* végrehajtása, a másik a *várakozó átmenetek*.

Tegyük fel, hogy az aktuális állapot (ℓ, v), azaz ℓ egy hely, $v : C \to \mathbb{R}_{\geq 0}$ egy értékelés, és a $v \models I(\ell)$ feltétel teljesül.

Ebben az állapotban akkor hajthat végre akciót az automata, ha van olyan ℓ-ből induló él E-ben, mondjuk $\ell \xrightarrow{g,a,R} \ell'$, melyhez tartozó g őrfeltételt kielégíti v. Ekkor az a akció végrehajtásával az ℓ' helyre juthat az automata, miközben az R-ben felsorolt órákat zérusra állítja. Tehát az új állapot (ℓ', v')-lesz, ahol $v' = v[R \mapsto 0]$. Ennek az átmenetnek a végrehajtása nem kerül időbe, így az R halmazban nem szereplő órák értéke az átmenet előtt és után megegyezik.

Az akciókon kívül, az (ℓ, v) állapotban $d \geq 0$ idejű várakozó átmenetet is végrehajthatunk. Ennek csupán annyi a feltétele, hogy az ℓ állapothoz kapcsolódó $I(\ell)$ invariáns feltétel ne sérüljön az idő múlásával. Nem nehéz látni, hogy ezt elég csak az utolsó megkövetelt pillanatban, azaz $v+d$ óra értékeléskor ellenőriznünk. Tehát, amennyiben $v+d \models I(\ell)$, akkor a rendszer (ℓ, v) állapotból $(\ell, v+d)$ állapotba léphet. Ennek jele $(\ell, v) \xrightarrow{\varepsilon(d)} (\ell, v+d)$ lesz. Vagyis a várakozó átmenetek végrehajtásakor a hely nem változik, de az összes automatában szereplő óra ideje egyenlő mértékben nő. Speciálisan az $\varepsilon(0)$ várakozó átmenet mindig végrehajtható. A formális definíció a következő:

Definíció. Legyen $\mathcal{A} = (L, \ell_0, E, I)$ egy időzített automata valamely akciók A és órák C halmaza felett. Az $\mathcal{A}$ *automatához rendelt T(A) átmeneti rendszer* alatt azt az A feletti időzített átmeneti rendszert értjük, melynek állapotai:

$$S = \{(\ell, v) \mid \ell \in L, v : C \to \mathbb{R}_{\geq 0}, v \models I(l)\},$$

átmenetei pedig:

$(\ell, v) \xrightarrow{a} (\ell', v')$, amennyiben $(\ell \xrightarrow{g,a,R} \ell') \in E, v \models g, v' = v[R \to 0]$ és $v' \models I(\ell')$,
és

$(\ell, v) \xrightarrow{\varepsilon(d)} (\ell, v+d)$, minden $d \in \mathbb{R}_{\geq 0}$ esetén, ha $v \models I(\ell), v+d \models I(l)$.

Továbbá az átmeneti rendszerben egyetlen *kezdő* nevű állapotparaméter van, és

$$S_{kezdő} = \{(\ell_0, v_0)\},$$

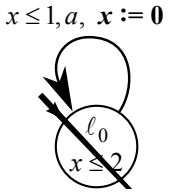
ahol v_0 a minden órához 0 értéket rendelő értékelés, vagyis $v_0(x) = 0$, minden $x \in C$-re. Ezen felül itt és a továbbiakban mindig feltesszük, hogy csak olyan időzített automatákkal dolgozunk, melyekben a kezdőhely invariánsát kielégíti v_0, azaz $v_0 \models I(\ell_0)$.

Nem nehéz látni, hogy az így definiált $T(\mathcal{A})$ valóban időzített átmeneti rendszer, mert kielégíti az időzített átmeneti rendszer definíciójában megkövetelt (1)-(3) feltételek mindegyikét.

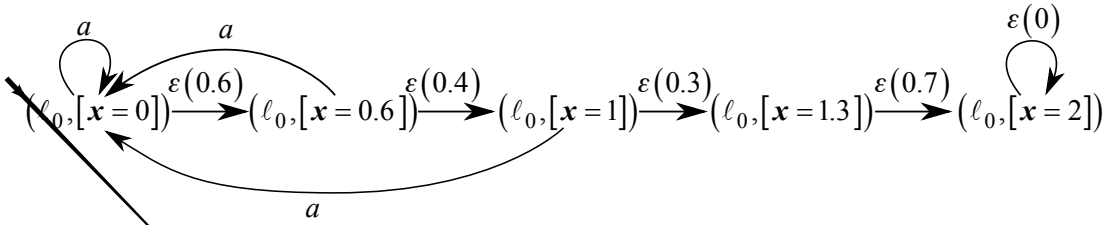
Példa. Tekintsük a 2.18 ábrán látható $\mathcal{A}$ automatát. Ennek az automatának csak egy állapota van, ℓ_0, melyhez az $I(\ell_0) = x \leq 2$ invariáns feltétel tartozik. Az egyetlen átmenet pedig

$$\ell_0 \xrightarrow{x \leq 1, a, \{x\}} \ell_0,$$

azaz, ha az $x \leq 1$ őrfeltétel teljesül, lehetőségünk van az a akciót végrehajtani, és ezzel egyidejűleg az x órát újraindítani. Az $\mathcal{A}$-hoz társított $T(\mathcal{A})$ időzített átmeneti rendszer néhány jellemző átmenetét a 2.19 ábrán láthatjuk.



2.18. ábra. A példában szereplő időzített automata



2.19. ábra. A példában szereplő $T(\mathcal{A})$ időzített átmeneti rendszer néhány átmenete

2.10. Időzített automaták szorzata

Nyilvánvaló, hogy az időzített rendszerek is általában több, párhuzamosan futó, időnként kommunikáló komponensek együtteséből állnak. Például egy ipari gyártósoron egy-egy konkrét részfeladat ellátására tervezett érzékelők és beavatkozók működnek szinkronizáltan.

Az összetett időzített rendszerek modellezésére az alábbiakban ismertetésre kerülő *időzített automata hálózatokat* fogjuk használni.

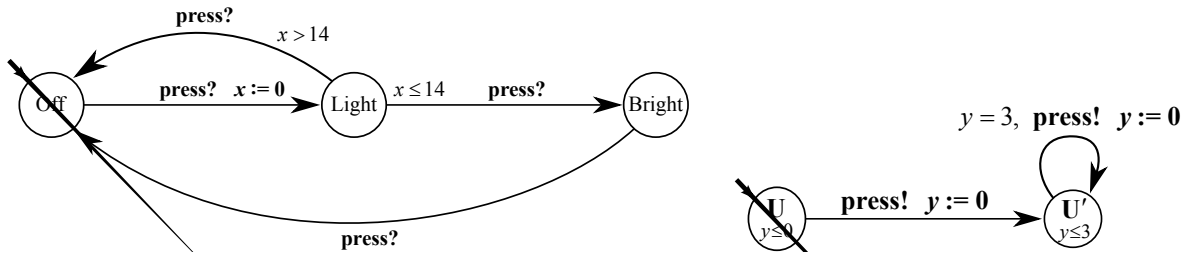
A kommunikációs lehetőségek közül csak az egyik legalapvetőbbet, a *kézfogáson alapuló szinkronizációt* (hand-shake synchronization) fogjuk bemutatni, mely az UPPAAL modellellenőrzőben is szerepel.

A szinkronizációra *szinkronizációs csatornákat* (synchronization channels) és *kommunikációs átmeneteket* használunk.

A továbbiakban rögzítsük a csatornák egy $Ch = \{a, b, \ldots\}$ halmazát. Ezt a halmazt úgy válasszuk meg, hogy diszjunkt legyen az összes tekintett időzített automata óra- és címkehalmazától. A kommunikációs átmenetek címkéi mindig egy-egy csatornához kötődnek és párokban fordulnak elő. Amennyiben $a \in Ch$, vagyis a egy csatorna, $a!$ illetve $a?$ az a csatornához tartozó kommunikációs átmenet pár.

Az $a!$ átmenetnek az a jelentése, hogy az a komponens, mely ezt végre kívánja hajtani, szinkronizációs igényt jelent be az a csatornán keresztül. Egy másik komponensben pedig $a?$ az első komponens kommunikációs igényének az elfogadására szolgál. A szinkronizáció csak akkor valósulhat meg, ha a két komponens egyszerre képes a saját küldő, illetve fogadó átmenetét végrehajtani. Ezért hívjuk ezt kézfogás típusú kommunikációnak, ugyanis mindig szinkronban történik. Fontos, hogy azt is feltesszük, hogy az akciókhoz rendelt, (azaz nem várakozó) átmenetekhez hasonlóan, a kommunikációs átmenetek is pillanatnyiak, azaz nulla időegység alatt játszódnak le. Ebben a speciális esetben adatátadás nem zajlik a két komponens között. Nézzünk ismét egy egyszerű példát.

Példa. A már bemutatott, az érintések között eltelt időre érzékeny kapcsoló és lámpa mellé tekintsünk egy olyan „türelmetlen felhasználót”, aki minden harmadik időegység leteltekor

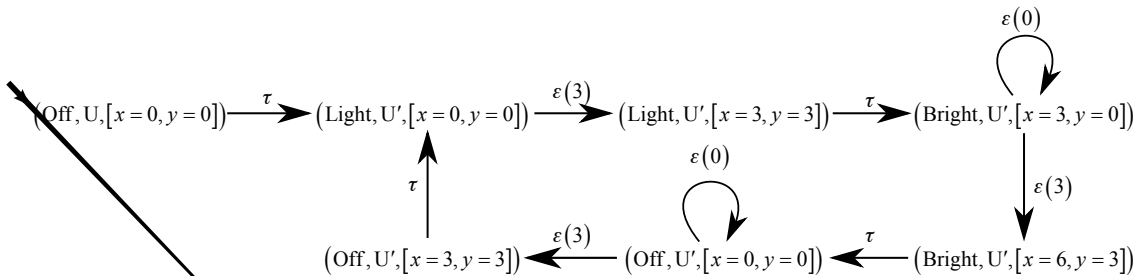


2.20. ábra. A példában említett két időzített automata

megérinti a kapcsolót. A 2.20 ábrán láthatjuk a hozzájuk tartozó két időzített automatát. A kapcsoló automatájában a konstansok értékét tízzel megszoroztuk, hogy egész számokat kapjunk.

A két komponens közötti kommunikáció a **press** csatornán keresztül zajlik, a **press!** és **press?** átmenetek egyidejű végrehajtásával. A felhasználó és a kapcsoló külön órákat használ, y-t és x-et, ezek azonban az összetett rendszer vizsgálatakor egyszerre és egyenlő lépték-kel növekszenek. Ha a megfelelő őrfeltételek teljesülnek, a felhasználó egy **press!** címkéjű átmenete szinkronban hajtható végre a kapcsoló egy **press?** címkéjű átmenetével. Az összetett rendszerben a kommunikációs átmenetek eredeti címkéi már nem láthatók, helyükre egy speciális címke, τ kerül. Ez azt modellezi, hogy a komponensekben a kommunikációt megvalósító komponensek belső átmenetei a rendszeren kívülről már nem láthatók.

Az összetett rendszer működése a 2.21 ábrán található átmeneti rendszerrel írható le. Mivel a várakozásokból adódóan végtelen sok lehetséges átmenet van, csak néhány jellemző átmenetet tudunk feltüntetni.



2.21. ábra. A példában szereplő időzített átmeneti rendszer néhány átmenete

Definíció. Legyen n pozitív egész szám, és tekintsünk n darab időzített automatát:

$$\mathcal{A}_i = (L_i, \ell_0^i, E_i, I_i), \quad 1 \le i \le n.$$

Az egyszerűség kedvéért tegyük fel, hogy az automaták óra-, csatorna- és akcióhalmaza azonos, jelölje ezeket a halmazokat rendre C, Ch és A. Ekkor az $\mathcal{A}_i$ *időzített automaták szorzata* az alábbi jelölés szerinti:

$$\langle \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n \rangle.$$

A fenti szorzatot *időzített automata hálózatnak* is nevezzük.

Persze a fenti definíció csak jelölés, az időzített automata hálózatok szemantikáját, azaz működését, a hálózathoz rendelt átmeneti rendszer definiálásával adhatjuk meg.

Definíció. Legyen $\mathcal{A} = \langle \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n \rangle$ az előző definícióban szereplő időzített automata hálózat, megtartva az ott bevezetett többi jelölést is. Feltesszük még, hogy az A közös akcióhalmaz az A_c kommunikációs és A_n normál akciók halmazára bontható. Azaz, $A = A_c \cup A_n$, ahol $A_c = \{a!, a? \mid a \in Ch\}$, és $A_c \cap A_n = \emptyset$. A hálózathoz rendelt $T(\mathcal{A})$ átmeneti rendszer az a címkézett átmeneti rendszer, melynek

állapotai:

$$S = \left\{ (\ell_1, \ell_2, \ldots, \ell_n, v) \mid \ell_i \in L_i, 1 \le i \le n, v : C \to \mathbb{R}_{\ge 0}, v \models \bigwedge_{i=1}^{n} I_i(\ell_i) \right\};$$

címkehalmaza:

$$A_{pr} = A_n \cup \{\tau\} \cup \{\varepsilon(d) \mid d \in \mathbb{R}_{\ge 0}\};$$

átmenetei:

(1) $(\ell_1, \ldots, \ell_i, \ldots, \ell_n, v) \xrightarrow{a} (\ell_1, \ldots, \ell_i', \ldots, \ell_n, v')$, amennyiben $a \in A_n$, $(\ell_i \xrightarrow{g,a,R} \ell_i') \in E_i$, $v \models g$, $v' = v[R \to 0]$ és $v' \models I_i(\ell_i') \wedge \bigwedge_{k\ i} I_k(\ell_k)$;

(2) $(\ell_1, \ldots, \ell_i, \ldots, \ell_j, \ldots, \ell_n, v) \xrightarrow{\tau} (\ell_1, \ldots, \ell_i', \ldots, \ell_j', \ldots, \ell_n, v')$, ahol $i \ \ j$, amennyiben $(\ell_i \xrightarrow{g_i,\alpha,R_i} \ell_i') \in E_i$ és $(\ell_j \xrightarrow{g_j,\beta,R_j} \ell_j') \in E_j$, ahol α és β komplementer kommunikációs akciók, azaz $\{\alpha, \beta\} = \{d!, d?\}$, valamely $d \in Ch$ csatornára, továbbá $v \models g_i \wedge g_j$, $v' = v[R_i \cup R_j \to 0]$ és $v' \models I_i(\ell_i') \wedge I_j(\ell_j') \wedge \bigwedge_{k\ i,j} I_k(\ell_k)$;

és

(3) $(\ell_1, \ldots, \ell_i, \ldots, \ell_n, v) \xrightarrow{\varepsilon(d)} (\ell_1, \ldots, \ell_i, \ldots, \ell_n, v+d)$, minden $d \in \mathbb{R}_{\ge 0}$ esetén, ha $v + d' \models \bigwedge_{i \in \{1,\ldots,n\}} I_i(\ell_i)$, minden d' valós számra az $[0, d]$ intervallumból.

Továbbá az átmeneti rendszerben egyetlen *kezdő* nevű **állapotparaméter** van, és

$$S_{kezdő} = \left\{ (\ell_0^1, \ldots, \ell_0^n, v_0) \right\},$$

ahol v_0 a minden órához 0 értéket rendelő értékelés.

Szavakkal kifejezve ez a következőket jelenti. Az $\langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ időzített automata hálózatban a komponensek az A_n-beli hagyományos (azaz nem kommunikációs) akcióihoz átmeneteiket egymástól függetlenül hajtják végre. Az ilyen, (1) pontban definiált átmenetek végrehajtásának feltétele hasonló, mint egyetlen automata esetében: a pillanatnyi óra értékelésnek ki kell elégíteni az átmenethez tartozó őrfeltételt és végrehajtás után is minden komponens helyének teljesítenie kell a helyhez tartozó invariánst.

A (2) pontban vannak a kommunikációs akciókhoz tartozó átmenetek. Az $\mathcal{A}_i$ kommunikál az $\mathcal{A}_j$ komponenssel. A szinkronizáció szempontjából mindegy, hogy melyikük a küldő,

melyikük a fogadó fél. Természetesen mindkét átmenet őrfeltételének igaznak kell lenni és az átmenet végrehajtása során minden aktuális helynek ki kell elégíteni a helyhez tartozó invariáns feltételt.

Végül a (3) várakozó átmenetek a rendszer minden komponensére egyszerre vonatkoznak, és az idő egyformán telik az összes komponensben. Itt is a várakozás teljes ideje alatt az invariáns feltételeknek fenn kell állni.

3. fejezet

Temporális logikák

Az előző fejezet az egymásra ható, dinamikusan változó, folyamatosan működő rendszerek átmeneti rendszerekkel való modellezését tárgyalta. Az átmeneti rendszerek állapotaira, átmeneteire, útjaira vonatkozó tulajdonságok megfogalmazására alkalmas eszköz a logika. Számos ilyen logika ismert. Ezek az átmeneti rendszer típusától (pl. címkézett, paraméterezett, időzített) és a vizsgálandó tulajdonság típusától (pl. lokális, időbeli állapot, út) függően alkalmazhatók.

A kijelentés logika formuláival az átmeneti rendszer állapotainak, illetve átmeneteinek lokális tulajdonságai fejezhetők ki.

A temporális logikák segítségével átmeneti rendszerek időbeli változásai adhatók meg. Az időbeliség alatt itt események sorrendiségét értjük az időtartományban, ami lehet diszkrét, illetve folytonos. A temporális logikák közül azok, melyek az időt egy idővonal mentén időpillanatok egymást követő sorozataként kezelik, vagyis lineárisan, az átmeneti rendszerben levő út tulajdonságok leírását teszik lehetővé. Vannak temporális logikák, melyek kezelni tudják azt, hogy egy eseménynek több rákövetkezője lehet, ezek alkalmasak az átmeneti rendszer elágazó struktúráját figyelembe vevő állapot tulajdonságok specifikálására.

A fejezet anyaga az [1], [4], [5], [14], [16], [24], [25] irodalmak alapján készült.

3.1. Kijelentés logika (Propositional Logic)

Legyen $\mathcal{A} = (S, T, \alpha, \beta, S_{x_1}, \ldots, S_{x_n}, T_{y_1}, \ldots, T_{y_m})$ $(\mathcal{X}, \mathcal{Y})$-paraméteres átmeneti rendszer $(\mathcal{X} = \{x_1, \ldots, x_n\}, \mathcal{Y} = \{y_1, \ldots, y_m\})$.
A logika szimbólumai:

- $\underline{0}, \underline{1}$ állapot logikai konstansok,
- P_x állapot atomi kijelentés, minden $x \in \mathcal{X}$-re,
- $\wedge$ (logikai és) kétváltozós operátor,
- $\neg$ (logikai negáció) egyváltozós operátor.

Jelölje $AP = \{P_x \mid x \in \mathcal{X}\}$ az állapot atomi kijelentések halmazát. Akkor az AP feletti kijelentés logika *állapotformulái*:

- $\underline{0}, \underline{1}$ állapot logikai konstansok,
- a, minden $a \in AP$-re,
- ha f_1 és f_2 állapotformulák, akkor $f_1 \wedge f_2$ állapotformula,
- ha f állapotformula, akkor $\neg f$ állapotformula.

Tetszőleges f állapotformula és $s \in S$ állapot esetén jelölje $\mathcal{A}, s \models f$ azt a tényt, hogy az $\mathcal{A}$ átmeneti rendszer az s állapotában kielégíti az f állapotformulát. Ezt az f állapotformula felépítése szerinti indukcióval a következőképpen definiáljuk:

- ha $f = \underline{0}$, akkor $\mathcal{A}, s \not\models f$,
- ha $f = \underline{1}$, akkor $\mathcal{A}, s \models f$,
- ha $f = a$, ahol $a \in AP$ és $a = P_x$ valamely $x \in \mathcal{X}$-re, akkor $\mathcal{A}, s \models f \Leftrightarrow s \in S_x$,
- ha $f = f_1 \wedge f_2$, akkor $\mathcal{A}, s \models f \Leftrightarrow \mathcal{A}, s \models f_1$ és $\mathcal{A}, s \models f_2$,
- ha $f = \neg f'$, akkor $\mathcal{A}, s \models f \Leftrightarrow \mathcal{A}, s \not\models f'$.

Az állapotformulákhoz hasonlóan átmenetformulák is definiálhatók a logikában.
Az átmenetformulákban megjelenő szimbólumok:

- $\underline{0}_\tau, \underline{1}_\tau$ átmenet logikai konstansok,
- Q_y atomi átmenet kijelentés minden $y \in \mathcal{Y}$-ra,
- $\wedge$ kétváltozós operátor,
- $\neg$ egyváltozós operátor.

Jelölje $AP_\tau = \{Q_y \mid y \in \mathcal{Y}\}$ az átmenet atomi kijelentések halmazát. Akkor az AP_τ feletti kijelentés logika *átmenetformulái*:

- $\underline{0}_\tau, \underline{1}_\tau$ átmenet logikai konstansok,
- a_τ, ahol $a_\tau \in AP_\tau$,
- ha g_1 és g_2 átmenetformulák, akkor $g_1 \wedge g_2$ átmenetformula,
- ha g átmenetformula, akkor $\neg g$ átmenetformula.

Tetszőleges g átmenetformula és $t \in T$ átmenet esetén jelölje $\mathcal{A}, t \models g$, azt, hogy az $\mathcal{A}$ átmeneti rendszer t átmenete kielégíti a g átmenetformulát. Ezt a g átmenetformula felépítése szerinti indukcióval a következőképpen definiáljuk:

- ha $g = \underline{0}_\tau$, akkor $\mathcal{A}, t \not\models g$,
- ha $g = \underline{1}_\tau$, akkor $\mathcal{A}, t \models g$,

- ha $g = a_\tau$, ahol $a_\tau \in AP_\tau$ és $a_\tau = Q_y$ valamely $y \in \mathcal{Y}$-ra, akkor $\mathcal{A}, t \models g \Leftrightarrow t \in T_y$,
- ha $g = g_1 \wedge g_2$, akkor $\mathcal{A}, t \models g \Leftrightarrow \mathcal{A}, t \models g_1$ és $\mathcal{A}, t \models g_2$,
- ha $g = \neg g'$, akkor $\mathcal{A}, t \models g \Leftrightarrow \mathcal{A}, t \not\models g'$.

A kijelentés logika állapotformuláival az átmeneti rendszer állapotaira vonatkozó lokális tulajdonságok írhatók le, míg az átmenetformuláival lokális átmenet tulajdonságok.

3.2. Lineáris temporális logika (Linear Temporal Logic)

A *lineáris temporális logika*, röviden LTL, átmeneti rendszerek útjai tulajdonságainak a leírására alkalmas logika.

Legyen $\mathcal{A} = (S, T, \alpha, \beta, S_{x_1}, \ldots, S_{x_n})$ egy $(\mathcal{X}, \emptyset)$-paraméteres átmeneti rendszer ($\mathcal{X} = \{x_1, \ldots, x_n\}$), melyről feltesszük, hogy minden állapotnak van rákövetkezője.

Jelölje $AP = \{P_x \mid x \in \mathcal{X}\}$ az állapot atomi kijelentések halmazát. Akkor az AP feletti LTL logika szimbólumai:

- az AP feletti kijelentés logika szimbólumai,
- **X** (next) egyváltozós temporális operátor,
- **U** (until) kétváltozós temporális operátor.

A logika *útformulái*:

- $\underline{0}, \underline{1}$ logikai konstansok,
- a, ahol $a \in AP$,
- ha f_1 és f_2 formulák, akkor $f_1 \wedge f_2$ formula,
- ha f formula, akkor $\neg f$ formula,
- ha f formula, akkor $\mathbf{X} f$ formula,
- ha f_1 és f_2 formulák, akkor $f_1 \mathbf{U} f_2$ formulák.

Gyakran használt rövidítések:

$$\mathbf{F} f = \underline{1}\mathbf{U} f, \qquad f_1 \mathbf{B} f_2 = \neg((\neg f_1)\mathbf{U} f_2), \quad \mathbf{G} f = \neg\mathbf{F}\neg f,$$
$$\mathbf{G}^\infty f = \mathbf{F}\mathbf{G} f, \quad \mathbf{F}^\infty f = \mathbf{G}\mathbf{F} f.$$

F helyett a $\Diamond$ és **G** helyett a $\Box$ jelölést is gyakran alkalmazzák. Az **U** operátor helyett ennek gyenge **W** változatát is szokták használni. Tetszőleges f_1 és f_2 útformulákra

$$f_1 \mathbf{W} f_2 = (f_1 \mathbf{U} f_2) \vee \mathbf{G} f_1.$$

Legyen $c = s_0, s_1, \ldots$ egy tetszőleges végtelen út az $\mathcal{A}$ átmeneti rendszerben, ahol minden $i \geq 0$-ra $s_i \in S$. Vezessük be a $c^i = s_i, s_{i+1}, \ldots$ jelölést, ahol $i \geq 0$.

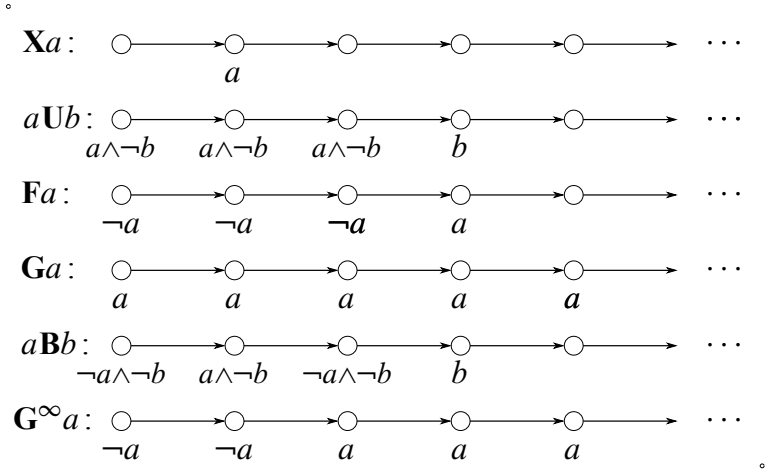
Az $\mathcal{A}, c \models f$ kielégítési reláció definiciója az f útformula felépítése szerinti indukcióval a következő:

- ha $f = \underline{0}$, akkor $\mathcal{A}, c \not\models f$,
- ha $f = \underline{1}$, akkor $\mathcal{A}, c \models f$,
- ha $f = a$, ahol $a \in AP$ és $a = P_x$ valamely $x \in \mathcal{X}$-re, akkor $\mathcal{A}, c \models f \Leftrightarrow s_0 \in S_x$,
- ha $f = f_1 \wedge f_2$, akkor $\mathcal{A}, c \models f \Leftrightarrow \mathcal{A}, c \models f_1$ és $\mathcal{A}, c \models f_2$,
- ha $f = \neg f'$, akkor $\mathcal{A}, c \models f \Leftrightarrow \mathcal{A}, c \not\models f'$,
- ha $f = \mathbf{X} f'$, akkor $\mathcal{A}, c \models f \Leftrightarrow \mathcal{A}, c^1 \models f'$,
- ha $f = f_1 \mathbf{U} f_2$, akkor $\mathcal{A}, c \models f \Leftrightarrow \exists j \geq 0$, hogy $\mathcal{A}, c^j \models f_2$ és $\forall 0 \leq i < j$-re $\mathcal{A}, c^i \models f_1$.

A most definiált kielégítési reláció alapján a bevezetett rövidítések szemantikája:

- $\mathcal{A}, c \models \mathbf{F} f \Leftrightarrow \exists i \geq 0$, hogy $\mathcal{A}, c^i \models f$.
- $\mathcal{A}, c \models \mathbf{G} f \Leftrightarrow \forall i \geq 0$-ra $\mathcal{A}, c^i \models f$.
- $\mathcal{A}, c \models \mathbf{F}^\infty f \Leftrightarrow \forall i \geq 0$-ra $\exists j \geq i$, hogy $\mathcal{A}, c^j \models f$.
- $\mathcal{A}, c \models \mathbf{G}^\infty f \Leftrightarrow \exists i \geq 0$, hogy $\forall j > i$-re $\mathcal{A}, c^j \models f$.
- $\mathcal{A}, c \models f_1 \mathbf{B} f_2 \Leftrightarrow \forall j \geq 0$-ra, ha $\mathcal{A}, c^j \models f_2$, akkor $\exists i < j$, hogy $\mathcal{A}, c^i \models f_1$.

A 3.1. ábra temporális operátort tartalmazó LTL formulákat kielégítő egy-egy utat szemléltet, ahol $AP = \{a, b\}$.

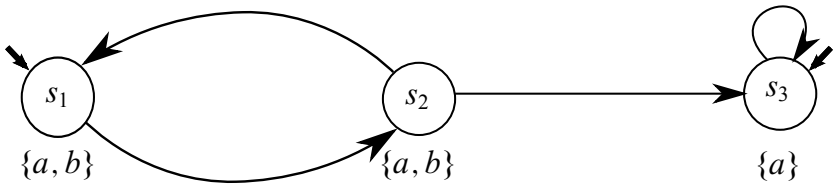


3.1. ábra. Példák LTL formulákat kielégítő egy-egy útra

Legyen $s \in S$ egy tetszőleges állapota az $\mathcal{A}$ átmeneti rendszernek és f egy AP feletti LTL formula. Azt mondjuk, hogy az $\mathcal{A}$ átmeneti rendszer s állapota kielégíti az f formulát, melynek jelölése $\mathcal{A}, s \models f$, ha az s állapotból induló minden c végtelen útra teljesül, hogy $\mathcal{A}, c \models f$.

Legyen $S_0 \subseteq S$ az $\mathcal{A}$ átmeneti rendszer kezdőállapotainak a halmaza. Azt mondjuk, hogy az $\mathcal{A}$ *átmeneti rendszer kielégíti az* f *formulát*, melynek jelölése $\mathcal{A} \models f$, ha minden $s \in S_0$-ra $\mathcal{A}, s \models f$ teljesül.

Példa. Legyen az $\mathcal{A}$ átmeneti rendszer a 3.2. ábrán látható, ahol $\mathcal{X} = \{a, b\}$ és az $x \in \mathcal{X}$-hez az állapot atomi kijelentés x.
Akkor teljesülnek az alábbi relációk:



3.2. ábra. $\{a, b\}$ *állapot-paraméteres* $\mathcal{A}$ *átmeneti rendszer*

$$
\begin{aligned}
\mathcal{A}, s_i &\models \mathbf{G}(a) \text{ minden } i = 1,2,3\text{-ra},\\
\mathcal{A}, s_1 &\models \mathbf{X}(a \wedge b),\\
\mathcal{A}, s_2 &\not\models \mathbf{X}(a \wedge b),\\
\mathcal{A}, s_3 &\not\models \mathbf{X}(a \wedge b),\\
\mathcal{A} &\not\models \mathbf{X}(a \wedge b),\\
\mathcal{A} &\models \mathbf{G}(\neg b \rightarrow \mathbf{G}(a \wedge \neg b)),\\
\mathcal{A} &\not\models b\mathbf{U}(a \wedge \neg b).
\end{aligned}
$$

Megjegyzés. Átmenet-paraméteres átmeneti rendszerekre is definiálható az LTL. Ekkor az LTL formuláinál az AP-beli állapot atomi kijelentések helyett az AP_τ-beli átmenet atomi kijelentések jelennek meg. Egy tetszőleges g LTL útformula és $c = t_0, t_1, \ldots$ végtelen útra, ahol $t_i \in T$ és $\beta(t_i) = \alpha(t_{i+1})$ minden $i \geq 0$-ra az $\mathcal{A}, c \models g$ definíciója az g felépítése szerinti indukcióval a következő:

- ha $g = \underline{0}_\tau$, akkor $\mathcal{A}, c \not\models g$,
- ha $g = \underline{1}_\tau$, akkor $\mathcal{A}, c \models g$,
- ha $g = a_\tau$, ahol $a_\tau \in AP_\tau$ és $a_\tau = Q_y$ valamely $y \in \mathcal{Y}$-ra, akkor $\mathcal{A}, c \models g \Leftrightarrow t_0 \in T_y$,
- ha $g = g_1 \wedge g_2$, akkor $\mathcal{A}, c \models g \Leftrightarrow \mathcal{A}, c \models g_1$ és $\mathcal{A}, c \models g_2$,
- ha $g = \neg g'$, akkor $\mathcal{A}, c \models g \Leftrightarrow \mathcal{A}, c \not\models g'$,
- ha $g = \mathbf{X}g'$, akkor $\mathcal{A}, c \models g \Leftrightarrow \mathcal{A}, c^1 \models g'$, ahol $c^1 = t_1, t_2, \ldots$,
- ha $g = g_1 \mathbf{U} g_2$, akkor $\mathcal{A}, c \models g \Leftrightarrow \exists j \geq 0$, hogy $\mathcal{A}, c^j \models g_2$ és $\forall 0 \leq i < j$-re $\mathcal{A}, c^i \models g_1$, ahol c^i a $t_i, t_{i+1}, \ldots$ végtelen utat jelöli.

3.3. HML logika (Hennessy-Milner logika)

A HML logika formuláival állapot tulajdonságok adhatók meg címkézett átmeneti rendszerekben.

Jelöljön A egy tetszőleges véges címkehalmazt. Az A címkehalmaz feletti logika szimbólumai:

- $\underline{0}, \underline{1}$ logikai konstansok,
- $\wedge$ (logikai és) kétváltozós operátor,
- $\neg$ (logikai negáció) egyváltozós operátor,
- $\langle a \rangle$ minden $a \in A$-ra, egyváltozós operátorok.

Az A címkehalmaz feletti logika *állapotformulái*:

- $\underline{0}, \underline{1}$ logikai konstansok,
- ha f_1 és f_2 állapotformulák, akkor $f_1 \wedge f_2$ állapotformula,
- ha f állapotformula, akkor $\neg f$ állapotformula,
- ha f állapotformula, akkor $\langle a \rangle f$ állapotformula, minden $a \in A$-ra.

Legyen $\mathcal{A}=(S, T, \alpha, \beta, \lambda)$ egy A címkehalmaz feletti címkézett átmeneti rendszer. Tetszőleges $s \in S$ állapotra és f állapotformulára jelölje $\mathcal{A}, s \models f$ azt, hogy az $\mathcal{A}$ átmeneti rendszer kielégíti az s állapotban az f formulát. Ezt az f állapotformula felépítése szerinti indukcióval a következőképpen definiáljuk:

- ha $f = \underline{0}$, akkor $\mathcal{A}, s \not\models f$,
- ha $f = \underline{1}$, akkor $\mathcal{A}, s \models f$,
- ha $f = f_1 \wedge f_2$, akkor $\mathcal{A}, s \models f \Leftrightarrow \mathcal{A}, s \models f_1$ és $\mathcal{A}, s \models f_2$,
- ha $f = \neg f'$, akkor $\mathcal{A}, s \models f \Leftrightarrow \mathcal{A}, s \not\models f'$,
- ha $f = \langle a \rangle f'$, akkor $\mathcal{A}, s \models f \Leftrightarrow \exists t \in T$, hogy $\alpha(t) = s$, $\lambda(t) = a$ és $\mathcal{A}, \beta(t) \models f'$.

Legyen $S_0 \subseteq S$ az $\mathcal{A}$ átmeneti rendszer kezdőállapotainak halmaza. Azt mondjuk, hogy az $\mathcal{A}$ kielégíti az f formulát, amit az $\mathcal{A} \models f$ jelöl, ha $\mathcal{A}, s \models f$ teljesül minden $s \in S_0$-ra.

A logika bevezetett operátoraival kifejezhető gyakran alkalmazott rövidítés az $[a] f = \neg \langle a \rangle \neg f$.

A HML logika az $\{\langle a \rangle \mid a \in A\}$-beli operátorai alapján elágazó idejű temporális logika, mivel lehetőséget ad arra, hogy egy állapot vizsgálatánál figyelembe lehessen venni bizonyos átmenetek végrehajtásával elérhető állapotok tulajdonságait.

Példa. HML formulák szemantikus jelentése:

- Ha $f = \langle a \rangle \underline{1}$, akkor $\mathcal{A}, s \models f \Leftrightarrow \exists t \in T$, hogy $\alpha(t) = s$ és $\lambda(t) = a$, vagyis indul a címkéjű átmenet az s-ből.
- Ha $f = \neg(\bigvee_{a \in A} \langle a \rangle \underline{1})$, akkor $\mathcal{A}, s \models f \Leftrightarrow s$ nem forrása egyetlen átmenetnek sem, vagyis nem indul belőle átmenet.
- Ha $f = \bigvee_{a \in A} \langle a \rangle \underline{1}$, akkor $\mathcal{A}, s \models f \Leftrightarrow s$-ből indul ki átmenet ($s$ nem holtpont).
- Ha $f = [a] f'$, akkor $\mathcal{A}, s \models f \Leftrightarrow \forall t \in T$-re, ahol $\alpha(t) = s$, $\lambda(t) = a$ teljesül, hogy $\mathcal{A}, \beta(t) \models f'$.
- Ha $f = [a]\langle b \rangle f'$, akkor $\mathcal{A}, s \models f \Leftrightarrow \forall t \in T$-re, ahol $\alpha(t) = s$, $\lambda(t) = a$ $\exists t' \in T$, melyre $\alpha(t') = \beta(t)$, $\lambda(t') = b$ és teljesül, hogy $\mathcal{A}, \beta(t') \models f'$.

A HML logika kiterjeszthető címkézett és állapot-paraméteres átmeneti rendszerekre. Ebben az esetben a logikában formulaként megjelenik minden állapot atomi kijelentés is. A HML logika kiterjeszthető úgy is, hogy nem csupán címkézett átmeneti rendszernél, hanem átmenet-paraméteres rendszernél is alkalmazható legyen állapot tulajdonságok megadására. Ekkor a kijelentés logika átmenetformulái a HML átmenetformulái lesznek, az ott megadott szemantika szerint. A HML állapotformulái a következők:

- $\underline{0}, \underline{1}$,
- ha f_1 és f_2 állapotformulák, akkor $f_1 \wedge f_2$, $\neg f_1$ állapotformulák,
- ha g átmenetformula és f állapotformula, akkor $\langle g \rangle f$ állapotformula.

Tetszőleges $s \in S$, f állapotformula és g átmenetformula esetén az $\langle g \rangle f$ szemantikája:

$$\mathcal{A}, s \models \langle g \rangle f \Leftrightarrow \exists t \in T, \text{hogy } \alpha(t) = s,\ \mathcal{A}, t \models g \text{ és } \mathcal{A}, \beta(t) \models f.$$

A $[g] f = \neg\langle g \rangle \neg f$ gyakran alkalmazott rövidítés szemantikája:

$$\mathcal{A}, s \models [g] f \Leftrightarrow \forall t \in T, \text{melyre } \alpha(t) = s,\ \mathcal{A}, t \models g,\ \text{teljesül, hogy } \mathcal{A}, \beta(t) \models f.$$

3.4. Dicky logika

A Dicky logika formuláival $(\mathcal{X}, \mathcal{Y})$-paraméteres átmeneti rendszerek tulajdonságai írhatók le. A logika szimbólumai:

- $\underline{0}_\sigma, \underline{1}_\sigma, \underline{0}_\tau, \underline{1}_\tau$ állapot, illetve átmenet logikai konstansok,
- P_x állapot atomi kijelentés, minden $x \in \mathcal{X}$-re,
- Q_y átmenet atomi kijelentés, minden $y \in \mathcal{Y}$-ra,

- kétváltozós állapot, illetve átmenet operátorok: $\wedge_\sigma, \vee_\sigma, -_\sigma, \wedge_\tau, \vee_\tau, -_\tau$,
- egyváltozós állapot, illetve átmenet operátorok: src, tgt, in, out.

(A σ jelzés állapot, míg a τ jelzés átmenet operátort jelöl.)
A logika formulái:
állapotformulák:

- $\underline{0}_\sigma, \underline{1}_\sigma, P_x$ minden $x \in \mathcal{X}$-re,
- ha f_1, f_2 állapotformulák, akkor $f_1 \vee_\sigma f_2$, $f_1 \wedge_\sigma f_2$, $f_1 -_\sigma f_2$ állapotformulák,
- ha g egy átmenetformula, akkor $\mathrm{src}(g), \mathrm{tgt}(g)$ állapotformulák.

átmenetformulák:

- $\underline{0}_\tau, \underline{1}_\tau, Q_y$ minden $y \in \mathcal{Y}$-ra,
- ha g_1, g_2 átmenetformulák, akkor $g_1 \vee_\tau g_2$, $g_1 \wedge_\tau g_2$, $g_1 -_\tau g_2$ átmenetformulák,
- ha f állapotformula, akkor $\mathrm{in}(f), \mathrm{out}(f)$ átmenetformulák.

Legyen $\mathcal{A} = (S, T, \alpha, \beta, S_{x_1}, \ldots, S_{x_n}, T_{y_1}, \ldots, T_{y_m})$ egy $(\mathcal{X}, \mathcal{Y})$-paraméteres átmeneti rendszer $(\mathcal{X} = \{x_1, \ldots, x_n\}, \mathcal{Y} = \{y_1, \ldots, y_m\})$, $s \in S$, $t \in T$, f állapotformula, g átmenetformula. Az $\mathcal{A}, s \models f$, illetve $\mathcal{A}, t \models g$ relációk definíciója az f, illetve g formulák felépítése szerinti indukcióval a következő:

- a logikai konstansok, az atomi kijelentések és a kijelentés logikában is szereplő kétváltozós operátorok esetén, mint a kijelentés logikánál,
- ha $f = f_1 -_\sigma f_2$, akkor $\mathcal{A}, s \models f \Leftrightarrow \mathcal{A}, s \models f_1$ és $\mathcal{A}, s \not\models f_2$,
- ha $f = \mathrm{src}(g)$, akkor $\mathcal{A}, s \models f \Leftrightarrow \exists t \in T$, hogy $\alpha(t) = s$ és $\mathcal{A}, t \models g$,
- ha $f = \mathrm{tgt}(g)$, akkor $\mathcal{A}, s \models f \Leftrightarrow \exists t \in T$, hogy $\beta(t) = s$ és $\mathcal{A}, t \models g$,
- ha $g = g_1 -_\tau g_2$, akkor $\mathcal{A}, t \models g \Leftrightarrow \mathcal{A}, t \models g_1$ és $\mathcal{A}, t \not\models g_2$,
- ha $g = \mathrm{in}(f)$, akkor $\mathcal{A}, t \models g \Leftrightarrow \mathcal{A}, \beta(t) \models f$,
- ha $g = \mathrm{out}(f)$, akkor $\mathcal{A}, t \models g \Leftrightarrow \mathcal{A}, \alpha(t) \models f$.

A Dicky logika nem használja a logikai negációt($\neg$), viszont ez kifejezhető a logika $-$ műveleteivel:

- ha f állapotformula, akkor

$$\mathcal{A}, s \models \neg f \Leftrightarrow \mathcal{A}, s \models \underline{1}_\sigma -_\sigma f,$$

- ha g átmenetformula, akkor

$$\mathcal{A}, t \models \neg g \Leftrightarrow \mathcal{A}, t \models \underline{1}_\tau -_\tau g.$$

A HML logika beágyazható a Dicky logikába, mivel a HML logika $\langle a \rangle f$ formulája kifejezhető a Dicky logikában a $\mathrm{src}(Q_a \wedge_\tau \mathrm{in}(f))$ formulával, vagyis

$$\mathcal{A}, s \models \langle a \rangle f \Leftrightarrow \mathcal{A}, s \models \mathrm{src}(Q_a \wedge_\tau \mathrm{in}(f)).$$

Példa. Az alábbiakban néhány Dicky formula szemantikus jelentését mutatjuk be:

- Ha $f = \mathrm{src}(\underline{1}_\tau)$, akkor $\mathcal{A}, s \models f \Leftrightarrow$ ha az s állapotból indul ki átmenet.
- Ha $f = \underline{1}_\sigma -_\sigma \mathrm{tgt}(\underline{1}_\tau)$, akkor $\mathcal{A}, s \models f \Leftrightarrow$ ha az s állapotba nem vezet átmenet.
- Ha $f = P_{x_1} \wedge_\sigma \mathrm{src}(\underline{1}_\tau \wedge_\tau \mathrm{in}(P_{x_2}))$, akkor $\mathcal{A}, s \models f \Leftrightarrow$ ha $s \in S_{x_1}$ és van olyan $t \in T$ átmenet, hogy $\beta(t) \in S_{x_2}$.
- Ha $g = \mathrm{out}(P_x)$, akkor $\mathcal{A}, t \models g \Leftrightarrow \alpha(t) \in S_x$.
- Ha $g = \mathrm{in}(P_x \wedge_\sigma \mathrm{src}(Q_y))$, akkor $\mathcal{A}, t \models g \Leftrightarrow \beta(t) \in S_x$ és $\beta(t)$-ből indul olyan $t' \in T$ átmenet, melyre $t' \in T_y$.

3.5. CTL (Computation Tree Logic)

Legyen $\mathcal{A} = (S, T, \alpha, \beta, S_{x_1}, \ldots, S_{x_n})$ olyan $(\mathcal{X}, \emptyset)$-paraméteres átmeneti rendszer ($\mathcal{X} = \{x_1, \ldots, x_n\}$), melyben minden állapotnak van rákövetkezője. Az $\mathcal{A}$ átmeneti rendszer egy s_0 állapotához tartozó *számítási fa* akkor egy olyan végtelen, irányított fa, melynek csúcsai az S állapothalmaz elemeivel címkézettek úgy, hogy a gyökércsúcs címkéje s_0 és valahányszor egy csúcs címkéje $s \in S$ és az s-ből T-beli átmenetekkel elérhető állapotok $s_1,\ldots,s_k$, akkor a csúcsnak k darab leszármazottja lesz, melynek címkéi $s_1,\ldots,s_k$.

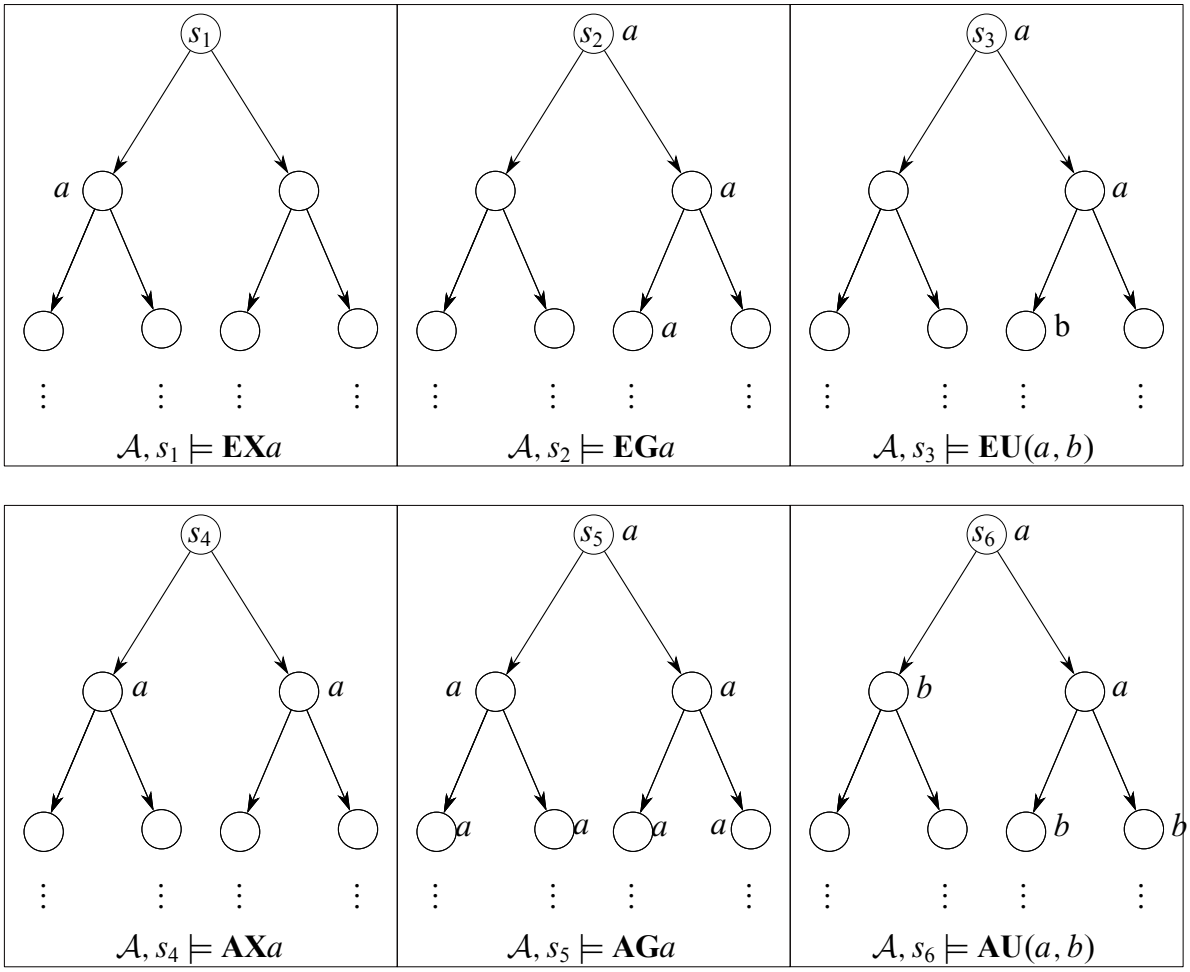
A CTL rövidítése a Computation Tree Logic angol elnevezésnek, ami magyarul számítási fa logikának fordítható. A logika neve arra utal, hogy a CTL formuláival kifejezhetők az állapotokhoz tartozó számítási fák elágazó tulajdonságai.

Jelölje $AP = \{P_x \mid x \in \mathcal{X}\}$ az állapot atomi kijelentések halmazát.
A logika szimbólumai:

- az AP feletti kijelentés logika szimbólumai,
- **EX**, **AX** egyváltozós operátorok,
- **EU**, **AU** kétváltozós operátorok.

A logika *állapotformulái*:

- $\underline{0}, \underline{1}$,
- a, minden $a \in AP$ esetén,
- ha f_1 és f_2 állapotformulák, akkor $f_1 \wedge f_2$ állapotformula,
- ha f állapotformula, akkor $\neg f$ állapotformula,



3.3. ábra. Példák LTL formulákat kielégítő egy-egy útra

- ha f állapotformula, akkor $\mathbf{EX}f$ és $\mathbf{AX}f$ állapotformulák,
- ha f_1 és f_2 állapotformulák, akkor $\mathbf{EU}(f_1, f_2)$ és $\mathbf{AU}(f_1, f_2)$ állapotformulák.

Jelöljön $\mathcal{A} = (S, T, \alpha, \beta, S_{x_1}, \ldots, S_{x_n})$ olyan $(\mathcal{X}, \emptyset)$-paraméteres átmeneti rendszert ($\mathcal{X} = \{x_1, \ldots, x_n\}$), melyben minden állapotnak van rákövetkezője. Legyen $s \in S$ tetszőleges állapot és f egy AP feletti CTL formula. Az $\mathcal{A}, s \models f$ kielégítési reláció definíciója az f felépítése szerinti indukcióval a következő:

- ha $f = \underline{0}$, akkor $\mathcal{A}, s \not\models f$,
- ha $f = \underline{1}$, akkor $\mathcal{A}, s \models f$,
- ha $f = a$, akkor $\mathcal{A}, s \models f \Leftrightarrow s \in S_x$, ahol $a = P_x \in AP$ valamely $x \in \mathcal{X}$-re,
- ha $f = f_1 \wedge f_2$, akkor $\mathcal{A}, s \models f \Leftrightarrow \mathcal{A}, s \models f_1$ és $\mathcal{A}, s \models f_2$,
- ha $f = \neg f'$, akkor $\mathcal{A}, s \models f \Leftrightarrow \mathcal{A}, s \not\models f'$,

- ha $f = \mathbf{EX}f'$, akkor $\mathcal{A}, s \models f \Leftrightarrow \exists t \in T$, hogy $\alpha(t) = s$ és $\mathcal{A}, \beta(t) \models f'$,
- ha $f = \mathbf{AX}f'$, akkor $\mathcal{A}, s \models f \Leftrightarrow \forall t \in T$-re, ha $\alpha(t) = s$ akkor $\mathcal{A}, \beta(t) \models f'$,
- ha $f = \mathbf{EU}(f_1, f_2)$, akkor $\mathcal{A}, s \models f \Leftrightarrow \exists c = s_0, s_1, \ldots$ végtelen út, ahol $s = s_0$ és $\exists j \geq 0$, hogy $\mathcal{A}, s_j \models f_2$ és $\forall 0 \leq i < j$-re $\mathcal{A}, s_i \models f_1$,
- ha $f = \mathbf{AU}(f_1, f_2)$, akkor $\mathcal{A}, s \models f \Leftrightarrow \forall c = s_0, s_1, \ldots$ végtelen útra, ahol $s = s_0$ teljesül, hogy $\exists j \geq 0$, hogy $\mathcal{A}, s_j \models f_2$ és $\forall 0 \leq i < j$-re $\mathcal{A}, s_i \models f_1$.

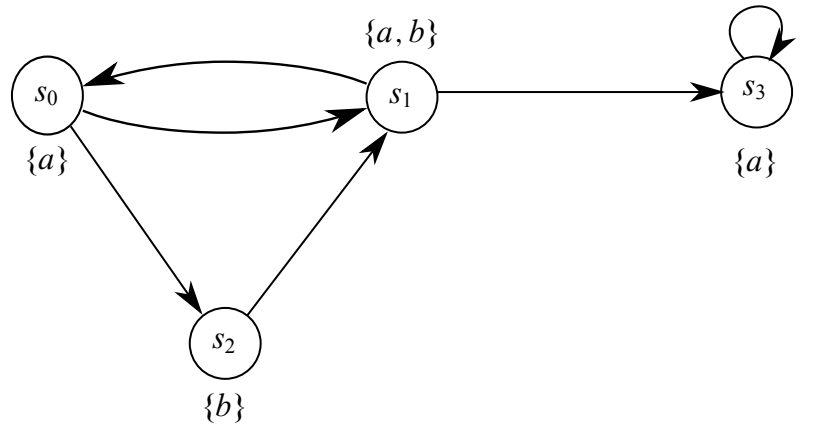
Az eddig bevezetett operátorokkal kifejezhető további gyakran alkalmazott operátorok:

$$\mathbf{EF}f = \mathbf{E}(\underline{1}\mathbf{U}f), \quad \mathbf{AF}f = \mathbf{A}(\underline{1}\mathbf{U}f),$$
$$\mathbf{EG}f = \neg\mathbf{AF}\neg f, \quad \mathbf{AG}f = \neg\mathbf{EF}\neg f.$$

A 3.3. ábrán egy olyan átmeneti rendszer számítási fái láthatók, melyek gyökércsúcsa által reprezentált állapot az $\mathbf{EX}a$, $\mathbf{EG}a$, $\mathbf{EU}(a,b)$, $\mathbf{AX}a$, $\mathbf{AG}a$, $\mathbf{AU}(a,b)$ formulákat elégítik ki, ahol $AP = \{a, b\}$. A számítási fákban a csúcsoknál a reprezentált állapot tulajdonságai a csúcs mellett szerepelnek.

Legyen f egy tetszőleges AP feletti CTL formula. Jelölje $S_f = \{s \in S \mid \mathcal{A}, s \models f\}$ az f formulát kielégítő állapotok halmazát. Legyen $S_0 \subseteq S$ az $\mathcal{A}$ átmeneti rendszer kezdőállapotainak halmaza. Azt mondjuk, hogy az $\mathcal{A}$ *kielégíti az* f *formulát*, amit az $\mathcal{A} \models f$ jelöl, ha $S_f \supseteq S_0$.

Példa. A 3.4. ábrán látható $\mathcal{A}$ átmeneti rendszer esetén az alábbi f CTL formulákra az S_f halmazok a következők:



3.4. ábra. $\mathcal{A}$ *átmeneti rendszer, ahol* $AP = \{a, b\}$

- $f = \mathbf{EX}a$: $S_f = \{s_0, s_1, s_2, s_3\}$,
- $f = \mathbf{AX}a$: $S_f = \{s_1, s_2, s_3\}$,
- $f = \mathbf{EG}a$: $S_f = \{s_0, s_1, s_3\}$,

- $f = \mathbf{AG}a$: $S_f = \{s_3\}$,
- $f = \mathbf{EF}(\mathbf{EG}a)$: $S_f = \{s_0, s_1, s_2, s_3\}$,
- $f = \mathbf{AU}(a, b)$: $S_f = \{s_0, s_1, s_2\}$,
- $f = \mathbf{EU}(a, (\neg a \wedge \mathbf{AU}(\neg a, b)))$: $S_f = \{s_0, s_1, s_2\}$.

Animált ábra. A 3. animált ábrán adott átmeneti rendszernél a formulákat kielégítő állapotok kiválasztására és a megoldás helyességének ellenőrzésére van lehetőség.

Legyenek f_1 és f_2 AP feletti CTL formulák. Azt mondjuk, hogy f_1 *ekvivalens* az f_2-vel, amit $f_1 \equiv f_2$ jelöl, ha minden AP feletti $\mathcal{A}$ átmeneti rendszerre teljesül, hogy $S_{f_1} = S_{f_2}$.

A logika operátorainak egy H halmazát *adekvátnak* nevezzük, ha a logika bármely f formulájához megadható olyan vele ekvivalens f' formula, melyben szereplő operátorok mindegyike H-nak eleme.

Néhány azonosság, amelyek alkalmazásával belátható, hogy az {**EX**, **EG**, **EU**} a CTL-ben használható temporális operátoroknak egy adekvát halmaza:

$$\mathbf{AX}f \equiv \neg\mathbf{EX}(\neg f), \quad \mathbf{EF}f \equiv \mathbf{EU}(\underline{1}, f),$$
$$\mathbf{AF}f \equiv \neg\mathbf{EG}(\neg f), \quad \mathbf{AG}f \equiv \neg\mathbf{EF}(\neg f) \equiv \neg\mathbf{EU}(\underline{1}, \neg f),$$
$$\mathbf{AU}(f_1, f_2) \equiv \neg\mathbf{EU}(\neg f_2, (\neg f_1 \wedge \neg f_2)) \wedge \neg\mathbf{EG}(\neg f_2).$$

Példa. Legyen $AP = \{k_1, k_2, ö, i, kr, t\}$, ahol az AP elemei az alábbi tulajdonságokat jelölik az átmeneti rendszerben:

- k_i: az i. folyamat a kritikus szekciókjában van, ahol $i = 1, 2$.
- $ö$: a rendszer összeomlott.
- i: a rendszer kezdőállapotban van.
- kr: kérés érkezett.
- t: kérés teljesítve.

A CTL alkalmazásával kifejezhetők például a következő tulajdonságok egy s állapotra vonatkozóan:

- $\mathbf{EF}\,k_1$: s-ből elérhető olyan állapot, melyben az 1. folyamat a kritikus szekciójában van.
- $\neg\mathbf{EF}\,ö$: s-ből nem érhető el a rendszer összeomlása.
- $\mathbf{AG}(\mathbf{EF}\,i)$): minden s-ből elérhető állapotból elérhető valamely kezdőállapot.
- $\mathbf{AG}\,\neg(k_1 \wedge k_2)$: az s-ből elérhető minden állapot olyan, hogy az 1. és a 2. folyamat nincs egyszerre a kritikus szekciójában.
- $\mathbf{AG}(kr \rightarrow \mathbf{AF}\,t)$: az s-ből elérhető minden állapot olyan, hogy ha az elért állapotban érkezett kérés, akkor ebből az állapotból mindig lesz olyan állapot, melyben a kérés teljesített.

3.6. CTL* logika

A CTL* logika a CTL kiterjesztése azzal, hogy a lineáris temporális operátorok előtt közvetlenül nem követeli meg útkvantor, vagyis **E** vagy **A** megjelenését.

Legyen $\mathcal{X}=\{x_1,\ldots,x_n\}$ állapotparaméterek halmaza, $AP=\{P_x \mid x\in\mathcal{X}\}$ atomi kijelentések halmaza.
A logika szimbólumai:

- az AP feletti kijelentés logika szimbólumai,
- **E** (egzisztenciális) útkvantor,
- **X**, **U** lineáris temporális operátorok.

A CTL* logikában vannak állapotformulák és útformulák.
A logika *állapotformulái*:

- $\underline{1}$,
- a, minden $a \in AP$-re,
- ha f állapotformula, akkor $\neg f$ állapotformula,
- ha f_1 és f_2 állapotformulák, akkor $f_1 \wedge f_2$ állapotformula,
- ha g útformula, akkor $\mathbf{E}g$ állapotformula.

A logika *útformulái*:

- ha f egy állapotformula, akkor f útformula,
- ha g útformula, akkor $\neg g$ útformula,
- ha g_1 és g_2 útformulák, akkor $g_1 \wedge g_2$ útformula,
- ha g útformula, akkor $\mathbf{X}g$ útformula,
- ha g_1 és g_2 útformulák, akkor $\mathbf{U}(g_1, g_2)$ útformula.

Legyen $\mathcal{A} = (S, T, \alpha, \beta, S_{x_1}, \ldots, S_{x_n})$ egy $(\mathcal{X}, \emptyset)$-paraméteres átmeneti rendszer ($\mathcal{X} = \{x_1, \ldots, x_n\}$), melynek minden állapotára van rákövetkező állapot. Tetszőleges $s \in S$ állapot és f állapotformula esetén definiáljuk, hogy az $\mathcal{A}$ átmeneti rendszer az s állapotában mikor elégíti ki az f állapotformulát, amit $\mathcal{A}, s \models f$ jelöl. Az $\mathcal{A}, s \models f$ reláció definíciója az f felépítése szerinti indukcióval a következő:

- ha $f = \underline{1}$, akkor $\mathcal{A}, s \models f$,
- ha $f = a$, ahol $a = P_x \in AP$, akkor $\mathcal{A}, s \models f \Leftrightarrow s \in S_x$,
- ha $f = \neg f'$, akkor $\mathcal{A}, s \models f \Leftrightarrow \mathcal{A}, s \not\models f'$,

- ha $f = f_1 \wedge f_2$, akkor $\mathcal{A}, s \models f \Leftrightarrow \mathcal{A}, s \models f_1$ és $\mathcal{A}, s \models f_2$,
- ha $f = \mathbf{E}g$, akkor $\mathcal{A}, s \models f \Leftrightarrow \exists c = s_0, s_1, \ldots$ végtelen út, ahol $s = s_0$ és $\mathcal{A}, c \models g$.

Tetszőleges $c = s_0, s_1, \ldots$ végtelen út és g útformula esetén definiáljuk, hogy az $\mathcal{A}$ átmeneti rendszer egy c útja mikor elégíti ki az g útformulát, amit $\mathcal{A}, c \models g$ jelöl. Az $\mathcal{A}, c \models g$ reláció definíciója az g felépítése szerinti indukcióval a következő:

- ha $g = f$, ahol f állapotformula, akkor $\mathcal{A}, c \models g \Leftrightarrow \mathcal{A}, s_0 \models f$,
- ha $g = \neg g'$, akkor $\mathcal{A}, c \models g \Leftrightarrow \mathcal{A}, c \not\models g'$,
- ha $g = g_1 \wedge g_2$, akkor $\mathcal{A}, c \models g \Leftrightarrow \mathcal{A}, c \models g_1$ és $\mathcal{A}, c \models g_2$,
- ha $g = \mathbf{X}g'$, akkor $\mathcal{A}, c \models g \Leftrightarrow \mathcal{A}, c^1 \models g'$,
- ha $g = \mathbf{U}(g_1, g_2)$, akkor $\mathcal{A}, c \models g \Leftrightarrow \exists j \geq 0$, hogy $\mathcal{A}, c^j \models g_2$, és bármely $0 \leq i < j$-re $\mathcal{A}, c^i \models g_1$.

Legyen f tetszőleges állapotformula CTL*-ban. Jelölje az f-et kielégítő $\mathcal{A}$-beli állapotok halmazát $S_f = \{s \in S \mid \mathcal{A}, s \models f\}$.

Legyen $S_0 \subseteq S$ az $\mathcal{A}$ kezdőállapotainak a halmaza. Azt mondjuk, hogy $\mathcal{A}$ *kielégíti az* f *állapotformulát*, ennek jelölése $\mathcal{A} \models f$, ha $S_0 \subseteq S_f$.

Ahogyan az LTL-nél és a CTL-nél, itt is használhatók a bevezetett operátorokkal kifejezhető további operátorok. Itt az **A** univerzális útkvantor definiálható az **E** egzisztenciális útkvantorral, vagyis $\mathbf{A}g = \neg\mathbf{E}\neg g$ tetszőleges g útformulára.

Például szintaktikusan helyes az $f = \mathbf{EG}\,\mathbf{F}g$ CTL* formula, ahol g egy útformula. Az f-et olyan átmeneti rendszerek elégítik ki, melyeknek van valamely kezdőállapotból induló olyan útja, melyen a g tulajdonság végtelen sokszor teljesül. Látható, hogy f nem CTL formula.

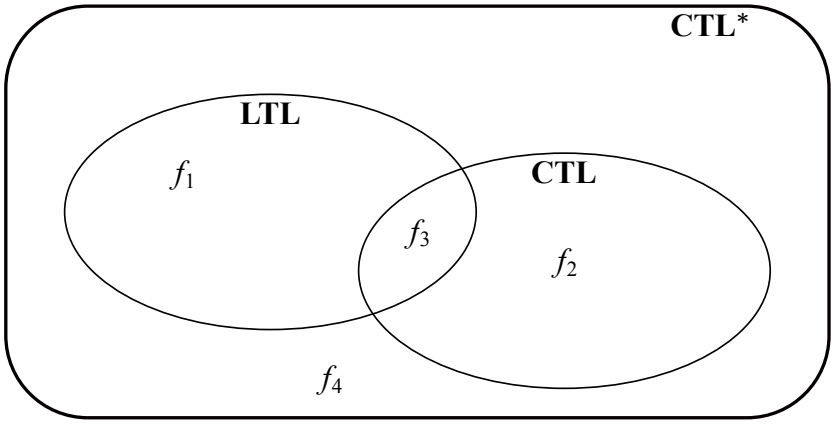
3.7. LTL, CTL és CTL* logikák kifejezőerejének összehasonlítása

Egy logika kifejezőerején a logikában kifejezhető tulajdonságok összességét értjük.

Jelölje AP az atomi kijelentések halmazát és f, g tetszőleges AP feletti LTL útformulát, CTL vagy CTL*-beli állapotformulát. Azt mondjuk, hogy f és g *ekvivalensek*, amit $f \equiv g$ jelöl, ha AP feletti bármely $\mathcal{A}$ átmeneti rendszerre és annak tetszőleges s állapotára teljesül, hogy $\mathcal{A}, s \models f \Leftrightarrow \mathcal{A}, s \models g$.

A 3.5. ábra szemlélteti az LTL, CTL és CTL* logikák kifejezőereje szerinti kapcsolatát adott AP atomi kijelentések felett.

Az LTL és a CTL logikák kifejezőereje nem hasonlítható össze, mivel létezik olyan LTL formula, amellyel nincs ekvivalens CTL formula, illetve létezik olyan CTL formula, amellyel nincs ekvivalens LTL formula. Például, ha $a \in AP$, akkor az alábbi f_1 LTL formulához nincs vele ekvivalens CTL formula, illetve az alábbi f_2 CTL formulához nincs vele ekvivalens LTL formula:



3.5. ábra. Az LTL, CTL és CTL logikák közötti kapcsolat*

$$f_1 = \mathbf{FG}a, \quad f_2 = \mathbf{AG}(\mathbf{EF}a).$$

Léteznek az LTL-ben és a CTL-ben is egyaránt kifejezhető tulajdonságok, pl. $f_3 = \mathbf{GF}a$, ami egy LTL formula és ekvivalens az $\mathbf{AG}(\mathbf{AF}a)$ CTL formulával.

Mivel tetszőleges f LTL formulával az $\mathbf{A}f$ CTL* formula ekvivalens, az LTL a CTL*-ba beágyazható. Továbbá, minden CTL formula egyben CTL* formula, így az LTL-ben és a CTL-ben kifejezhető tulajdonságok CTL*-ban is kifejezhetők. Másrészt a CTL* kifejezőereje nagyobb, mint az LTL és a CTL logikáké. Például az $f_4 = \mathbf{AFG}a \vee \mathbf{AG}(\mathbf{EF}a)$ CTL* formulával nincs ekvivalens LTL, illetve CTL formula.

3.8. TCTL logika (Timed Computation Tree Logic)

A TCTL logika a CTL logika valós idejű változata, mellyel időzített automaták tulajdonságai fejezhetők ki.

Jelölje C az órák halmazát és $\mathcal{AB}(C)$ a C feletti *atomi órafeltételek halmazát*, vagyis azoknak az órafeltételeknek a halmazát, melyekben az $\wedge$ (és) operátor nem fordul elő. Jelölje a $v : C \to \mathbb{R}_{\geq 0}$ leképezések halmazát $V(C)$.

A TCTL *állapotformulái* az AP atomi kijelentések és a C órahalmaz felett:

- $\underline{1}$,
- a, minden $a \in AP$-re,
- g, minden $g \in \mathcal{AB}(C)$-re,
- ha f_1 és f_2 állapotformulák, akkor $f_1 \wedge f_2$ állapotformula,
- ha f állapotformula, akkor $\neg f$ állapotformula,
- ha f_1 és f_2 állapotformulák, akkor $\mathbf{EU}_J(f_1, f_2)$ és $\mathbf{AU}_J(f_1, f_2)$ állapotformulák, ahol $J \subseteq \mathbb{R}_{\geq 0}$ természetes szám korlátú intervallum.

A J tehát $[n,m]$, $(n,m]$, $[n,m)$, (n,m) alakú, ahol $n,m \in \mathbb{N}$ és $n \leq m$, illetve megengedett a jobbról nem korlátos intervallum ($m = \infty$).

A TCTL kiterjeszti a CTL logikát egyrészt azzal, hogy megenged órafeltételeket formulaként, másrészt úgy, hogy a CTL-ben lévő **EU** és **AU** operátorok helyett a $J \subseteq \mathbb{R}_{\geq 0}$ természetes szám korlátú intervallumokhoz kapcsolódóan az $\mathbf{EU}_J$ és $\mathbf{AU}_J$ operátorok jelennek meg a logikában. A formulák szemantikája olyan lesz, hogy a $J = [0,\infty)$ nem korlátos intervallumhoz tartozó $\mathbf{EU}_J$ és $\mathbf{AU}_J$ szemantikája a CTL-ben levő **EU**, illetve **AU** operátorokéval megegyezik.

A CTL-nél alkalmazható **EF**, **AF**, **EG**, **AG** operátoroknak az időzített változata a TCTL-ben a következő:

$$\mathbf{EF}_J f = \mathbf{EU}_J(\underline{1}, f), \quad \mathbf{EG}_J f = \neg\mathbf{AF}_J \neg f,$$
$$\mathbf{AF}_J f = \mathbf{AU}_J(\underline{1}, f), \quad \mathbf{AG}_J f = \neg\mathbf{EF}_J \neg f.$$

A CTL **X** (next) operátora a TCTL-ben nem jelenik meg, mivel itt az időtartomány folytonos.

Ebben a fejezetben megengedjük, hogy egy C órahalmaz és A akcióhalmaz feletti időzített automatának több kezdőhelye is legyen. A kezdőhelyek halmazát L_0-lal jelölünk. Feltesszük, hogy az A akcióhalmaz véges, és az $\mathcal{A} = (L, L_0, E, I)$ időzített automata helyei egy $\rho : L \to \to \mathcal{P}(AP)$ címkézőfüggvény szerint címkézettek, ahol AP az atomi kijelentések véges halmaza. Az ilyen időzített automatát ezután $\mathcal{A} = (L, A, C, L_0, E, I, AP, \rho)$ jelöli.

Az $\mathcal{A}$ időzített automata $T(\mathcal{A})$ átmeneti rendszerében legyen

$$\pi = q_0 \xrightarrow{\tau_0} q_1 \xrightarrow{\tau_1} q_2 \to \dots$$

egy tetszőleges végtelen átmenet sorozat (út), ahol $\tau_i \in A \cup \mathbb{R}_{>0}$. Vezessük be a következő jelöléseket:

$$\mathrm{ext}(\pi) = \sum_{i=0}^{\infty} \mathrm{ext}(\tau_i), \text{ ahol}$$

$$\mathrm{ext}(\tau) = \begin{cases} 0 & \text{ha } \tau \in A; \\ d & \text{ha } \tau = \varepsilon(d),\ d \in \mathbb{R}_{>0}. \end{cases}$$

Egy π végtelen átmenet sorozatot *idő-divergensnek* nevezzünk, ha $\mathrm{ext}(\pi) = \infty$, egyébként *idő-konvergensnek* nevezzük.

Az időzített automaták vizsgálatának fókuszában az idő-divergens utak állnak, vagyis azok az utak, melyekben az idő minden határon túl nő.

Jelölés. Legyen

$$\pi = q_0 \xrightarrow{\varepsilon(d_0^1)} \dots \xrightarrow{\varepsilon(d_0^{k_0})} q_0 + d_0 \xrightarrow{a_0} q_1 \xrightarrow{\varepsilon(d_1^1)} \dots \xrightarrow{\varepsilon(d_1^{k_1})} q_1 + d_1 \xrightarrow{a_1} q_2 \to \dots$$

egy $T(\mathcal{A})$-beli végtelen sok akciós átmenetet tartalmazó idő-divergens átmenet sorozat, ahol $k_i \in \mathbb{N}$, $d_i \in \mathbb{R}_{\geq 0}$, $a_i \in A$ és $\sum_{j=1}^{k_i} d_i^j = d_i$ minden $i \geq 0$-ra. Mivel nem érdekes a vizsgálat szempontjából, hogy q_i-ből a q_{i+1} konfigurációt milyen várakozási átmenetek sorozatával éri el, ezért a π utat a

$$q_0 \overset{d_0}{\Longrightarrow} q_1 \overset{d_1}{\Longrightarrow} q_2 \overset{d_2}{\Longrightarrow} \dots$$

sorozattal is jelöljük.

Ha pedig π véges sok akciós átmenetet tartalmazó idő-divergens átmenet sorozat, pontosabban

$$\pi = q_0 \xrightarrow{\varepsilon(d_0^1)} \ldots \xrightarrow{\varepsilon(d_0^{k_0})} q_0 + d_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{n-2}} q_{n-1} \xrightarrow{\varepsilon(d_{n-1}^1)} \ldots \xrightarrow{\varepsilon(d_{n-1}^{k_{n-1}})}$$
$$\xrightarrow{\varepsilon(d_{n-1}^{k_{n-1}})} q_{n-1} + d_{n-1} \xrightarrow{a_{n-1}} q_n \xrightarrow{\varepsilon(1)} q_{n+1} \xrightarrow{\varepsilon(1)} \ldots,$$

akkor π-t a

$$q_0 \overset{d_0}{\Longrightarrow} q_1 \overset{d_1}{\Longrightarrow} \ldots \overset{d_{n-1}}{\Longrightarrow} q_n \overset{1}{\Longrightarrow} q_{n+1} \overset{1}{\Longrightarrow} q_{n+2} \overset{1}{\Longrightarrow} \ldots$$

sorozattal jelöljük.

$T(\mathcal{A})$-ban egy q konfigurációt *elérhetőnek* nevezünk, ha van valamely kezdőkonfigurációból induló véges átmenetsorozat, ami q-ba vezet.

Egy $\mathcal{A}$ időzített automatát *idő-divergensnek* nevezünk, ha minden $T(\mathcal{A})$-beli elért konfiguráció olyan, hogy van belőle induló idő-divergens átmenet sorozat.

A továbbiakban feltesszük, hogy az automata idő-divergens. Az idő-konvergens utak egyébként azért nem játszanak szerepet a vizsgálatokban, mert nem felelnek meg a valóságnak, vagyis annak, hogy az idő minden határon túl nő egy folyamatosan működő rendszernél, illetve véges idő alatt csak véges sok akció végrehajtása történhet meg.

Legyen $q = \langle l, v \rangle$ egy tetszőleges konfiguráció $T(\mathcal{A})$-ban, és legyen $d \in \mathbb{R}_{\geq 0}$. Ekkor jelölje $q + d$ a $\langle l, v + d \rangle$ konfigurációt.

A TCTL formulák szemantikája:

Legyen $\mathcal{A} = (L, A, C, L_0, E, I, AP, \rho)$ egy véges időzített automata, $a \in AP$, $g \in \mathcal{AB}(C)$, $J \subseteq \mathbb{R}_{\geq 0}$. Tetszőleges $q = \langle l, v \rangle$ $T(\mathcal{A})$-beli konfiguráció és f AP és C feletti TCTL formula esetén definiáljuk, hogy a $T(\mathcal{A})$ átmeneti rendszer q konfigurációja mikor elégíti ki az f formulát, amit a $T(\mathcal{A}), q \models f$ jelöl. A definíció az f felépítése szerinti indukcióval a következő:

- ha $f = \underline{1}$, akkor $T(\mathcal{A}), q \models f$,
- ha $f = a$, ahol $a \in AP$, akkor $T(\mathcal{A}), q \models f \Leftrightarrow a \in \rho(l)$,
- ha $f = g$, ahol $g \in \mathcal{AB}(C)$, akkor $T(\mathcal{A}), q \models f \Leftrightarrow v \models g$,
- ha $f_1 \wedge f_2$, akkor $T(\mathcal{A}), q \models f \Leftrightarrow T(\mathcal{A}), q \models f_1$ és $T(\mathcal{A}), q \models f_2$,
- ha $f = \neg f'$, akkor $T(\mathcal{A}), q \models f \Leftrightarrow T(\mathcal{A}), q \not\models f'$,
- ha $f = \mathbf{EU}_J(f_1, f_2)$, akkor $T(\mathcal{A}), q \models f \Leftrightarrow \exists \pi = q_0 \overset{d_0}{\Longrightarrow} q_1 \overset{d_1}{\Longrightarrow} \ldots$ idő-divergens út, ahol $q_0 = q$, melyre teljesül, hogy van olyan $i \geq 0$, $d \in [0, d_i]$, hogy

$$\left(\sum_{k=0}^{i-1} d_k\right) + d \in J,\ T(\mathcal{A}), q_i + d \models f_2, \tag{3.1}$$

és bármely olyan $j \leq i$ és $d' \in [0, d_j]$ esetén, ahol

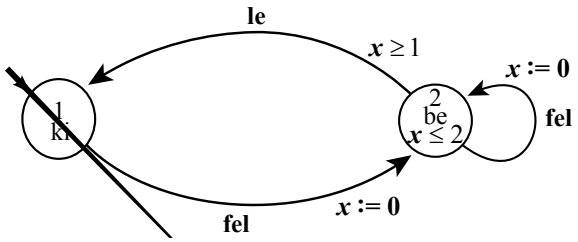
$$\left(\sum_{k=0}^{j-1} d_k\right) + d' < \left(\sum_{k=0}^{i-1} d_k\right) + d \text{ teljesül, hogy } T(\mathcal{A}), q_j + d' \models f_1 \vee f_2,$$
$$\text{ahol } q_i = \langle \ell_i, v_i \rangle \text{ esetén } q_i + d = \langle \ell_i, v_i + d \rangle. \tag{3.2}$$

- ha $f = \mathbf{AU}_J(f_1, f_2)$, akkor $T(\mathcal{A}), q \models f \Leftrightarrow \forall \pi$ q-ból induló idő-divergens útra teljesül (3.1) és (3.2).

Tehát egy $\mathbf{EU}_J(f_1, f_2)$ formulát akkor elégít ki egy $q = \langle l, v \rangle$ konfiguráció, ha a q-ból van olyan idő-divergens számítási út, hogy valamely $t \in J$ időpontra f_2 tulajdonságú konfigurációt ér el, és minden korábbi időpontban $f_1 \vee f_2$ tulajdonságú konfigurációban van. Ha $J = [0, \infty)$, akkor az $\mathbf{EU}_J$ és $\mathbf{AU}_J$ operátorok TCTL szerinti és az $\mathbf{EU}$, illetve $\mathbf{AU}$ operátorok CTL szerinti szemantikája megegyezik.

Legyen f tetszőleges állapotformula TCTL-ben. Jelölje $S_f = \{q \in L \times V(C) \mid T(\mathcal{A}), q \models \models f\}$, vagyis a $T(\mathcal{A})$ összes f-et kielégítő konfigurációk halmazát.

Azt mondjuk, hogy az $\mathcal{A}$ *időzített automata kielégíti az* f *TCTL formulát*, ha $\forall \ell_0 \in L_0$-ra az $\langle \ell_0, v_0 \rangle \in S_f$, ahol $v_0(x) = 0$ minden $x \in C$-re.



3.6. ábra. Egy lámpakapcsoló $\mathcal{A}$ időzített automatája

Példa. A 3.6. ábrán látható $\mathcal{A}$ időzített automatánál néhány f TCTL formulára megadjuk az S_f halmazt:

- $f_1 = \mathbf{AF}_{<1}\text{ki}$, akkor $S_{f_1} = \{\langle 1, t \rangle \mid t \geq 0\} \cup \{\langle 2, t \rangle \mid 1 < t \leq 2\}$,
- $f_2 = \mathbf{EF}_{<1}\text{ki}$, akkor $S_{f_2} = \{\langle 1, t \rangle \mid t \geq 0\} \cup \{\langle 2, t \rangle \mid 0 < t \leq 2\}$,
- $f_3 = \mathbf{AF}(\text{be} \wedge (x = 0))$, akkor $S_{f_3} = \{\langle 2{,}0 \rangle\}$,
- $f_4 = \mathbf{AF}(\text{be} \wedge (x = 1))$, akkor $S_{f_4} = \{\langle 2, t \rangle \mid 0 \leq t \leq 1\}$.

Mivel $\langle 1{,}0 \rangle \in S_{f_1}$ és $\langle 1{,}0 \rangle \in S_{f_2}$, ezért $\mathcal{A} \models f_1$ és $\mathcal{A} \models f_2$.
Legyen $f_5 = \mathbf{AG}((\text{be} \wedge (x = 0)) \rightarrow \mathbf{AF}(\text{be} \wedge (x = 1)))$, akkor $\mathcal{A} \models f_5$, mivel az $\mathbf{A}$ útkvantor csak idő-divergens utakat vizsgál, vagyis pl. $\langle 1{,}0 \rangle$, $\langle 2{,}0 \rangle$, $\langle 2, \frac{1}{2} \rangle$, $\langle 2, \frac{3}{4} \rangle$, ... utat nem vizsgálja, mert ez idő-konvergens.

4. fejezet

A modell-ellenőrzés algoritmusai

4.1. A modell-ellenőrzés alapfeladata

A modell-ellenőrzésnél feltételezzük, hogy a vizsgálandó rendszert véges átmeneti rendszer modellezi és a rendszer vizsgálandó tulajdonsága logikai formulával adott.

A modell-ellenőrzés (model checking) feladata: Adott M véges (vagy végesen megadható) átmeneti rendszer és egy x objektuma (út, állapot vagy átmenet), és adott egy f logikai formula (útformula, állapotformula vagy átmenetformula), melynek típusa megegyezik az x objektum típusával. A lokális modell-ellenőrzés arra keresi a választ, hogy $M, x \models f$ teljesül-e. A globális modell-ellenőrzés célja adott M átmeneti rendszer és f logikai formula esetén meghatározni azon x objektumok halmazát (x és f típusa megegyezik), melyek kielégítik f-et, vagyis az $\{x \mid M, x \models f\}$ halmazt. A globális modell-ellenőrzés eredménye végtelen halmaz is lehet útformula esetén. Ekkor a halmaz egy leírását, pl. egy Büchi-automatát keres a modell-ellenőrzés. A globális modell-ellenőrzés eredményének ismeretében a lokális modell-ellenőrzési kérdések megválaszolhatók.

Ebben a fejezetben a modell-ellenőrzés alábbi technikái jelennek meg:

- Szemantika-alapú megközelítés, ami a logika szemantikai szabályai és a modell alapján induktívan számítja ki a formulát kielégítő objektumok halmazát. Ez a technika elsősorban globális modell-ellenőrzésre alkalmas, elágazó logikák esetén. A jegyzetben a CTL modell-ellenőrzésnél jelenik meg.
- Automata-elméleti alapú megközelítés, ami a logikai formula és a modell alapján Büchi-automatákat konstruál, és az ezek által felismert nyelvek vizsgálatát végzi. Ez a technika elsősorban lokális modell-ellenőrzésre alkalmas, lineáris logikáknál. A jegyzetben az LTL modell-ellenőrzésnél jelenik meg.
- Tabló-módszer alapú megközelítés, ami bizonyítási fát épít fel a formula és a modell alapján. Általában egyszerűbb logikáknál, lokális modell-ellenőrzésnél alkalmazzák. A jegyzetben a Hennessy-Milner logika modell-ellenőrzésénél szerepel.
- TCTL modell-ellenőrzés, ami időzített automatákat verifikál időzített TCTL formulákkal felírt specifikációkra nézve, régió-átmenet rendszerek megkonstruálásának segítségével visszavezetve a feladatot a CTL modell-ellenőrzés problémájára.

A modell-ellenőrzésnél a vizsgálandó tulajdonságot gyakran célszerű vagy elvárás a logika valamely adott adekvát operátor halmazába tartozó operátorok segítségével megadni. Ez nem jelent megszorítást, mert ekvivalens átalakításokkal bármely adott logikához tartozó formula olyan alakra hozható, ami már csak az adekvát operátor halmazba tartozó operátorokat tartalmazza.

A fejezet anyaga az [1], [5], [14], [16], [24], [25] irodalmak alapján készült.

4.2. CTL szemantikai alapú modell-ellenőrzés

Legyen $M = (S, T, \alpha, \beta, S_{x_1}, \ldots, S_{x_n})$ egy $(\mathcal{X}, \emptyset)$-paraméteres ($\mathcal{X} = \{x_1, \ldots, x_n\}$) véges átmeneti rendszer, melynek minden $s \in S$ állapotából indul átmenet, és legyen f egy CTL formula.

Az a feltétel, hogy minden $s \in S$ állapotból indul átmenet, nem jelent megszorítást a modell-ellenőrzésben, mivel bővíthető az átmeneti rendszer úgy, hogy azokból az állapotokból, amelyekből nem indul átmenet, egy-egy átmenet vezessen egy újonnan felvett állapotba, ahová átmenet vezet önmagából. Az új állapot egy új x paraméterre az S_x halmaz egyetlen eleme.

A CTL globális modell-ellenőrzés feladata az $S_f = \{s \in S \mid M, s \models f\}$ halmaz meghatározása.

4.2.1. A CTL modell-ellenőrzés algoritmusa

Bemenet: M átmeneti rendszer, f CTL formula
Kimenet: S_f halmaz
Módszer:

1. Megvizsgáljuk, hogy az f formulában szereplő operátorok mindegyike az $\{\wedge, \neg, \mathbf{EX}, \mathbf{EU}, \mathbf{EG}\}$ operátor halmazba tartozik-e. Ha ez nem teljesül, akkor ekvivalens átalakítások alkalmazásával f-et olyan CTL formulává alakítjuk át, amelyben már csak a megengedett operátorok fordulnak elő.
2. Az f minden közvetlen g részformulájához meghatározzuk az S_g halmazt, majd ezek felhasználásával az S_f halmazt. Az f formula felépítése szerint több esetet különböztetünk meg.

   - Ha $f = \underline{1}$, akkor $S_f = S$.
   - Ha $f = P_x, x \in \mathcal{X}$, akkor $S_f = S_x$.
   - Ha $f = \neg g$, akkor $S_f = S - S_g$.
   - Ha $f = g_1 \wedge g_2$, akkor $S_f = S_{g_1} \cap S_{g_2}$.
   - Ha $f = \mathbf{EX}\, g$ akkor, mivel tetszőleges $s_0 \in S$ állapotra $M, s_0 \models \mathbf{EX}g \Leftrightarrow \exists c = s_0, s_1, \ldots$ végtelen út, hogy $M, s_1 \models g$, ezért

     $$S_f = \{s \in S \mid \exists t \in T, \text{ amelyre } \alpha(t) = s \text{ és } \beta(t) \in S_g\}.$$

   - Ha $f = \mathbf{EU}(g_1, g_2)$, akkor tetszőleges $s_0 \in S$-re $M, s_0 \models \mathbf{EU}(g_1, g_2) \Leftrightarrow \exists c = s_0, s_1, \ldots$ végtelen út, hogy vagy $M, s_0 \models g_2$ vagy $\exists j > 0$, hogy $M, s_j \models g_2$ és minden

$0 \le i < j$-re $M, s_i \models g_1$. Ez utóbbi annak felel meg, hogy $M, s_0 \models g_1$ és $M, s_1 \models$ $\models \mathbf{EU}(g_1, g_2)$. Ezért S_f a legszűkebb olyan halmaz, amely tartalmazza az S_{g_2} halmazt, és amelyre tetszőleges $t \in T$ esetén, ha $\alpha(t) \in S_{g_1}$ és $\beta(t) \in S_f$, akkor $\alpha(t) \in S_f$.
Az S_f halmazt meghatározhatjuk az alábbi véges iterációval:

- $S_0 := S_{g_2}$,
- $S_{i+1} := S_i \cup \{ s \in S \mid s \in S_{g_1}$ és $\exists t \in T$, hogy $\alpha(t) = s$ és $\beta(t) \in S_i \}$, ahol $0 \le i < |S| = n$,
- $S_f := S_n$.

Nyilvánvaló, ha valamely $0 \le i < n$-re $S_{i+1} = S_i$, akkor a fenti iteráció befejezhető, és $S_f = S_i$.

Megjegyzés. Legyen $\tau : \mathcal{P}(S) \to \mathcal{P}(S)$ az alábbi leképezés:
$\tau(z) = S_{g_2} \cup [S_{g_1} \cap \{s \in S \mid \exists t \in T$, hogy $\alpha(t) = s$ és $\beta(t) \in z\}]$. Belátható, hogy S_f a $\tau(z)$ legkisebb fixpontja.

- Ha $f = \mathbf{EG}\, g$, akkor tetszőleges $s_0 \in S$-re $M, s_0 \models \mathbf{EG}g \Leftrightarrow \exists c = s_0, s_1, \ldots$ végtelen út, hogy minden $i \ge 0$-ra $M, s_i \models g$, ami annak felel meg, hogy $M, s_0 \models$ $\models \mathbf{EG}g \Leftrightarrow M, s_0 \models g$ és $M, s_1 \models \mathbf{EG}g$. Ezért S_f az S_g legbővebb olyan részhalmaza, melyre tetszőleges $t \in T$ esetén, ha $\beta(t) \in S_f$, akkor $\alpha(t) \in S_f$. Az S_f halmazt meghatározhatjuk iterációval:

  - $S_0 := S_g$,
  - $S_{i+1} := S_i \cap \{s \in S \mid \exists t \in T$, amelyre $\alpha(t) = s$ és $\beta(t) \in S_i\}$, ahol $0 \le i < |S| = n$,
  - $S_f := S_n$.

  A fenti iteráció befejezhető az első olyan $0 \le i < n$-re, melyre $S_{i+1} = S_i$, és ekkor $S_f = S_i$.

  Megjegyzés. Legyen $\tau : \mathcal{P}(S) \to \mathcal{P}(S)$ leképezés az alábbi:
  $\tau(z) = S_g \cap \{s \in S \mid \exists t \in T$, melyre $\alpha(t) = s$ és $\beta(t) \in z\}$. Belátható, hogy S_f a $\tau(z)$ legnagyobb fixpontja.

Példa. Tekintsük a 4.1. ábrán látható $M = (S, T, \alpha, \beta, S_a, S_b, S_c)$ $(\mathcal{X}, \emptyset)$-paraméteres átmeneti rendszert, ahol $\mathcal{X} = \{a, b, c\}$.

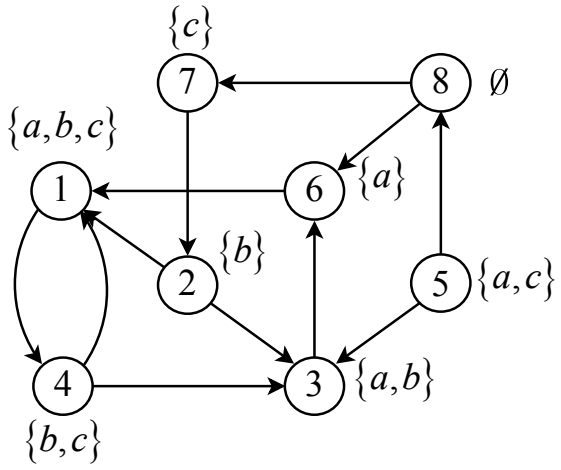
a) Meghatározzuk az $S_{\mathbf{EG}P_b} = \{s \in S \mid M, s \models \mathbf{EG}\, P_b\}$ halmazt az algoritmus szerint.

$S_0 = S_{P_b} = \{1,2,3,4\}$.
$S_1 = S_0 \cap \{s \in S \mid \exists t \in T,\ \alpha(t) = s,\ \beta(t) \in S_0\} = \{1,2,3,4\} \cap \{1,2,4,5,6,7\} = \{1,2,4\}$.
$S_2 = S_1 \cap \{s \in S \mid \exists t \in T,\ \alpha(t) = s,\ \beta(t) \in S_1\} = \{1,2,4\} \cap \{1,2,4,6,7\} = \{1,2,4\}$.

Mivel $S_1 = S_2$ így $S_{\mathbf{EG}P_b} = \{1,2,4\}$.

b) A következő példában alkalmazzuk az ekvivalencia ($\leftrightarrow$) kétváltozós operátort. Ha f és f' CTL formulák, akkor

$$f \leftrightarrow f' \equiv \neg\left[\neg(f \wedge f') \wedge \neg(\neg f \wedge \neg f')\right].$$



4.1. ábra. M átmeneti rendszer az $S_{\mathbf{EG}P_b}$ és $S_{\mathbf{EU}(1,[(P_a \leftrightarrow P_c) \wedge (P_a \leftrightarrow \neg P_b)])}$ kiszámításához

Meghatározzuk az $S_{\mathbf{EU}(1,[(P_a \leftrightarrow P_c) \wedge (P_a \leftrightarrow \neg P_b)])}$ halmazt az algoritmus szerint.

Jelölje $f_1 = P_a \leftrightarrow P_c$, $f_2 = P_a \leftrightarrow \neg P_b$, $f = f_1 \wedge f_2$ és $g = \underline{1}$ formulát.

$S_f = S_{f_1} \cap S_{f_2} = \{1,2,5,8\} \cap \{2,4,5,6\} = \{2,5\}$.

$S_g = S = \{1,2,3,4,5,6,7,8\}$.

$S_0 = S_f = \{2,5\}$.

$S_1 = S_0 \cup \{s \in S \mid \exists t \in T,\ \alpha(t) = s,\ \beta(t) \in S_0,\ s \in S_g\} = \{2,5\} \cup \{7\} = \{2,5,7\}$.

$S_2 = S_1 \cup \{s \in S \mid \exists t \in T,\ \alpha(t) = s,\ \beta(t) \in S_1,\ s \in S_g\} = \{2,5,7\} \cup \{7,8\} = \{2,5,7,8\}$.

$S_3 = S_2 \cup \{s \in S \mid \exists t \in T,\ \alpha(t) = s,\ \beta(t) \in S_2,\ s \in S_g\} = \{2,5,7\} \cup \{5,7,8\} = \{2,5,7,8\}$.

Mivel $S_3 = S_2$ így $S_{\mathbf{EU}(1,f)} = \{2,5,7,8\}$.

4.3. CTL modell-ellenőrzési algoritmus megvalósítása

Legyen $M = (S, T, \alpha, \beta, S_{x_1}, \ldots, S_{x_n})$ egy $(\mathcal{X}, \emptyset)$-paraméteres ($\mathcal{X} = \{x_1, \ldots, x_n\}$), véges átmeneti rendszer, melynek minden állapotára van rákövetkező állapot. Legyen f olyan CTL formula, amelynek operátorai a $\{\wedge, \neg, \mathbf{EX}, \mathbf{EU}, \mathbf{EG}\}$ halmazba tartoznak (vagyis a modell-ellenőrzés 1. lépésére nem térünk ki).

A modell-ellenőrzést megvalósító címkéző eljárást pszeudo-kóddal adjuk meg. Minden $s \in S$ állapothoz és f minden g részformulájához egy $s.g$ változó tartozik, melynek értékét az eljárás *true* értékre állítja, ha $M, s \models g$, különben az értéke *false* lesz. Minden $s \in S$-re az $s.nb$ az M átmeneti rendszerben az s-ből induló átmenetek számát tartalmazza.

A főeljárás neve: *címkéző(p)*, ahol a p input paraméternek a vizsgálandó CTL formula adható meg.

A címkéző eljárás befejeztével az M átmeneti rendszer és az f CTL formulára vonatkozó globális modell-ellenőrzésre a választ az $\{s \in S \mid s.f = true\}$ halmaz adja meg.

A címkéző eljárás a megengedett alkalmazható operátorok és a logikai konstans, illetve atomi kijelentés alapján 7 esetre bomlik, amelyek az alábbiak:

case$(p = \underline{1})$:
    forall $s \in S$ do
        $s.p := true$
    endforall

case$(p = P_x)$: /* ahol $x \in \mathcal{X}$ */
    forall $s \in S$ do
        if $s \in S_x$ then $s.p := true$
                else $s.p := false$;
    endforall

case$(p = \neg g)$:
    címkéző(g);
    forall $s \in S$ do
        $s.p :=$ not$(s.g)$;
    endforall

case$(p = g_1 \wedge g_2)$:
    címkéző(g_1);
    címkéző(g_2);
    forall $s \in S$ do
        $s.p :=$ and$(s.g_1, s.g_2)$;
    endforall

case$(p = \mathbf{EX}\ g)$:
    címkéző(g);
    forall $s \in S$ do
        $s.p := false$;
    endforall;
    forall $t \in T$ do
        if $\beta(t).g = true$ then $\alpha(t).p := true$;
    endforall

case$(p = \mathbf{EU}(g_1, g_2))$:
    címkéző(g_1);
    címkéző(g_2);
    forall $s \in S$ do
        $s.p := false$;
        $s.segéd := false$;
    endforall
    $V := \emptyset$;
    forall $s \in S$ do
        if $s.g_2 = true$ then
        begin
            $V := V \cup \{s\}$;
            $s.segéd := true$;

```
        end
    endforall
    while V ≠ ∅ do begin
        V := V − {s};    /* kivesz V-ből egy s állapotot */
        s.p := true;
        forall t ∈ T do
            if (β(t) = s) and (α(t).segéd = false) and (α(t).g1 = true) then
            begin
                α(t).segéd := true;
                V := V ∪ {α(t)};
            end
        endforall
    end
```

case$(p = \mathbf{EG}(g_1))$:

```
    címkéző(g1);
    V := ∅;
    forall s ∈ S do
        if s.g1 = false then V := V ∪ {s}
                        else s.p := true;
    endforall
    while V ≠ ∅ do begin
        V := V − {s};    /* kivesz V-ből egy s állapotot */
        s.p := false;
        forall t ∈ T do
            if (β(t) = s) and (α(t).p = true) then α(t).nb := α(t).nb − 1;
            if α(t).nb = 0 then V := V ∪ {α(t)}
        endforall
    end
```

Megjegyzés. Az eljárás p=**EG**(g_1) esetben az első ciklusban az S_p iterációs kiszámítás szerint minden $s \in S_0$ állapotnál $s.p$-t $true$ értékre és az $\overline{S_0}$-be tartozó állapotoknál $false$ értékre állítja. A második ciklus i-dik lefutása után minden $s \in \overline{S_i}$ állapotnál $s.p$ változó értéke $false$.

Az $S_{\mathbf{EG}g_1}$ halmazt a

$$\tau'(z) = S_{\neg g_1} \cup \{s \in S \mid \forall t \in T, \text{ ha } \alpha(t) = s, \text{ akkor } \beta(t) \in z\}$$

leképezés legkisebb fixpontjának komplementereként is megkaphatjuk.

A címkéző eljárás időigénye:

- Az eljárás biztosan terminál, mivel a while ciklusokat tartalmazó esetekben a V halmazba minden állapot legfeljebb egyszer kerülhet be.
- Az első 3 esetben az időigény $\mathcal{O}(|S|)$.
- A 4. esetben az időigény az inicializálás $\mathcal{O}(|S|)$ időigénye és az összes átmenet egyszeri vizsgálatának $\mathcal{O}(|T|)$ időigénye.

- Az 5. és 6. esetben az időigény az inicializálások $\mathcal{O}(|S|)$ időigénye és a while ciklus időigénye, amelyben minden átmenetet legfeljebb egyszer vizsgál, melynek időigénye $\mathcal{O}(|T|)$.

- Az eljárás teljes időigénye $\mathcal{O}(|f| \cdot (|S| + |T|))$. Ha az $|S| + |T|$-t az M átmeneti rendszer méretének, $|M|$-nek tekintjük, akkor a címkéző eljárás a bemeneti f CTL formula méretében és a bemeneti M átmeneti rendszer méretében lineáris.

4.4. Állapotrobbanás

A modell-ellenőrzésnek az egyik bemeneti adata a vizsgálandó rendszert modellező átmeneti rendszer. A CTL modell-ellenőrzésre adott eljárás ugyan lineáris az átmeneti rendszer méretében, de a gyakorlati feladatoknál ez a méret nagyon nagy.

Ha például tekintünk egy olyan M rendszert, amely $M_1, \ldots, M_n$ átmeneti rendszerek szinkronizált szorzata, akkor az M legrosszabb esetben a komponens átmeneti rendszerek méreteinek sorozata, vagyis $|M_1| \cdot \ldots \cdot |M_n|$. Feltéve, hogy mindegyik komponensnek 2 állapota van, az M állapotainak száma 2^n lehet, vagyis a komponensek számában exponenciális méretű.

Annak érdekében, hogy a CTL modell-ellenőrzés a gyakorlatban is használható legyen, érdemes az átmeneti rendszer valamilyen gazdaságosabb reprezentációját választani. Ilyen reprezentálásra ad lehetőséget a redukált rendezett bináris döntési diagramok (ROBDD) használata. Az ROBDD-kel támogatott modell-ellenőrzéseket szimbolikus modell-ellenőrzésnek nevezzük.

4.5. Halmaz reprezentálása ROBDD-vel

Definíció. *Bináris döntési fának* (BDF) nevezünk egy teljesen kiegyensúlyozott, véges bináris fát, melynek minden nem levél csúcsa szintenként azonos változóval címkézett, ezekből a csúcsokból induló egyik él 0-val a másik 1-gyel címkézett, és a levél csúcsok (terminális) 0-val vagy 1-gyel címkézettek. A fában a különböző szinteken előforduló változó címkék különböznek.

Az n-változós Boole-függvények és az n változót tartalmazó BDF-ek bijektív módon megfeleltethetők egymásnak, az alábbiak szerint: Legyen t egy BDF, amelyben a csúcsok címkéi a gyökér csúcsával kezdve szintenként $x_1, \ldots, x_n$. Feleltessük meg t-nek azt a $\varphi(x_1, \ldots, x_n)$ Boole-függvényt, amely igazságtáblájának a sorai t-ben a gyökérből induló utaknak felelnek meg úgy, hogy ha x_i-ből az út 0 címkéjű élen vezet, akkor az igazságtábla sorában x_i értéke 0, ha az út az 1 címkéjű élen vezet, akkor az x_i értéke a táblázat sorában 1, és ha az út végén a levél címkéje 0, akkor az igazságtábla sorában a φ értéke 0, egyébként 1.

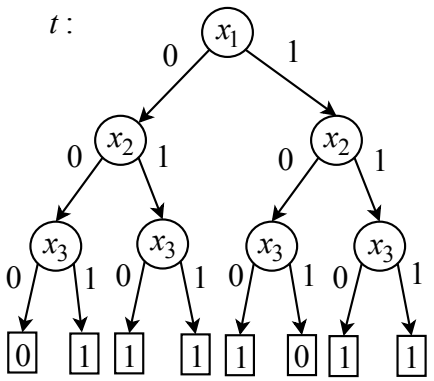
Legyen H egy olyan véges halmaz, amelynek elemeit n-hosszúságú bit vektorral kódoljuk. Azt mondjuk, hogy a H halmazt a $\varphi(x_1, \ldots, x_n)$ Boole-függvényt reprezentálja, ha minden $(u_1, \ldots, u_n) \in \{0,1\}^n$ elemre teljesül, hogy $\varphi(u_1, \ldots, u_n) = 1$ akkor és csak akkor, ha $(u_1, \ldots, u_n) \in H$. Így ha a $\varphi(x_1, \ldots, x_n)$ Boole-függvénynek a t BDF felel meg, akkor t a H halmazt is reprezentálja.

x_1	x_2	x_3	φ
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

4.1. táblázat. A φ Boole-függvény igazságtáblája

Egy t BDF által reprezentált halmaz elemeinek kódjai a gyökérből az 1 értékű levelekhez vezető utakhoz rendelt igazságtábla sorokban a változók értékeiből álló vektor.

Jelölje a t BDF által reprezentált halmazt H_t.



4.2. ábra. Példa BDF-re

Példa. A 4.2 ábrán látható t az $\{x_1, x_2, x_3\}$ változó halmaz felett egy BDF. A t-ben a belső csúcsokat ◯, míg a leveleket □ jelöli. A t által reprezentált halmaz H_t={(0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,1,0), (1,1,1)}. A t-nek megfeleltetett $\varphi(x_1, x_2, x_3)$ Boole-függvényt igazságtábláját a 4.1 táblázat adja meg.

Definíció. *Bináris döntési diagram* (BDD) változók egy $X = \{x_1, \ldots, x_n\}$ halmaza felett egy gyökércsúccsal rendelkező, ciklus-mentes, véges irányított gráf, melynek belső csúcsaiból egy 0-val és egy 1-gyel címkézett él indul, a gyökércsúcsból minden csúcs elérhető, a belső csúcsok címkéje X-beli változó, a termináló (levél) csúcsok címkéje 0 vagy 1, és a gyökérből induló termináló csúcsig vezető utakon minden X-beli címke legfeljebb egyszer fordul elő.

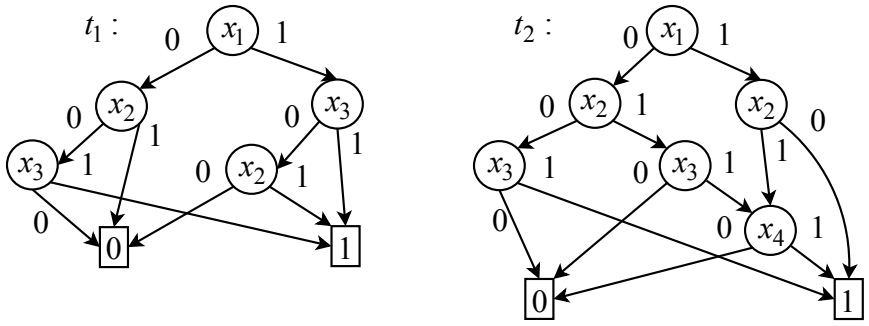
Minden BDF egyben BDD is.

Legyen t az $X = \{x_1, \ldots, x_n\}$ halmaz feletti BDD. Jelölje $[t]$ a t gyökércsúcsát, t_0, t_1 azt a rész BDD-t, ami a $[t]$ csúcsból induló 0, illetve 1 címkéjű éllel kapcsolódik a gyökércsúcshoz, ha $[t]$ belső csúcs. Továbbá t-beli tetszőleges n csúcsnál jelölje $c(n)$ az n csúcs címkéjét.

Ekkor a t BDD egy $\varphi_t(x_1, \ldots, x_n)$ Boole-függvényt reprezentál, amit a következőképpen definiálunk:

- Ha $[t]$ levél csúcs, akkor $\varphi_t(x_1,\ldots,x_n) = c([t])$.
- Ha $[t]$ belső csúcs, és $c([t]) = x_i$, valamely $1 \le i \le n$ esetén, akkor $\varphi_t(x_1,\ldots,x_n) = (\neg x_i \wedge \varphi_{t_0}(x_1,\ldots,x_n)) \vee (x_i \wedge \varphi_{t_1}(x_1,\ldots,x_n))$.

Definíció. Legyen t_1 és t_2 egy-egy BDD. A t_1 *ekvivalens* a t_2-vel ($t_1 \equiv t_2$) akkor és csak akkor, ha t_1 és t_2 ugyanazt a Boole-függvényt reprezentálják.
Definíció. *Rendezett bináris döntési diagramnak* (OBDD) nevezzük azokat a BDD-ket, melyekhez megadható a változóknak olyan lineáris rendezése, hogy minden gyökértől levélig vezető úton a változók egymásra következése ennek a rendezésnek nem mond ellent.



4.3. ábra. Példa nem-rendezett és rendezett BDD-re

Példa. A 4.3 ábrán lévő t_1 BDD nem rendezett mivel van olyan út gyökértől levélig, amelyen a változók egymásra következése x_1, x_2, x_3 és van olyan is amelyen x_1, x_3, x_2. A t_2 BDD minden gyökérből induló levélig vezető útján a változók egymásra következése eleget tesz az $x_1 < x_2 < x_3 < x_4$ rendezésnek vagyis t_2 OBDD.
Definíció. *Redukált bináris döntési diagramnak* (RBDD) nevezzük azokat a BDD-ket, amelyek nem tartalmaznak izomorf rész BDD-ket és olyan csúcsot, melynek mind a két kimenő éle ugyanabba a csúcsba vezet.

Egy BDD-hez vele ekvivalens RBDD-t megkonstruáló *redukáló algoritmus:*
Bemenet: t BDD.
Kimenet: t' RBDD, amelyre teljesül, hogy $t \equiv t'$.
Módszer:

1. Ha t-ben van 1 címkéjű csúcs, akkor egyet kiválasztunk, a többi 1 címkéjű csúcsba vezető élet átirányítjuk a kiválasztott csúcsba és a kiválasztott 1 címkéjű csúcson kívüli többi 1 címkéjű csúcsot töröljük.
   Hasonló módon járunk el, ha t-ben 0 címkéjű csúcsból több is van.

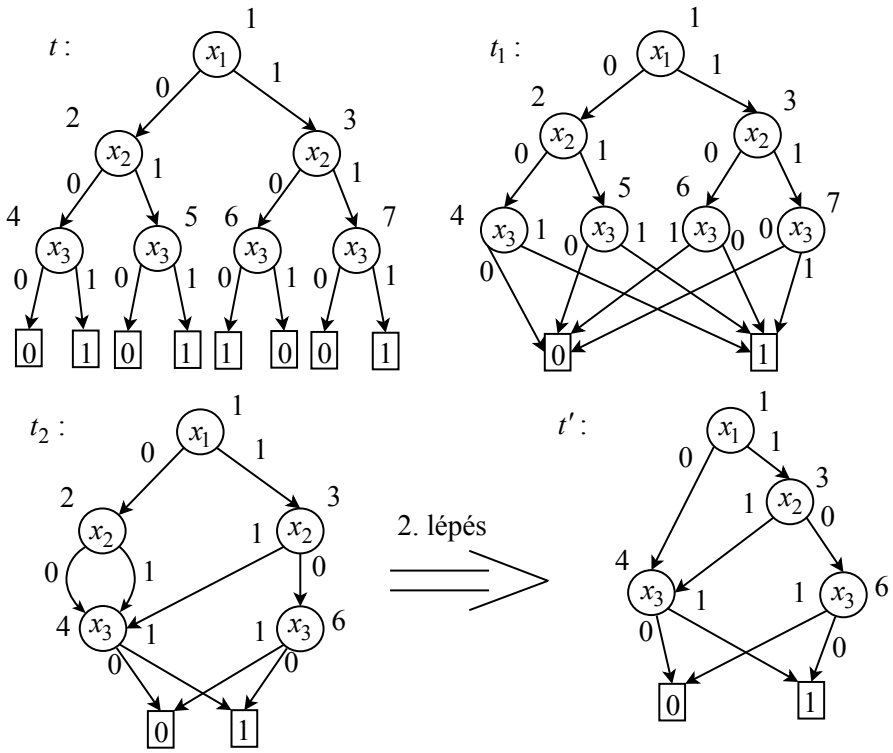
2. Ha egy x belső csúcsból induló élek ugyanabba az y csúcsba vezetnek, akkor az x-be vezető éleket az y csúcsba irányítjuk át és töröljük az x csúcsot.

3. Ha x és y belső csúcsok, x y, $c(x) = c(y)$, az x-ből és az y-ból induló 0 címkéjű élek ugyanabba a csúcsba vezetnek, és az x-ből és y-ból induló 1 címkéjű élek ugyanabba a csúcsba vezetnek, akkor, mondjuk, az y csúcsba vezető éleket átirányítjuk az x csúcsba, és az y csúcsot töröljük a belőle induló 0 és 1 címkéjű éllel együtt.

Az algoritmus lépéseit addig ismétli, amíg a redukálási lépések már változatlanul hagyják a BDD-t.

Belátható, hogy a redukáló algoritmus véges lépésben befejeződik és a bemeneti BDD-vel ekvivalens RBDD-t konstruál meg.

Definíció. *Redukált rendezett bináris döntési diagramnak* (ROBDD) nevezzük azokat az RBDD-ket, melyek rendezettek.



4.4. ábra. A redukáló algoritmus alkalmazása a t BDD-re

Példa. A 4.4 ábrán adott t BDF-re alkalmazzuk a redukáló algoritmus 1. lépését, így kapjuk a t_1 BDD-t. A t_1-ben a 4, 5 és 7 csúcsokra az algoritmus 3. lépésének általánosítását alkalmazzuk. A 4-es csúcsot tartjuk meg, az 5 és 7 csúcsokba mutató éleket a 4-es csúcsba irányítjuk át, ill. az 5 és 7 csúcsokat a belőlük induló élekkel együtt töröljük, melynek eredményeként kialakul a t_2 BDD. A t_2-ben a 2-es csúcsból induló mindkét él ugyanabba a csúcsba vezet. Alkalmazzuk a t_2 2-es csúcsára az algoritmus 2. lépését, így kapjuk a t' BDD-t, ami már redukált.

Megjegyzések.

- A bináris döntési fák rendezettek. A redukáló algoritmus a változók kiindulási rendezését megőrzi, vagyis az előző példában eredményül kapott t' ROBDD.
- A Boole-függvényeknek az ROBDD-vel való reprezentálása lehet nagyon tömör. A reprezentáció tömörsége függ a változók sorrendjétől.
- Tegyük fel, hogy f és g ugyanarra a változó sorrendre rendezett ROBDD-k, az f a φ,

a g a ψ Boole-függvényeket reprezentálják. Belátható, hogy $\varphi = \psi$ akkor és csak akkor, ha f és g izomorfak.

4.6. Műveletek ROBDD-k felett

i) Halmaz ürességének vizsgálata:
Egy t ROBDD-vel reprezentált H_t halmaz akkor és csak akkor üres, ha a t egyetlen 0 címkéjű csúcsból áll.

ii) Halmazok azonosságának vizsgálata:
Legyen f és g két azonos változó sorrend szerint rendezett ROBDD. $H_f = H_g$ akkor és csak akkor, ha f izomorf g-vel.

iii) Halmaz komplementerét reprezentáló ROBDD konstruálása:
Legyen t egy ROBDD. A H_t komplementerét a $\overline{H_t}$ halmazt reprezentáló $\bar{t}$ ROBDD megadható úgy, hogy a t-ben ha van 0 címkéjű levél csúcs, akkor ennek címkéjét 1-re, illetve ha van 1 címkéjű levél csúcs, akkor ennek címkéjét 0-ra változtatjuk. Belátható, hogy ha t a φ Boole-függvényt reprezentáló ROBDD, akkor a $\bar{t}$ a $\neg\varphi$ Boole-függvényt reprezentáló ROBDD.

iv) Halmazok metszetét reprezentáló ROBDD konstruálása:
Legyen f és g két, azonos változó sorrend szerint rendezett ROBDD. A $H_f \cap H_g$ halmazt reprezentáló, az f és g-nél alkalmazott változó sorrendet megőrző t ROBDD ekkor az f és g ismeretében megkonstruálható.

Metszet eljárás:
Bemenet: f, g azonos változó rendezettségű ROBDD-k.
Kimenet: t OBDD, ami a $H_f \cap H_g$ halmazt reprezentálja.
Módszer:

- Ha $c([f])$ és $c([g])$ egyaránt változók, akkor
  - ha $c([f]) = c([g])$, akkor
    $c([t]) := c([f])$,
    $t_0 :=$ metszet(f_0, g_0),
    $t_1 :=$ metszet(f_1, g_1).
  - $c([f]) < c([g])$, akkor
    $c([t]) := c([f])$,
    $t_0 :=$ metszet(f_0, g),
    $t_1 :=$ metszet(f_1, g).
  - ha $c([f]) > c([g])$, akkor
    $c([t]) := c([g])$,
    $t_0 :=$ metszet(f, g_0),
    $t_1 :=$ metszet(f, g_1).
- Ha $c([f]) = 0$, akkor $t := f$.

- Ha $c([g]) = 0$, akkor $t := g$.
- Ha $c([f]) = 1$, akkor $t := g$.
- Ha $c([g]) = 1$, akkor $t := f$.

A $H_f \cap H_g$ halmazt reprezentáló ROBDD-t a redukál(metszet(f, g)) eljárás konstruálja meg.

Belátható, hogy ha f a φ Boole-függvényt reprezentáló ROBDD és g a ψ Boole-függvényt reprezentáló ROBDD, akkor a redukál(metszet(f, g)) eljárás kimenete a $\varphi \wedge \psi$ függvényt reprezentáló ROBDD.

v) *Unió eljárás:*
Bemenet: f, g azonos változó rendezettségű ROBDD-k.
Kimenet: t OBDD, ami a $H_f \cup H_g$ halmazt reprezentálja.
Módszer:

- Ha $c([f])$ és $c([g])$ egyaránt változók, akkor
  - ha $c([f]) = c([g])$, akkor
    $c([t]) := c([f])$,
    $t_0 :=$ unió(f_0, g_0),
    $t_1 :=$ unió(f_1, g_1).
  - ha $c([f]) < c([g])$, akkor
    $c([t]) := c([f])$,
    $t_0 :=$ unió(f_0, g),
    $t_1 :=$ unió(f_1, g).
  - ha $c([f]) > c([g])$, akkor
    $c([t]) := c([g])$,
    $t_0 :=$ unió(f, g_0),
    $t_1 :=$ unió(f, g_1).
- Ha $c([f]) = 0$, akkor $t := g$.
- Ha $c([g]) = 0$, akkor $t := f$.
- Ha $c([f]) = 1$, akkor $t := f$.
- Ha $c([g]) = 1$, akkor $t := g$.

A $H_f \cup H_g$ halmazt reprezentáló ROBDD-t a redukál(unió(f, g)) eljárás konstruálja meg.

Belátható, hogy ha f a φ, g pedig a ψ Boole-függvényt reprezentáló ROBDD, akkor a redukál(unió(f, g)) eljárás kimenete a $\varphi \vee \psi$ Boole-függvényt reprezentáló ROBDD.

vi) Legyen t a φ n-változós Boole-függvényt reprezentáló ROBDD. Jelölje $t[x_i = k]$ a $\varphi(x_1, \ldots, x_{i-1}, k, x_{i+1}, \ldots, x_n)$ Boole-függvényt reprezentáló ROBDD-t, ahol $1 \leq i \leq n$ és $k \in \in \{0,1\}$.
Projekció eljárás:
Bemenet: t ROBDD, x változó, $k \in \{0,1\}$.
Kimenet: $t[x = k]$ ROBDD.
Módszer:

a) A t minden olyan n csúcsából induló élet, ami x-el címkézett valamely n' csúcsba vezet, átirányítjuk az n'-ből k címkéjű éllel kapcsolt leszármazott csúcsba.

b) Az a) lépésben megkonstruált OBDD-re végül a redukál eljárást végrehajtjuk.

vii) Egzisztenciális absztrakció

Legyen t egy $\varphi(x_1, \ldots, x_n)$ Boole-függvényt reprezentáló ROBDD. Jelölje $\exists x_i\, t$ a

$$\exists x_i \varphi = \varphi(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \vee \varphi(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$

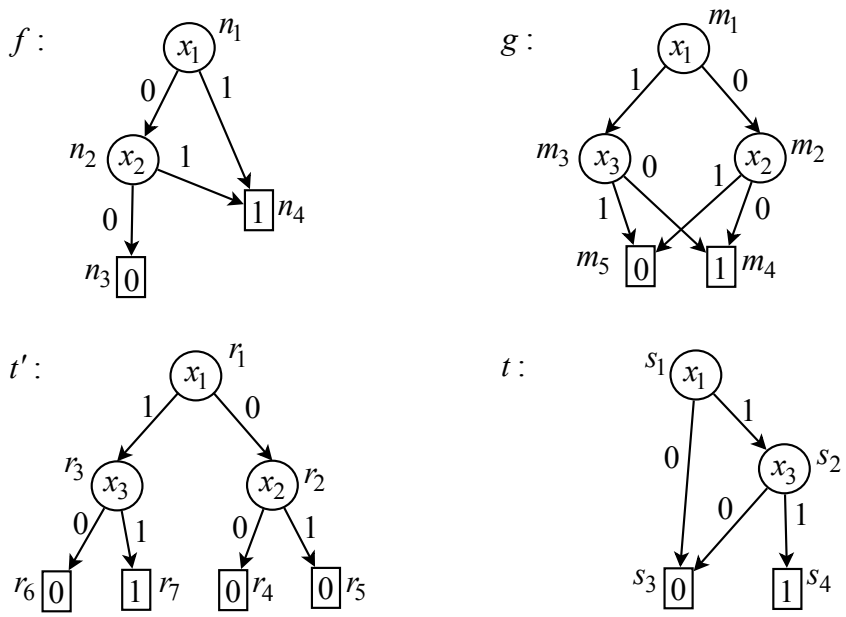
Boole-függvényt reprezentáló ROBDD-t.

Egzisztenciális absztrakció eljárás:

Bemenet: t ROBDD, x változó.

Kimenet: $\exists x t$ ROBDD.

Módszer: $\exists x t :=$ redukál(unió($t[x=0], t[x=1]$)).



4.5. ábra. Az f és g ROBDD-k metszetének megkonstruálása

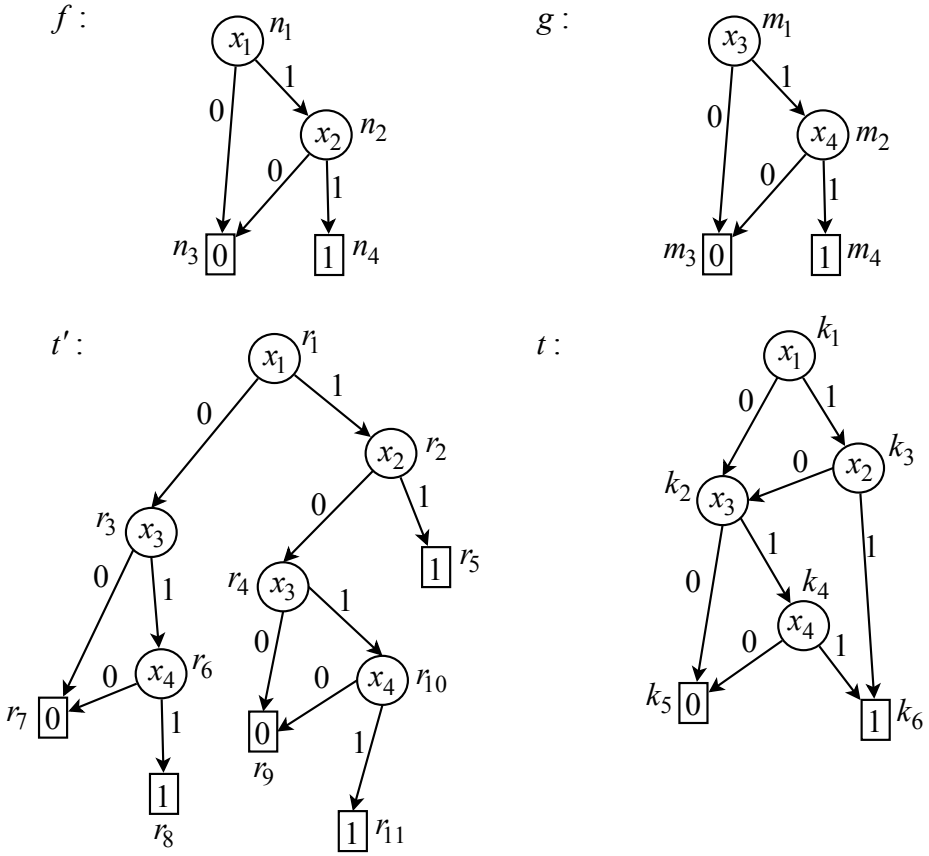
Példa. Egy-egy példán megmutatjuk az ROBDD-ken a metszet, az unió, a projekció és az egzisztenciális absztrakció műveleteket.

A 4.5. ábrán az $x_1 < x_2 < x_3$ változó rendezésnél az f és g ROBDD-k metszetének konstruálása látható. A t' OBDD a metszet(f, g) eljárás végrehajtásával jön létre, majd a t' OBDD-re alkalmazva a redukáló eljárást kapjuk a t ROBDD-t.

A 4.6. ábrán az $x_1 < x_2 < x_3 < x_4$ változó rendezésnél az f és g ROBDD-k uniójának konstruálása szerepel. A t' OBDD az unió(f, g) eredménye, majd a t' OBDD-ből a redukáló algoritmus állítja elő a t ROBDD-t.

A 4.7. ábra az $x_1 < x_2 < x_3 < x_4$ változó rendezés mellett az $f[x_2 = 1]$, $f[x_2 = 0]$ és a $\exists x_2 f$ ROBDD-ket szemlélteti.

Animált ábra. A 4. animált ábrán 5 példa esetén követhetők nyomon azok a lépések, melyekkel az algoritmus az ROBDD-k metszetét konstruálja meg.



4.6. ábra. Az f és g ROBDD-k uniójának megkonstruálása

4.7. Szimbolikus CTL modell-ellenőrzés

A szimbolikus modell-ellenőrzés a modellben szereplő halmazok, relációk explicit tárolása helyett az ezeket reprezentáló Boole-függvényeket tárolja, és a modell-ellenőrzés algoritmusában szereplő halmazműveleteket is Boole-függvények alkalmazásával végzi el.

Jelentősen kevesebb helyet igényelhet a Boole-függvények alkalmasan megválasztott változó rendezés melletti ROBDD-kel való tárolása, mint a reprezentált halmaz közvetlen tárolása.

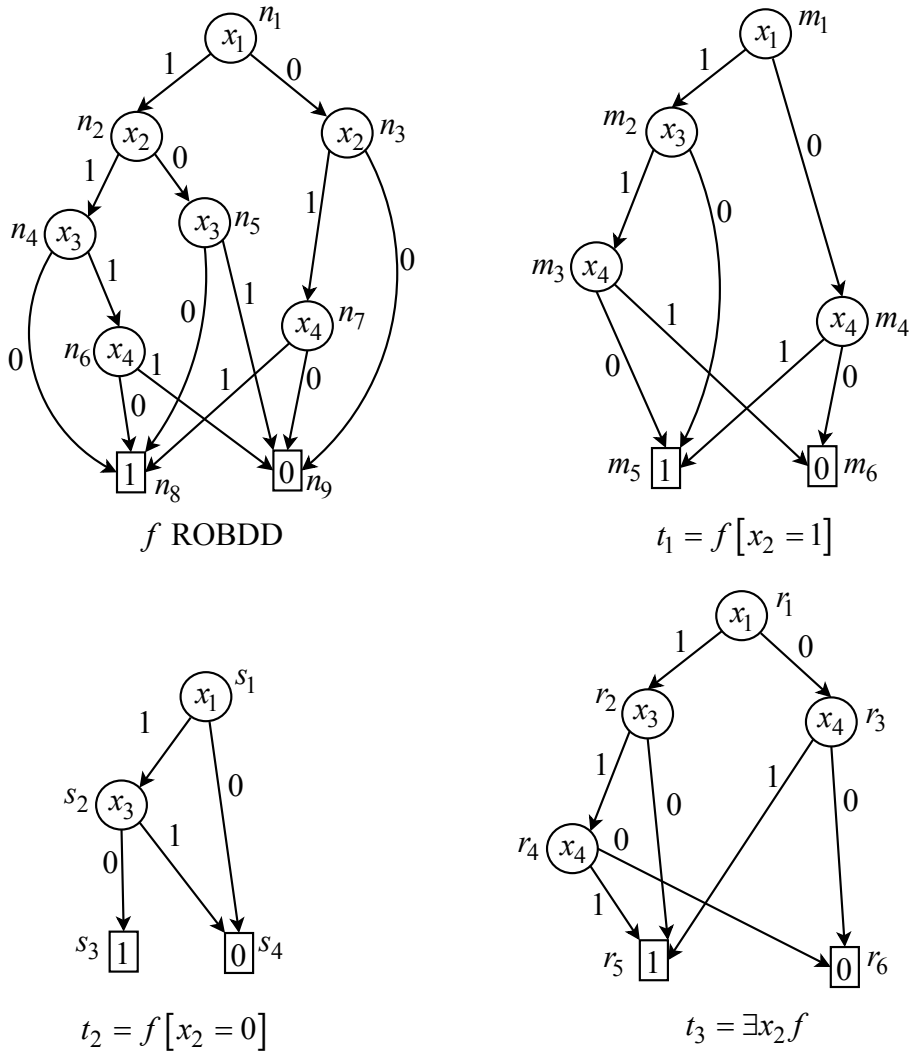
A modell-ellenőrzés algoritmusában levő halmazműveletek a Boole-függvények ROBDD-kel való reprezentálása esetén hatékonyan végezhetők el. Feltétel, hogy a Boole-függvényeket reprezentáló ROBDD-kben a változók rendezése azonos.

Jelölések:

$\bar{x}$: $x_1 \dots x_k$ változók sorozata, illetve $\bar{x}' = x'_1 \dots x'_k$ változók sorozata.

φ' : a φ Boole-függvényben az $x_1, \dots, x_k$ változók $x'_1, \dots, x'_k$-ra való átnevezésével előálló Boole-függvény.

D' : a D ROBDD-ben az $x_1, \dots, x_k$ változók $x'_1, \dots, x'_k$-re való átnevezésével előálló ROBDD.



4.7. ábra. Az f ROBDD-n x_2 szerinti projekció és egzisztenciális absztrakció alkalmazása

$[\varphi]$: a φ Boole-függvény által reprezentált halmaz.

φ_T : a T halmazt reprezentáló Boole-függvény.

D_T : a T halmazt reprezentáló ROBDD.

$D_{[g]}$: az S_g halmazt reprezentáló ROBDD, ahol g CTL formula.

Legyen $M = (S, T, \alpha, \beta, S_{x_1}, \ldots, S_{x_n})$ egy $(\mathcal{X}, \emptyset)$-paraméteres, véges átmeneti rendszer ($\mathcal{X} = \{x_1, \ldots, x_n\}$), melynek minden $s \in S$ állapotából indul átmenet. Tegyük fel, hogy S elemei már bit vektorokkal kódoltak, melyek hossza $k = \lceil log_2(|S|) \rceil$.

Legyen $\varphi_S(x_1, \ldots, x_k)$ az S halmazt reprezentáló Boole-függvény. Az M átmeneti rendszer átmeneteit leíró $R = \{(\alpha(t), \beta(t)) \mid t \in T\} \subseteq S \times S$ halmazba tartozó elemek kódja $2k$-bites lesz, ahol az első k bit az $\alpha(t)$ kódja, a további k bit a $\beta(t)$ kódja, minden $t \in T$-re. Az R halmazt reprezentáló Boole-függvény a $\varphi_R(x_1, \ldots, x_k, x'_1, \ldots, x'_k)$. Az $S_{x_i} \subseteq S$ halmazt a

$\varphi_{S_{x_i}}(x_1, \ldots, x_n)$ Boole-függvény reprezentálja, ahol $1 \leq i \leq n$. A Boole-függvényeket reprezentáló ROBDD-k az $x_1 < x_2 < \cdots < x_k < x'_1 < \cdots < x'_k$ változó rendezés szerintiek.

A következő algoritmus a 4.2.1 fejezetben leírt algoritmus megvalósítása ROBDD műveletekkel.

A *szimbolikus CTL modell-ellenőrzés algoritmusa:*

Bemenet: M átmeneti rendszer, melynek reprezentálása a fentiek szerinti és f CTL formula.
Kimenet: S_f halmazt reprezentáló Boole-függvény.
Módszer:

1. Az f formulán ekvivalens átalakításokat végzünk úgy, hogy csak olyan operátorok maradjanak az átalakított formulában, amelyek mindegyike a
   $\{\neg, \wedge, \mathbf{EX}, \mathbf{EU}, \mathbf{EG}\}$ halmazba esik.

2. Az f minden g közvetlen részformulájához meghatározzuk a $D_{[g]}$ ROBDD-t, ami a φ_{S_g} Boole-függvényt reprezentálja. Ezek felhasználásával határozzuk meg a $D_{[f]}$ ROBDD-t.

   - Ha $f = \underline{1}$, akkor $D_{[f]} = D_S$.
   - Ha $f = P_x$, ahol $x \in \mathcal{X}$, akkor $D_{[f]} = D_{S_x}$.
   - Ha $f = \neg g$, akkor $D_{[f]} = \bar{D}_{[g]}$.
   - Ha $f = \mathbf{EX}g$, akkor $D_{[f]} = \exists \bar{x}'(\text{redukál}(\text{metszet}(D_R, D'_{[g]})))$.
   - Ha $f = \mathbf{EU}(g, p)$, akkor $D_{[f]}$ kiszámítása:

     $D_0 = D_{[p]}$,
     $D_{i+1} = \text{redukál}(\text{unió}(D_i, \text{redukál}(\text{metszet}(D_{[g]}, K))))$, ahol
     $K = \exists \bar{x}'(\text{redukál}(\text{metszet}(D_R, D'_i)))$ és $0 \leq i < |S| = n$,
     $D_{[f]} = D_n$.
   - Ha $f = \mathbf{EG}\, g$, akkor $D_{[f]}$ kiszámítása:

     $D_0 = D_{[g]}$,
     $D_{i+1} = \text{redukál}(\text{metszet}(D_{[g]}, \exists \bar{x}'(\text{redukál}(\text{metszet}(D_R, D'_i)))))$, ahol $0 \leq i < |S| = n$,
     $D_{[f]} = D_n$.

Ahogyan a 4.2.1 fejezetben bemutatott algoritmusnál itt is a fenti két iterációs számítás befejeződhet a legkisebb olyan $0 \leq i < n$-re, melyre $D_{i+1} = D_i$, és ekkor $D_{[f]} = D_i$.

4.8. LTL automata-elméleti alapú modell-ellenőrzés

Legyen $M = (S, T, \alpha, \beta, S_{x_1}, \ldots, S_{x_n})$ egy $(\mathcal{X}, \emptyset)$-paraméteres átmeneti rendszer ($\mathcal{X} = \{x_1, \ldots, x_n\}$), melyben minden $s \in S$ állapotra van rákövetkező állapot, $S_0 \subseteq S$ az M kezdőállapotainak halmaza és f egy LTL formula az $AP = \{P_x \mid x \in \mathcal{X}\}$ felett. Annak eldöntése a feladat, hogy teljesül-e minden $s \in S_0$ állapotból induló M-beli végtelen c útra, hogy $M, c \models f$. Ez a probléma visszavezethető egy Büchi-automata által felismert nyelv ürességének vizsgálatára.

Jelölje $L : S \to \mathcal{P}(AP)$ azt a leképezést, melyre tetszőleges $a \in AP$ és $s \in S$ esetén teljesül, hogy $a \in L(s) \Leftrightarrow s \in S_x$, ahol $a = P_x$ valamely $x \in \mathcal{X}$-re.

Nevezzünk egy $v = v_1 v_1 \ldots$ végtelen $\mathcal{P}(AP)$ feletti szót a $c = s_0, s_1, \ldots$ M-beli végleten út *lenyomatának*, ha $L(s_i) = v_i$ minden $i \geq 0$-ra.

Néhány fogalom és állítás, melyeket a modell-ellenőrzés felhasznál:

- Legyen Σ egy véges ábécé. A Σ feletti végtelen szavak halmazát Σ^ω-val jelöljük: $\Sigma^\omega = \{v_0 v_1 \ldots \mid \forall i \geq 0\text{-ra } v_i \in \Sigma\}$. Tetszőleges Σ^ω-beli $v = v_0 v_1 \ldots$ szó esetén bevezetjük a v^i jelölést a v szó v_i-vel kezdődő részére, vagyis v^i a $v_i v_{i+1} \ldots$ szót jelöli minden $i \geq 0$-ra.

- *Büchi-automatának* nevezünk egy $B = (Q, \Sigma, \delta, I, F)$ rendszert, ahol

  - Q véges halmaz, az *állapotok* halmaza,
  - Σ véges halmaz, az *input ábécé*,
  - $\delta \subseteq Q \times \Sigma \times Q$ az *átmenetek* halmaza,
  - $I \subseteq Q$ a *kezdőállapotok* halmaza,
  - $F = \{F_1, \ldots, F_k\}$ és $\forall 1 \leq i \leq k$-ra $F_i \subseteq Q$.

  Legyen $v \in \Sigma^\omega$ és $\rho = q_0, q_1, \ldots$ állapotoknak egy végtelen sorozata. Azt mondjuk, hogy v-nek a ρ egy *számítási sorozata* B-ben, ha $\forall i \geq 0$-ra $(q_i, v_i, q_{i+1}) \in \delta$. Legyen $inf(\rho) = \{q \in Q \mid \text{végtelen sok } i\text{-re } q_i = q\}$.

  A $v \in \Sigma^\omega$ *szót elfogadja a B Büchi-automata*, ha van olyan $\rho = q_0, q_1, \ldots$ számítási sorozata B-ben, melyre teljesül, hogy

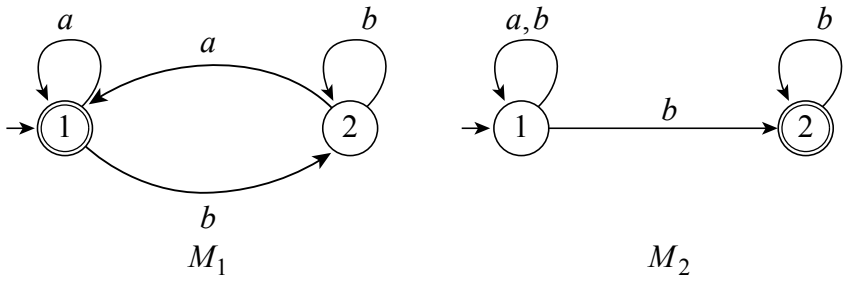
  - $q_0 \in I$
  - $\forall 1 \leq i \leq k$-ra $inf(\rho) \cap F_i \neq \emptyset$.

  A B Büchi-automata felismeri az $L \subseteq \Sigma^\omega$ nyelvet, ha $\forall v \in \Sigma^\omega$-ra $v \in L$ akkor és csak akkor, ha B elfogadja a v a szót. A B Büchi-automata által felismert nyelvet $L(B)$-vel jelöljük.

Ismeretesek az alábbi állítások:

- A Büchi-automatákkal felismerhető nyelvek osztálya zárt a metszetre és a komplementer képzésre (algoritmikusan megkonstruálhatók).

- Tetszőleges B Büchi-automatára eldönthető, hogy $L(B) = \emptyset$ teljesül-e.



4.8. ábra. Példák Büchi-automatákra

Példa. A 4.8 ábra két Büchi-automatát szemléltet, ahol I elemeit a csúcsba mutató nyíl, F_1 elemeit a kettős kör jelöli. Az M_1 Büchi-automata pontosan azokat a végtelen szavakat fogadja el, melyben végtelen sok a betű fordul elő. Az M_2 Büchi-automata azokat a végtelen szavakat fogadja el, melyekben csak véges sok a betű szerepel.

Az LTL automata-elméleti alapú modell-ellenőrzés főbb lépési:

- Megadunk egy olyan B_M Büchi-automatát, amely pontosan azokat a $\mathcal{P}(AP)$ feletti végtelen szavakat fogadja el, melyek az M átmeneti rendszer kezdőállapotaiból induló végtelen utak lenyomatai.

- Megadunk egy olyan B_f Büchi-automatát, amely azokat a $\mathcal{P}(AP)$ feletti végtelen szavakat fogadja el, melyek az f LTL formulát kielégítő utak lenyomatai.

- A B_f Büchi-automata ismeretében megadható olyan B' Büchi-automata, melyre $L(B')=\overline{L(B_f)}$, és megadható olyan B Büchi-automata, melyre $L(B)=L(B_M)\cap L(B')$. A modell-ellenőrzésre a választ az $L(B)$ nyelv ürességének vizsgálata adja meg.

1. Az M átmeneti rendszerhez és az $S_0 \subseteq S$ halmazhoz megadunk egy olyan B_M Büchi-automatát, melyre

$$L(B_M) = \{L(s_0)L(s_1)\ldots \mid s_0, s_1, \ldots \ M\text{-beli végtelen út és } s_0 \in S_0\}.$$

Legyen $B_M = (Q, \Sigma, \delta, I, F)$, ahol

- $Q = S \cup \{q_0\}$, $q_0 \notin S$,
- $\Sigma = \mathcal{P}(AP)$,
- minden $q, p \in S$-re $(q, a, p) \in \delta$ akkor és csak akkor, ha van olyan $t \in T$, hogy $\alpha(t)=q$, $\beta(t) = p$ és $a \in L(p)$,
- $(q_0, a, q) \in \delta$ akkor és csak akkor, ha $q \in S_0$ és $a \in L(q)$,
- $I = \{q_0\}$,
- $F = \{Q\}$.

2. Feltesszük, hogy az AP feletti f LTL formulában szereplő operátorok mindegyike a $\{\wedge, \neg, \mathbf{X}, \mathbf{U}\}$ operátor halmaznak eleme. Ez nem jelent megszorítást, mivel a $\{\wedge, \neg, \mathbf{X}, \mathbf{U}\}$ az LTL operátoroknak adekvát halmaza.

   Legyen $szavak(f) = \{v \in (\mathcal{P}(AP))^\omega \mid v \models f\}$, ahol $\models \subseteq (\mathcal{P}(AP))^\omega \times LTL$ a legszűkebb olyan reláció, amelyre tetszőleges $v = v_0 v_1 \ldots \in (\mathcal{P}(AP))^\omega$, $a \in AP$, g, g_1 és g_2 LTL formulák esetén teljesülnek a következők:

   - $v \models \underline{1}$,
   - $v \models a \Leftrightarrow a \in v_0$,
   - $v \models \neg g \Leftrightarrow v \not\models g$,
   - $v \models g_1 \wedge g_2 \Leftrightarrow v \models g_1$ és $v \models g_2$,
   - $v \models \mathbf{X} g \Leftrightarrow v^1 \models g$,
   - $v \models g_1 \mathbf{U} g_2 \Leftrightarrow \exists j \geq 0$, hogy $v^j \models g_2$ és $\forall 0 \leq i < j$-re $v^i \models g_1$.

   Egy $v \in (\mathcal{P}(AP))^\omega$ szó akkor és csak akkor van benne a $szavak(f)$ halmazban, ha olyan végtelen útnak a lenyomata, melyre az f teljesül.

   Legyen f egy LTL formula. Jelölje $C(f)$ a $\{g, \bar{g} \mid g$ az f részformulája$\}$ halmazt, ahol

$$\bar{g} = \begin{cases} g', & \text{ha } g = \neg g' \text{ alakú;} \\ \neg g, & \text{különben.} \end{cases}$$

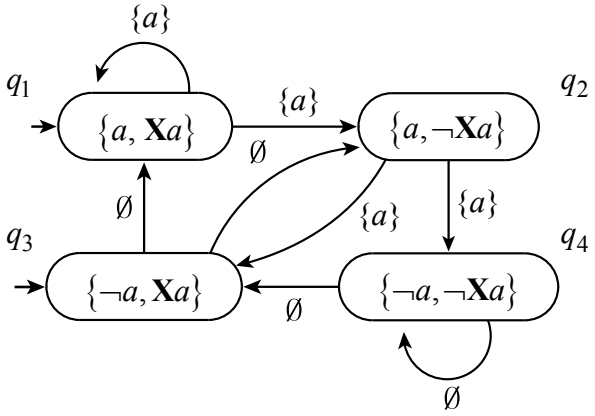
   Az AP feletti f LTL formulához megadunk egy olyan B_f Büchi-automatát, melyre $L(B_f) = szavak(f)$. Legyen $B_f = (Q, \Sigma, \delta, I, F)$, ahol

   - $\Sigma = \mathcal{P}(AP)$,
   - $Q \subseteq \mathcal{P}(C(f))$, melynek bármely $q \in Q$ állapotára teljesül, hogy
     - q maximális, vagyis bármely $q \in C(f)$-re ha $g \notin q$, akkor $\neg g \in q$,
     - minden $g_1 \wedge g_2$, $g \in C(f)$-re:
       $g_1 \wedge g_2 \in q \Leftrightarrow g_1 \in q$ és $g_2 \in q$,
       $g \in q \Leftrightarrow \neg g \notin q$,
     - q lokálisan konzisztens az $\mathbf{U}$ operátor szerint, vagyis minden $g_1 \mathbf{U} g_2 \in C(f)$-re
       ha $g_2 \in q$, akkor $g_1 \mathbf{U} g_2 \in q$,
       ha $g_1 \mathbf{U} g_2 \in q$ és $g_2 \notin q$, akkor $g_1 \in q$,
   - $I = \{q \in Q \mid f \in q\}$,
   - átmenetek δ halmaza:
     legyen $a \in \Sigma$, $q, r \in Q$, akkor $(q, a, r) \in \delta$ akkor és csak akkor, ha $a = \{p \in AP \mid p \in q\}$ és teljesülnek az alábbiak:
     - ha $\mathbf{X} g \in C(f)$, akkor $\mathbf{X} g \in q \Leftrightarrow g \in r$,
     - ha $g \mathbf{U} h \in C(f)$, akkor $g \mathbf{U} h \in q \Leftrightarrow (h \in q$ vagy $g \in q$ és $g \mathbf{U} h \in r)$,
   - $F = \{F_{g \mathbf{U} h} \mid g \mathbf{U} h \in C(f)\}$, ahol $F_{g \mathbf{U} h} = \{q \in Q \mid h \in q$ vagy $g \mathbf{U} h \notin q\}$.

A B_f Büchi-automata konstrukció a következőn alapszik. Legyen $v = v_0 v_1 \ldots$ tetszőleges végtelen szó a $szavak(f)$-ben. Vegyük azt a $c = B_0, B_1, \ldots$ végtelen sorozatot, melyben minden $i \geq 0$-ra a B_i a v_i-nek az f olyan g részformuláival való kiterjesztése, melyre teljesül, hogy $g \in B_i$ akkor és csak akkor, ha $v^i \models g$, Ezek a B_i halmazok a B_f Büchi-automata állapotai lesznek és $(B_i, v_i, B_{i+1}) \in \delta$.

A B_f Büchi-automata végállapotait pedig úgy definiáljuk, hogy a v szót a c számítási sorozat alapján elfogadja.

A B_f Büchi-automata megkonstruálásának hely- és időigénye $2^{|f|}$, mivel $Q \subseteq \mathcal{P}(C(f))$.



4.9. ábra. $\mathbf{X}a$ *LTL formulához konstruált* $B_{\mathbf{X}a}$ *Büchi-automata*

Példa. Két példában mutatjuk meg az f LTL formulához a B_f Büchi-automata konstruálását.

a) Legyen $AP = \{a\}$ és $f = \mathbf{X}a$. Akkor $C(\mathbf{X}a) = \{a, \neg a, \mathbf{X}a, \neg\mathbf{X}a\}$.
A $B_{\mathbf{X}a} = (Q, \Sigma, \delta, I, F)$ Büchi-automata komponensei a következők:

$Q = \{q_1, q_2, q_3, q_4\}$, ahol $q_1 = \{a, \mathbf{X}a\}$, $q_2 = \{a, \neg\mathcal{X}a\}$, $q_3 = \{\neg a, \mathbf{X}a\}$, $q_4 = \{\neg a, \neg\mathbf{X}a\}$.

$I = \{q_1, q_3\}$, a kezdőállapotok halmaza.

$\Sigma = \mathcal{P}(AP) = \{\emptyset, \{a\}\}$.

$F = \emptyset$ mivel $C(\mathbf{X}a)$-ban nincs **U**-os formula.

δ-ba tartozó átmeneteket és a teljes $B_{\mathbf{X}a}$ Büchi-automatát a 4.9. ábra szemlélteti. Az ábrán a kezdőállapotokat a csúcsba mutató nyíl jelöli.
Mivel $F = \emptyset$ ezért minden q_1 vagy q_3-ból induló végtelen számítási sorozathoz tartozó v végtelen szót a $B_{\mathbf{X}a}$ elfogad, és ezek mindegyike a $szavak(\mathbf{X}a)$-nak eleme, mivel $v^1 \models a$.

b) Legyen $AP = \{a, b\}$ és $f = a\mathbf{U}b$. Akkor $C(f) = \{a, b, \neg a, \neg b, a\mathbf{U}b, \neg(a\mathbf{U}b)\}$. A $B_{a\mathbf{U}b} = (Q, \Sigma, \delta, I, F)$ Büchi-automata komponensei a következők:

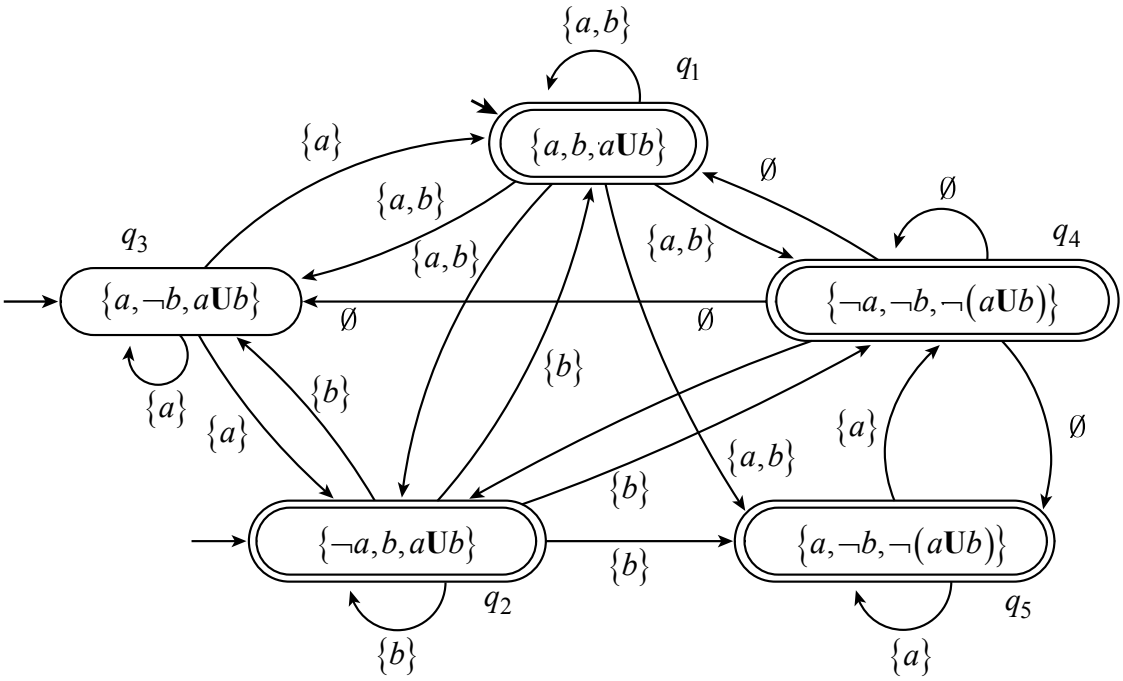
$Q = \{q_1, q_2, q_3, q_4, q_5\}$, ahol $q_1 = \{a, b, f\}$, $q_2 = \{\neg a, b, f\}$, $q_3 = \{a, \neg b, f\}$, $q_4 = \{\neg a, \neg b, \neg f\}$, $q_5 = \{a, \neg b, \neg f\}$.

$\Sigma = \mathcal{P}(AP)$

$I = \{q_1, q_2, q_3\}$, a kezdőállapotok halmaza.

$F = \{F_f\}$, ahol $F_f = \{q_1, q_2, q_4, q_5\}$, a végállapotok halmaza.

δ-ba tartozó átmeneteket és a teljes $B_{a\mathbf{U}b}$ Büchi-automatát a 4.10. ábra szemlélteti. Ezen az ábrán is a kezdőállapotokat a csúcsba mutató nyíl, az F_f elemeit a dupla vonalú keretezés jelöli.



4.10. ábra. $a\mathbf{U}b$ LTL formulához konstruált $B_{a\mathbf{U}b}$ Büchi-automata

Például könnyen látható, hogy a q_3^ω végtelen számítási sorozathoz tartozó $\{a\}^\omega \notin L(B_{a\mathbf{U}b})$, míg a $q_3q_1q_4^\omega$ végtelen számítási sorozathoz tartozó $\{a\}\{a,b\}\emptyset^\omega \in L(B_{a\mathbf{U}b})$, mely megfelel annak, hogy $\{a\}^\omega \notin szavak(a\mathbf{U}b)$, illetve $\{a\}\{a,b\}\emptyset^\omega \in szavak(a\mathbf{U}b)$.

3. Az M átmeneti rendszerhez és $S_0 \subseteq S$ halmazhoz konstruált B_M Büchi-automata és az AP feletti f LTL formulához konstruált B_f Büchi-automata ismeretében annak eldöntése, hogy teljesül-e minden $s_0 \in S$ állapotból induló végtelen c útra az $M, c \models f$, visszavezethető az $L(B_M) \subseteq L(B_f)$ vizsgálatára.

$$\begin{aligned} L(B_M) \subseteq L(B_f) &\Leftrightarrow L(B_M) \cap \overline{L(B_f)} = \emptyset \\ &\Leftrightarrow L(B_M) \cap L(B_{\neg f}) = \emptyset. \end{aligned}$$

Mivel a Büchi-automatákkal felismerhető nyelvek osztálya zárt a komplementer és metszet képzésre, így megkonstruálható egy olyan B Büchi-automata, amely az $L(B_M) \cap \overline{L(B_f)}$ nyelvet ismeri fel. A Büchi-automatákról eldönthető, hogy a felismert nyelvük üres-e, így eldönthető, hogy $L(B) = \emptyset$ teljesül-e.

Ismert, hogy az LTL automata-elméleti alapú modell-ellenőrzés időbonyolultsága

$$\mathcal{O}(|S| \cdot 2^{|f|}).$$

4.9. Tabló-módszer alapú modell-ellenőrzés Hennessy-Milner formulákra

A Boole formuláknál alkalmazható tabló-módszer segítségével logikai formulák kielégíthetősége vizsgálható. A módszer bizonyítást keres a vizsgált formula szabályok szerinti (tablók) felbontásán alapuló szemantikus fa megkonstruálásával.

A Hennessy-Milner logika (HML) lokális modell-ellenőrzési feladatának megválaszolására a tabló módszer alkalmas.

Legyen $M = (S, T, \alpha, \beta, S_{x_1}, \ldots, S_{x_n}, T_{y_1}, \ldots, T_{y_m})$ egy $(\mathcal{X}, \mathcal{Y})$-paraméteres átmeneti rendszer $(\mathcal{X} = \{x_1, \ldots, x_n\}, \mathcal{Y} = \{y_1, \ldots, y_m\})$, melyre teljesül, hogy minden $t \in T$ a T_{y_1}, …, T_{y_m} halmazok közül pontosan egynek az eleme.

Legyen $\lambda : T \to \mathcal{Y}$ az a leképezés, melyre minden $t \in T$ esetén teljesül, hogy $\lambda(t) = y_i$, ahol $t \in T_{y_i}$.

Legyen $\mu_\tau : S \to \mathcal{P}(\mathcal{X})$ az a leképezés, melyre minden $s \in S$-re $\mu_\tau(s) = \{x_i \mid s \in S_{x_i}, 1 \leq \leq i \leq n\}$.

Jelölje $AP = \{P_x \mid x \in \mathcal{X}\}$ a HML atomi kijelentéseinek halmazát. Legyen f egy olyan HML formula, amelyben a negálás operátor összetett részformulára nincs alkalmazva. Ez nem jelent megszorítást, mivel azonosságok alkalmazásával a negálás operátor összetett részformulákról átvihető az atomi kijelentésekre.

A HML modell-ellenőrzés feladata eldönteni a fenti tulajdonsággal bíró M átmeneti rendszer, $s_0 \in S$ és f HML formula esetén, hogy $M, s_0 \models f$ teljesül-e.

Az $M, s \models g$ jelölés helyett az $s \vdash g$ jelölést alkalmazzuk. Jelölje $R(f)$ az f formula összes részformuláinak halmazát és V az $\{s \vdash g \mid s \in S,\ g \in R(f)\} \cup \{0,1\}$ halmazt. Tetszőleges $s \in S$ állapot és $a \in \mathcal{Y}$ átmenet címke esetén jelölje $A(s,a)$ a $\{q \in S \mid \exists t \in T,\ \alpha(t) = s,\ \lambda(t) = = a,\ \beta(t) = q\}$ halmazt.

A tabló-módszer az $M, s_0 \models f$ feladathoz egy olyan véges, címkézett *szemantikus fát* épít fel, melyben a csúcsok címkéi a $\mathcal{P}(V)$ elemei. Jelölje $c(v)$ a v csúcs címkéjét a szemantikus fában.

A szemantikus t fa konstruálásánál alkalmazott szabályok:
Kezdetben a t fának egyetlen v csúcsa van és $c(v) = \{s_0 \vdash f\}$. Legyen v a t fa egy olyan levél csúcsa, melyre $0 \notin c(v)$, és válasszuk $c(v)$-nek valamely $s \vdash g$ elemét.

1. Ha $g = p$ vagy $g = \neg p$, ahol $p \in AP$, akkor az M alapján az $s \vdash g$ helyébe 1-et vagy 0-t írunk annak megfelelően, hogy $M, s \models g$ vagy $M, s \not\models g$.

2. Ha $g = \underline{0}$ vagy $g = \underline{1}$, akkor az $s \vdash g$ helyett a g-t írjuk be a $c(v)$-be.

3. Ha $g = g_1 \wedge g_2$, akkor v-nek egy v_1 leszármazottja lesz, és

$$c(v_1) = (c(v) - \{s \vdash g\}) \cup \{s \vdash g_1, s \vdash g_2\}.$$

4. Ha $g = g_1 \vee g_2$, akkor v-nek két leszármazottja lesz v_1 és v_2, és

$$c(v_1) = (c(v) - \{s \vdash g\}) \cup \{s \vdash g_1\},$$
$$c(v_2) = (c(v) - \{s \vdash g\}) \cup \{s \vdash g_2\}.$$

5. Ha $g = [a]g_1$ és $A(s,a) = \{s_1, \ldots, s_k\} \neq \emptyset$, akkor v-nek egy v_1 leszármazottja lesz és

$$c(v_1) = (c(v) - \{s \vdash g\}) \cup \{s_1 \vdash g_1, \ldots, s_k \vdash g_1\}.$$

   Ha $A(s,a) = \emptyset$, akkor $c(v) = (c(v) - \{s \vdash g\}) \cup \{1\}$.

6. Ha $g = \langle a \rangle g_1$ és $A(s,a) = \{s_1, \ldots, s_k\} \neq \emptyset$, akkor v-nek k leszármazottja lesz, jelölje ezeket $v_1, \ldots, v_k$ és

$$c(v_i) = (c(v) - \{s \vdash g\}) \cup \{s_i \vdash g_1\}$$

   minden $1 \leq i \leq k$-ra. Ha $A(s,a) = \emptyset$, akkor $c(v) = (c(v) - \{s \vdash g\}) \cup \{0\}$.

A t fa megkonstruálása akkor fejeződik be, mikor a fa minden levelének címkéje $\{1\}$ vagy tartalmazza a 0-t. A szemantikus fa véges sok lépésben megkonstruálható, mivel a 3., 4., 5. és 6. esetben a fa v csúcsához kapcsolt bármely v' csúcs $c(v')$ címkéje a $c(v)$-től annyiban tér el, hogy a $c(v)$-ben levő valamely $s \vdash g$ elem helyébe olyan $r \vdash p$ V-beli elem kerül, melyre p a g részformulája.

A konstrukció 1. és 2. esete szerint a $c(v)$-ben levő $s \vdash g$ elem helyébe az általa jelölt $M, s \models g$ reláció teljesülése esetén 1, egyébként a 0 értéket írjuk. Tegyük fel, hogy a szemantikus fa megkonstruálása során egy v csúcshoz leszármazottként a v' csúcsot kapcsoljuk (a 3., 4., 5., vagy 6. eset szerint). Belátható a HML formulák szemantikája alapján, hogy ha a $c(v')$-ben szereplő minden $r \vdash p$ által jelölt $M, r \models p$ reláció teljesülne, ahol $r \in S$, $p \in R(f)$, akkor a $c(v)$-ben levő minden V-beli elem által jelölt reláció is teljesülne. Továbbá, ha $c(v') = \{1\}$, akkor a $c(v)$-ben levő elemek által jelölt reláció is teljesülne.

A szemantikus fa konstruálása nemdeterminisztikus, vagyis egy relációhoz több szemantikus fa is konstruálható.

Nevezzük *zártnak* a szemantikus fát, ha minden levél csúcs címkéje a 0 értéket tartalmazza, egyébként nevezzük *nyitottnak*.

Belátható, hogy tetszőleges s_0 és f esetén az alábbi 3 állítás ekvivalens:

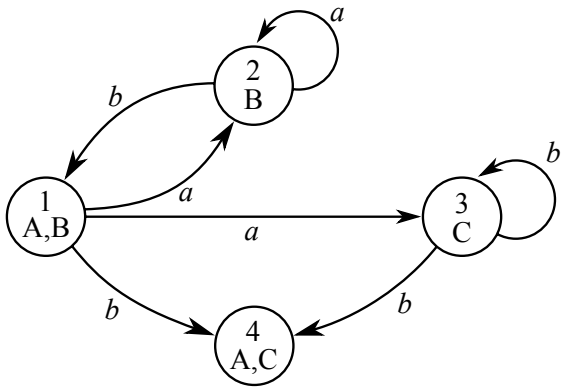
(1) $M, s_0 \models f$ teljesül.

(2) Az $\{s_0 \vdash f\}$ címkéjű csúcshoz konstruálható nyitott szemantikus fa.

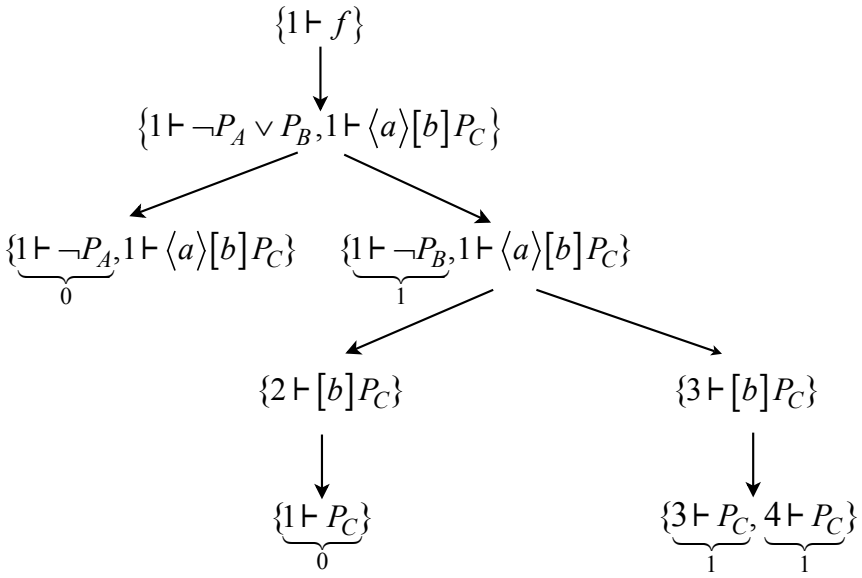
(3) Az $\{s_0 \vdash f\}$ címkéjű csúcshoz konstruálható minden szemantikus fa nyitott.

Példa. Tekintsük a 4.11 ábrán látható $M = (S, T, \alpha, \beta, S_A, S_B, S_C, T_a, T_b)$ $(\{A, B, C\}, \{a, b\})$-paraméteres átmeneti rendszert. Ekkor $AP = \{P_A, P_B, P_C\}$. Látható, hogy M minden átmenete pontosan egy átmenet tulajdonsággal rendelkezik, vagyis M egy címkézett átmeneti rendszer.

Legyen $f = (\neg P_A \vee P_B) \wedge \langle a \rangle [b] P_C$. A 4.12. ábra egy t szemantikus fát szemléltet az $M,1 \models f$ vizsgálatához. Mivel a t szemantikus fában van olyan levél csúcs, melynek címkéjében csak az 1 érték van, így a t nyitott fa, vagyis az $M,1 \models f$ teljesül.



4.11. ábra. Az $M(\{A, B, C\}, \{a, b\})$-paraméteres átmeneti rendszer



4.12. ábra. $(\neg P_A \vee P_B) \wedge \langle a \rangle [b] P_C$ formulához egy t szemantikus fa M felett

4.10. TCTL modell-ellenőrzés

Ebben a fejezetben a 2.9. és a 3.8. fejezetekben bevezetett jelöléseket alkalmazzuk. Legyen $\mathcal{A} = (L, A, C, L_0, E, I, AP, \rho)$ egy olyan véges időzített automata, melynek $T(\mathcal{A})$ átmeneti rendszere idő-divergens és legyen ϕ egy TCTL formula az AP atomi kijelentések és C órák halmaza felett.

A TCTL modell-ellenőrzés feladata eldönteni, hogy az $\mathcal{A} \models \phi$ teljesül-e, vagyis $T(\mathcal{A}) \models \phi$ teljesül-e.

A $T(\mathcal{A})$ átmeneti rendszer konfigurációs gráfja végtelen, így a gráf bejárásával történő elemzés nem járható út annak megválaszolására, hogy $T(\mathcal{A})$ minden kezdő konfigurációja kielégíti-e a ϕ formulát.

A módszer leírásánál először a lépéseket tekintjük át, majd ezeket részletesen is tárgyaljuk. A TCTL modell-ellenőrzésnél alkalmazott módszer:

Bemenet: $\mathcal{A}$ időzített automata és ϕ TCTL formula az AP atomi kijelentések halmaza és a C órák halmaza felett.
Kimenet: "igen", ha $\mathcal{A} \models \phi$, egyébként "nem".
Módszer:

1. A ϕ formulából az idő paraméterek eliminálásával a $\hat{\phi}$ CTL formula meghatározása.

2. A $T(\mathcal{A})$ konfigurációi halmazán a $\cong$ ekvivalencia reláció szerinti osztályok (régiók) meghatározása. Egy $\mathcal{A}$ időzített automatában véges sok óra van és ezekre az órákra véges sok órafeltétel, illetve a vizsgált ϕ formulában is véges sok órafeltétel fordul elő. Ez lehetővé teszi, hogy a végtelen sok óra értékelés véges sok olyan osztályba besorolható legyen, melyekbe tartozó óra értékelések az $\mathcal{A}$ specifikációjában és a ϕ-ben szereplő órafeltételek közül ugyanazokat az órafeltételeket elégítik ki. A $T(\mathcal{A})$ végtelen sok konfigurációja ezután besorolható lesz véges sok olyan osztályba, melybe tartozó konfigurációk ugyanazon ϕ formulákat elégítik ki.

3. Az $RT(\mathcal{A}, \phi)$ régió-átmenet rendszer megkonstruálása úgy, hogy teljesüljön, $T(\mathcal{A}) \models \phi$ akkor és csak akkor, ha $RT(A, \phi) \models \hat{\phi}$.

4. A CTL modell-ellenőrzés algoritmus alkalmazásával az $RT(\mathcal{A}, \phi) \models \hat{\phi}$ vizsgálata. Ha $RT(\mathcal{A}, \phi) \models \hat{\phi}$ teljesül, akkor a kimenet "igen", egyébként "nem".

Bevezetjük az $\mathcal{AB}(\mathcal{A})$ és az $\mathcal{AB}(\phi)$ jelöléseket az $\mathcal{A}$ specifikációjában, illetve ϕ-ben előforduló atomi órafeltételek halmazának jelölésére.

1. lépés: Az idő paraméterek eliminálása.

- Ha a ϕ formulában operátor idő paraméterként a J $[0, \infty)$ fordul elő, akkor ennek eliminálásához bővítjük a C órahalmazt egy új z órával, töröljük ϕ-ben a J operátor idő paramétert, és konjunkcióval a formulához kapcsoljuk a $(z \in J)$ atomi órafeltételt.
- Jelölje $\mathrm{TCTL}_{\geq 0}$ azoknak a TCTL formuláknak a halmazát, melyekben operátor idő paraméterként csak a $[0, \infty)$ intervallum fordul elő. Ezekben a paraméterek törölhetőek, mivel a formula szemantikájában nem játszanak szerepet. A törléssel előálló formulákban időre vonatkozó rész csak atomi kijelentésként jelenhet meg, így ezek a formulák már CTL formulák lesznek az $AP' = AP \cup \mathcal{AB}(\mathcal{A}) \cup \mathcal{AB}(\phi)$ atomi kijelentések felett, vagyis a $\mathrm{TCTL}_{\geq 0}$ beágyazható CTL-be.

Amennyiben $z \in C$ új óra, jelölje $\mathcal{A} \oplus z$ az $(L, A, C \cup \{z\}, L_0, E, I, AP, \rho)$ időzített automatát. Tetszőleges $v \in V(C)$ óra értékelés, $d \in \mathbb{R}_{\geq 0}$ és $z \in C$ esetén legyen

$$v\{z := d\}(x) = \begin{cases} v(x), & \text{ha } x \in C \\ d, & \text{ha } x = z. \end{cases}$$

Tetszőleges $q = \langle \ell, v \rangle$ $T(\mathcal{A})$-beli konfigurációnál a $q\{z := d\}$ jelölje a $\langle \ell, v\{z := d\} \rangle$ $T(\mathcal{A} \oplus z)$-beli konfigurációt.

Belátható tetszőleges q $T(\mathcal{A})$-beli konfigurációra, $\mathbf{EU}_J(\phi,\psi)$, illetve $\mathbf{AU}_J(\phi,\psi)$ TCTL formulákra, hogy

$$T(\mathcal{A}),q \models \mathbf{EU}_J(\phi,\psi) \Leftrightarrow T(\mathcal{A}\oplus z),q\{z:=0\} \models \mathbf{EU}[(\phi\vee\psi),(z\in J)\wedge\psi],$$

illetve

$$T(\mathcal{A}),q \models \mathbf{AU}_J(\phi,\psi) \Leftrightarrow T(\mathcal{A}\oplus z),q\{z:=0\} \models \mathbf{AU}[(\phi\vee\psi),(z\in J)\wedge\psi].$$

Az 1. lépéssel előálló $\hat{\phi}$ formula már AP és C' feletti $\mathrm{TCTL}_{\geq 0}$ formula lesz, ahol C' a C új órákkal való bővítése.

Példa. Ha $\phi_1 = \mathbf{EF}_{\leq 2}\psi$, akkor $\hat{\phi_1} = \mathbf{EF}[(z\leq 2)\wedge\hat{\psi}]$. Egy $\phi_2 = \neg\mathbf{AF}_{\leq 2}\neg\psi$ alakú formula esetén (amely ekvivalens az $\mathbf{EG}_{\leq 2}\psi$ formulával), a $\hat{\phi_2} = \neg\mathbf{AF}[(z\leq 2)\wedge\neg\hat{\psi}]$.

2. lépés: $\cong$ ekvivalencia reláció megadása a $T(\mathcal{A})$ konfigurációs halmazán.
Az 1. lépés után feltehető, hogy $\mathcal{A}$ már a szükséges órákkal bővített időzített automata és ϕ egy $\mathrm{TCTL}_{\geq 0}$ halmazba tartozó formula az AP' felett.

Jelölések:
Ha $d\in\mathbb{R}_{\geq 0}$, akkor $\lfloor d\rfloor$ jelölje a d egészrészét és $\langle d\rangle$ jelölje a d törtrészét. Ha $x\in C$, akkor jelölje c_x azt a legnagyobb konstanst, ami az x-re vonatkozó órafeltételekben előfordul. (Az órafeltételek a ϕ formulában vagy az $\mathcal{A}$ időzített automatában az átmeneteknél őrfeltételként vagy helyinvariánsként szerepelnek.)

Definíció. Legyen C egy véges órahalmaz, v,v' tetszőleges $V(C)$-beli elemek. Azt mondjuk, hogy v és v' *óra-ekvivalensek*, amit $v\cong_\sigma v'$-vel jelölünk, akkor és csak akkor, ha mindegyik $x,y\in C$-re teljesül:

1. $v(x)>c_x$ és $v'(x)>c_x$ vagy $\lfloor v(x)\rfloor=\lfloor v'(x)\rfloor$;
2. ha $v(x)<c_x$, akkor $\langle v(x)\rangle = 0 \Leftrightarrow \langle v'(x)\rangle = 0$;
3. ha $v(x)\leq c_x$ és $v(y)\leq c_x$, akkor $\langle v(x)\rangle \leq \langle v(y)\rangle \Leftrightarrow \langle v'(x)\rangle \leq \langle v'(y)\rangle$.

A $\cong_\sigma$ reláció a $V(C)$ halmazon ekvivalencia reláció. Jelölje $V(C)/\cong_\sigma$ a $V(C)$ halmaz $\cong_\sigma$ reláció szerinti osztályainak (*óra-régiók*) halmazát. Bármely $r\in V(C)/\cong_\sigma$ óra-régióra és $v,v'\in r$-re belátható, hogy minden $g\in\mathcal{AB}(\mathcal{A})\cup\mathcal{AB}(\phi)$ esetén $v\models g \Leftrightarrow v'\models g$.

A $V(C)$ végtelen halmaz $\cong_\sigma$ reláció szerinti osztályainak száma véges. Az osztályok számának egy felső korlátja: $2^{|C|}\cdot|C|!\cdot\prod_{x\in C}(2c_x+2)$.
Jelölje r_∞ a $\{v\in V(C)\mid \forall x\in C\text{-re } v(x)\geq c_x\}$ halmazt. Tetszőleges $v\in V(C)$-re jelölje $[v]$ a $\{v'\in V(C)\mid v\cong_\sigma v'\}$ halmazt, vagyis azt az óra-régiót, melynek eleme v.

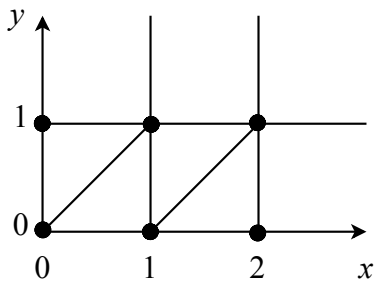
Definíció. Legyen $r,r'\in V(C)/\cong_\sigma$. Azt mondjuk, hogy r-nek az r' *leszármazott óra-régiója*, amit $\rightarrow(r)$ jelöl,

- ha $r=r_\infty$ és $r=r'$ vagy
- ha $r\ r_\infty$, $r\ r'$ és $\forall v\in r$-re $\exists d\in\mathbb{R}_{>0}$, hogy $v+d\in r'$ és $\forall 0\leq d'\leq d$-re $v+d'\in r\cup r'$.

Definíció. Legyen $r \in V(C)/\cong_\sigma$ és $g \in \mathcal{AB}(\mathcal{A}) \cup \mathcal{AB}(\phi)$. Azt mondjuk, hogy r *kielégíti* a g órafeltételt, amit $r \models g$ jelöl, ha $\exists v \in V(C)$, hogy $v \in r$ és $v \models g$. (Mint ahogy ezt már említettük, ez ekvivalens azzal, hogy $v' \models g$ minden $v' \in r$ értékelésre.)

Bármely $r \in V(C)/\cong_\sigma$, $v, v' \in r$ és $D \subseteq C$-re belátható, hogy $v[D \mapsto 0] \cong_\sigma v'[D \mapsto 0]$. Jelölje $r[D \mapsto 0] = \{v[D \mapsto 0] \mid v \in r\}$.

Példa. Legyen $C = \{x, y\}$, $c_x = 2$, $c_y = 1$. A 4.13. ábra szemlélteti grafikusan a $\cong_\sigma$ reláció szerinti osztályokat. A $V(C)$ összesen 28 ekvivalencia osztályt tartalmaz. Az osztályok



4.13. ábra. Óra-ekvivalencia szerinti osztályok, ahol $C = \{x, y\}$, $c_x = 2$, $c_y = 1$

az ábra szerint a következők: a 6 sarokpont (metszéspont), a 9 nyílt szakasz, az 5 félegyenes, a 4 háromszög belső területe és a 4 nem korlátos terület. Például sarokpont az $[x = 0, y = 0]$, nyílt szakasz az $[1 < x < 2, y = 1]$ vagy a $[0 < x < 1, 0 < y < 1, x = y]$, félegyenes a $[2 < x, y = 1]$, háromszög belső területe a $[0 < x < 1, 0 < y < 1, x > y]$, nem korlátos terület a $[2 < x, 0 < y < 1]$.

Néhány példa leszármazott óra-régióra:

$$\to [x = 0,\ y = 0] = [0 < x < 1,\ 0 < y < 1,\ x = y],$$

$$\to [0 < x < 1,\ 0 < y < 1,\ x = y] = [x = 1,\ y = 1],$$

$$\to [x = 2,\ y > 1] = r_\infty = [x > 2,\ y > 1],$$

$$\to [1 < x < 2,\ 0 < y < 1,\ x = y] = [x = 2,\ y = 1].$$

A továbbiakban Q-val jelöljük a $T(\mathcal{A})$ konfigurációs halmazát, mely $q = \langle \ell, v \rangle$ alakú párokból áll, ahol $\ell \in L$ és $v \in V(C)$.

Definíció. Legyen $\mathcal{A}$ egy időzített automata, ϕ egy TCTL$_{\geq 0}$-beli formula, $\langle \ell, v \rangle$ és $\langle \ell', v' \rangle$ tetszőleges $T(\mathcal{A})$-beli konfigurációk. Az $\langle \ell, v \rangle$ és $\langle \ell', v' \rangle$ *konfiguráció-ekvivalensek*, amit $\langle \ell, v \rangle \cong \langle \ell, v' \rangle$ jelöl, akkor és csak akkor, ha $\ell = \ell'$ és $v \cong_\sigma v'$.

A $T(\mathcal{A})$ Q konfigurációs halmazán a $\cong$ reláció ekvivalencia reláció. Jelölje $[q]$ a $Q/\cong$ *konfiguráció-régiók* halmazának azt az elemét, melynek eleme q. Tetszőleges $q = \langle \ell, v \rangle \in Q$ esetén bevezetjük a $[q] = \langle \ell, [v] \rangle$ jelölést is a q-t tartalmazó ekvivalencia osztályra (konfiguráció-régióra).

Belátható tetszőleges $q, q' \in Q$-ra, hogy ha $q \cong q'$, akkor $q \models a \Leftrightarrow q' \models a$ minden $a \in AP$-re.

3. lépés: $RT(\mathcal{A}, \phi)$ régió-átmenet rendszer megkonstruálása.

Definíció. Legyen $\mathcal{A} = (L, A, C, L_0, E, I, AP, \rho)$ egy idő-divergens időzített automata és ϕ egy $\mathrm{TCTL}_{\geq 0}$-beli formula. Akkor $\mathcal{A}$-nak a *régió-átmeneti rendszere* az $RT(\mathcal{A}, \phi) = (S, A', T, S_0, AP', \rho')$ rendszer, ahol

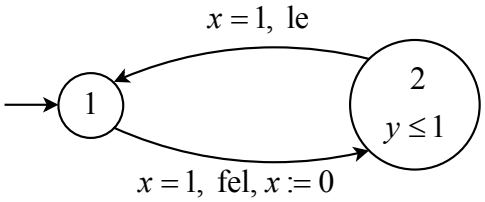
- $S = Q/\cong = \{[q] \mid q \in Q\}$ az állapotok halmaza, ahol Q a $T(\mathcal{A})$ konfigurációs halmaza,
- $AP' = \mathcal{AB}(\mathcal{A}) \cup \mathcal{AB}(\phi) \cup AP$ az atomi kijelentések halmaza,
- $\rho' : S \to \mathcal{P}(AP')$, amelyre $\forall s = \langle \ell, [v] \rangle \in S$-re teljesül, hogy

$$\rho'(\langle \ell, [v] \rangle) = \rho(\ell) \cup \{g \in \mathcal{AB}(\mathcal{A}) \cup \mathcal{AB}(\phi) \mid [v] \models g\},$$

- $S_0 = \{[q] \mid q \in Q_0\}$ a kezdőállapotok halmaza, ahol Q_0 a $T(\mathcal{A})$ kezdő konfigurációinak halmaza,
- $A' = A \cup \{\varepsilon\}$ az akciók halmaza, ahol $\varepsilon \notin A$,
- $T \subseteq S \times A' \times S$ az átmeneti relációk halmaza, mely a következő átmenetekből áll:
  - ha $\ell \xrightarrow{g,\alpha,D} \ell' \in E$, $r \models g$ és $r[D \mapsto 0] \models I(\ell')$, ahol $r \in V(C)/\cong_\sigma$, akkor $(\langle \ell, r \rangle, \alpha, \langle \ell', r[D \mapsto 0] \rangle) \in T$,
  - ha $r \models I(\ell)$ és $\to (r) \models I(\ell)$, ahol $r \in V(C)/\cong_\sigma$, akkor $(\langle \ell, r \rangle, \varepsilon, \langle \ell, \to (r) \rangle) \in T$.

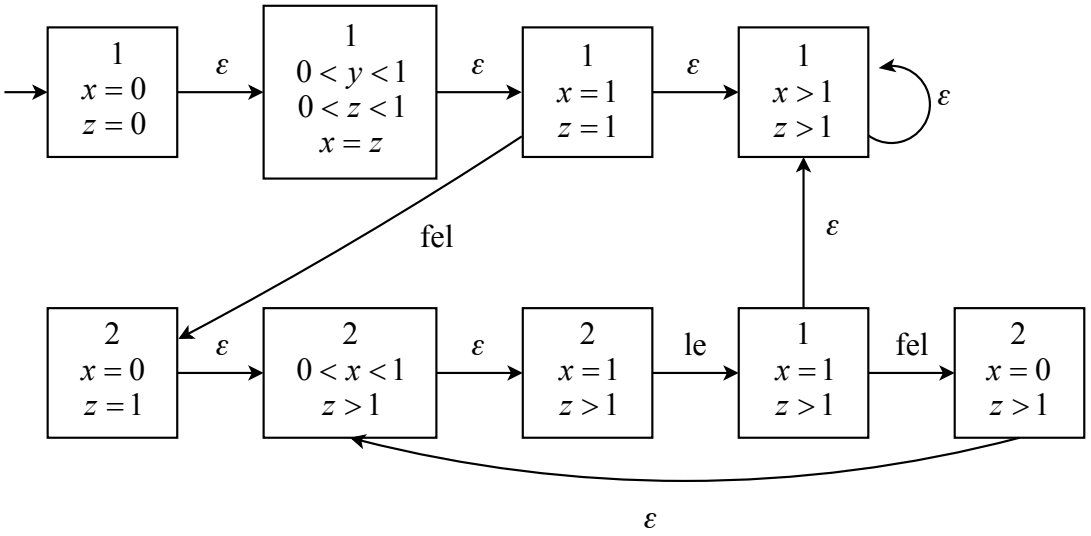
A régió-átmeneti rendszerben a T-beli átmeneti relációk definíciója szerint, ha a $T(\mathcal{A})$-ban valamely $\langle \ell, v \rangle$ konfigurációból egy $a \in A$ akcióra van átmenet az $\langle \ell', v' \rangle$ konfigurációba, akkor az $\langle \ell, [v] \rangle \in S$ állapotból van átmenet az a akcióra az $\langle \ell', [v'] \rangle \in S$ állapotba. Továbbá tetszőleges $\ell \in L$, $v, v' \in V(C)$ esetén, ha $v \models I(\ell)$, $v' \models I(\ell)$ és $\to [v] = [v']$, akkor az $\langle \ell, [v] \rangle \in S$ állapotból ε akcióra vezet átmenet az $\langle \ell, [v'] \rangle \in S$ állapotba.

Az $RT(\mathcal{A}, \phi)$ régió-átmeneti rendszerben az S, A', T, AP' véges halmazok.



4.14. ábra. A lámpakapcsoló $\mathcal{A}$ időzített automata modellje

Példa. Legyen az $\mathcal{A}$ időzített automata a 4.14. ábrán látható, ahol $AP = \{ki, be\}$, $\rho(1) = \{ki\}$, $\rho(2) = \{be\}$ és $\phi = \mathbf{EF}_{\leq 1} ki$ a TCTL formula. Ekkor $\hat{\phi} = \mathbf{EF}[(z \leq 1) \wedge ki]$, ahol z egy új óra. Az $RT(\mathcal{A} \oplus z, \hat{\phi})$ régió-átmeneti rendszert a 4.15. ábra szemlélteti.



4.15. ábra. Az $RT(\mathcal{A}, \phi)$ régió-átmeneti rendszer

4. lépés: A TCTL modell-ellenőrzés algoritmusa.

Legyen $\mathcal{A}$ egy idő-divergens időzített automata és ϕ egy TCTL formula. Belátható, hogy $\mathcal{A} \models \phi$ a TCTL szemantika szerint, akkor és csak akkor teljesül, ha $RT(\mathcal{A}, \phi) \models \hat{\phi}$ a CTL szemantika szerint.

Így azt a kérdést, hogy vajon az $\mathcal{A}$ időzített automata kielégíti-e a ϕ TCTL formulát, visszavezetjük annak vizsgálatára, hogy az $RT(\mathcal{A}, \phi)$ kielégíti-e a $\hat{\phi}$ AP' feletti CTL formulát, ami a CTL modell-ellenőrzési algoritmus alapján megválaszolható.

A következőkben egy olyan TCTL modell-ellenőrzési algoritmust vázolunk fel, melyben a ϕ formulában levő időparaméteres operátorokból az időparaméter eliminálása az algoritmus végrehajtása során történik. Így nem szükséges minden egyes időparaméteres operátornál az időparaméter eliminálásához külön-külön órákat alkalmazni, elegendő egy új z óra bevezetése. Az $RT(\mathcal{A} \oplus z, \phi)$ régió-átmeneti rendszer megkonstruálásának alapja az $\mathcal{A} \oplus z$ időzített automata és az $\mathcal{AB}(\phi)$, ahol most az $\mathcal{AB}(\phi)$ egyrészt tartalmazza azokat az atomi órafeltételeket, melyek a ϕ-ben részformulaként szerepelnek, másrészt szintén tartalmazza az összes $(z \in J)$ alakú atomi órafeltételt, ahol J a ϕ-ben előforduló valamely operátor időparamétere.

Az alábbi modell-ellenőrzési algoritmusban a következő jelöléseket alkalmazzuk:
$S_R(\psi) = \{s \in S \mid R, s \models \psi\}$, ahol S az állapothalmaz az R régió-átmenet rendszerben.
$Sub(\phi)$ a ϕ részformuláinak halmazát jelöli.
Az algoritmus végrehajtása során minden $s \in S$ állapotra $Lab(s)$ egyrészt tartalmazza a $\rho'(s)$ elemeit, továbbá azokat az a_ψ CTL formulákat, melyek a ψ-ből az operátor időparamétereinek eliminálásával állnak elő, és melyeket az algoritmus már megvizsgált, továbbá rájuk $R, s \models \psi$ teljesül.
$S_{\mathrm{CTL}}(\psi)$ azt jelöli, hogy a CTL modell-ellenőrzési módszer alapján határozzuk meg a ψ-t kielégítő S-beli állapotok halmazát.

A TCTL modell-ellenőrzési algoritmus váza:

Bemenet: $\mathcal{A}$ idő-divergens időzített automata és ϕ TCTL formula.
Kimenet: "igen", ha $\mathcal{A} \models \phi$, egyébként "nem".
$R := RT(\mathcal{A} \oplus z, \phi)$;
forall $i \leq |\phi|$ do
  forall $\psi \in Sub(\phi)$ és $|\psi| = i$ do
    switch(ψ):
      1: $S_R(\psi) := S$;
      a: $S_R(\psi) := \{s \in S \mid a \in Lab(s)\}$;
      $\psi_1 \wedge \psi_2$: $S_R(\psi) := \{s \in S \mid \{a_{\psi_1}, a_{\psi_2}\} \subseteq Lab(s)\}$;
      $\neg\psi'$: $S_R(\psi) := \{s \in S \mid a_{\psi'} \notin Lab(s)\}$;
      $\mathbf{EU}_J(\psi_1, \psi_2)$: $S_R(\psi) := S_{\mathrm{CTL}}(\mathbf{EU}((a_{\psi_1} \vee a_{\psi_2}), (z \in J) \wedge a_{\psi_2}))$;
      $\mathbf{AU}_J(\psi_1, \psi_2)$: $S_R(\psi) := S_{\mathrm{CTL}}(\mathbf{AU}((a_{\psi_1} \vee a_{\psi_2}), (z \in J) \wedge a_{\psi_2}))$;
    endswitch
    forall $s \in S$ és $s\{z := 0\} \in S_R(\psi)$ do
      $Lab(s) := Lab(s) \cup \{a_\psi\}$;
    od
  od
od
if $S_0 \subseteq S_R(\phi)$ then return "igen" else return "nem" fi

Mivel a TCTL modell-ellenőrzés a CTL modell-ellenőrzést alkalmazza, így az $\mathcal{A} \models \phi$ eldönthető $\mathcal{O}((|S| + |T|) \cdot |\phi|)$ időben, ahol S és T az $RT(\mathcal{A}, \phi)$ régió-átmeneti rendszer állapothalmaza, illetve átmenethalmaza. Viszont a régió-átmeneti rendszer mérete az órák számában és a ϕ formulában szereplő órafeltételekben megjelenő maximális konstansokban exponenciális. A modell-ellenőrzés időigénye tehát javítható az időzített automatának a régió-átmeneti rendszernél kisebb méretű reprezentálásával. Végül megemlítjük a részletes tárgyalás nélkül, hogy bizonyos konfiguráció-régiók úgynevezett *zónákká* vonhatók össze anélkül, hogy a modell-ellenőrzés helyességét elrontanánk. Például a 4.15. ábrán látható $RT(\mathcal{A}, \phi)$ régió-átmeneti rendszerben az $\langle 1,\ [x = 0,\ z = 0]\rangle$ és az $\langle 1,\ [0 < x < 1,\ 0 < z < 1,\ x = z]\rangle$ konfiguráció-régiók összevonhatók egy $\langle 1,\ [0 \leq x < 1,\ 0 \leq z < 1]\rangle$ zónába.

5. fejezet

A modell-ellenőrzés gyakorlata

Ebben a fejezetben két modell-ellenőrző szoftver rendszert mutatunk be, nevezetesen a **SPIN** [48] és az **UPPAAL** [50] programokat. Célunk példák segítségével bemutatni, hogyan lehet egyidejű (konkurens), többszálú, valamint időzített rendszereket ezekkel a szoftverekkel modellezni, velük formális specifikációkat megfogalmazni, és azokat ellenőrizni.

5.1. A SPIN modell-ellenőrző rendszer

A **SPIN** modell-ellenőrző rendszer egy széles körben alkalmazott program osztott szoftver rendszerek időzítéstől független tulajdonságainak formális, automatikus ellenőrzésére.

5.1.1. A SPIN fejlesztése és alkalmazásai

A **SPIN** modell-ellenőrző rendszert a 80-as és 90-es években fejlesztették ki a Bell Laboratories-ban (USA) Gerard J. Holzmann vezetésével [21]. A program 1991-től szabadon elérhető a `http://spinroot.com` ([48]) weboldalon. Az eszköz fejlesztése ma is folyamatos, 2011 áprilisában a 6.0-ás verzió a legfrissebb változat.

A **SPIN** használatát megkönnyítendő több grafikus felhasználói felület is született. Ilyenek a Bell Laboratories-ben készült **XSpin**, az ezt helyettesítő, újabb **iSpin**, és a Mordechai Ben-Ari által készített **jSpin** [31].

2002 áprilisában e program nyerte el az ACM (Association for Computing Machinery) évenként odaítélt, igen rangos Szoftver Rendszer Díját, a Software System Award-ot. Ezzel a kitüntetéssel korábban olyan rendszereket jutalmaztak, mint a Unix, TEX, TCP/IP, Tcl/Tk és a Java.

A **SPIN**-t sok esetben sikeresen alkalmazták valós ipari fejlesztésekben. Például a 90-es évek végén a **SPIN**-nel verifikáltak egy új árvízvédelmi rendszert vezérlő algoritmust Hollandiában, Rotterdam mellett [23]. 1999 és 2001 között használták a Bell Laboratories-ben a PathStar telefonközpont hívás feldolgozó szoftverének ellenőrzésére [22]. Valamint egyre nagyobb szerepet kap a NASA űrkutatási projektjeiben használt missziókritikus szoftverek egyes kulcsfontosságú algoritmusainak ellenőrzésében is [19].

A **SPIN** használatához az alapvető irodalom a G. J. Holzmann által írt felhasználói kézikönyv [21] és az M. Ben-Ari által készített bevezető jellegű tankönyv [7]. Sokszor nélkülözhetetlenek az online elérhető felhasználói kézikönyv oldalak [49] is.

5.2. A Promela modellező nyelv

A **SPIN** modell-ellenőrző konkurens rendszerek, különösen kommunikációs protokollok logikai konzisztenciájának elemzésére készült. A rendszert a Promela (Process Meta Language) modellező nyelven adhatjuk meg, mely kifejezetten a modellezést szolgálja olyan eszközökkel, mint a dinamikus processzus-létrehozás, nemdeterminisztikus vezérlési szerkezetek. A nyelv számos lehetőséget biztosít a processzusok kommunikációjára : támogatja a csatornán keresztül történő szinkron (randevú) és aszinkron (pufferelt) üzenetátadást.

A Promela sok tekintetben a C programozási nyelvhez hasonlít. A C-ből veszi át többek között a Boole és aritmetikai operátorok szintaxisát, az értékadást (egyetlen egyenlőségjel), az egyenlőség tesztelését (dupla egyenlőségjel), a változó és paraméter deklarációkat, a változók kezdeti értékadását, a megjegyzéseket, és azt, hogy az utasítás-blokkokat kapcsos zárójelek közé kell tenni.

Ugyanakkor a Promela modellező és nem programozási nyelv. Így hiányoznak belőle olyan, a programozási nyelvekben megszokott jellemzők, mint a függvények visszatérési értéke, kifejezések mellékhatásai, adat és függvény pointerek, stb. Ennek oka egyszerű : a Promela nem általános célú programozásra, hanem magas szintű verifikációs modellek konstruálására készült.

A Promela nyelvben készült modellek az alábbi három alap objektum típusból épülnek fel :

- processzusok (processes, folyamatok),
- adat objektumok (data objects, pl. változók),
- üzenet csatornák (message channels).

A processzusok mindig globális objektumok, az üzenet csatornák és a változók lehetnek globálisak vagy lokálisak. A processzusok a rendszer viselkedését specifikálják, a csatornák és a globális változók pedig azt a környezetet, melyben a processzusok futnak. A továbbiakban ezeket az alap objektumokat tekintjük át.

5.2.1. Processzusok

A Promela modellek konkurens, azaz egyidejűleg futó processzusokból épülnek fel. Minden tevékenységet a processzusok végeznek, például csak egy processzus képes egy változó értékét megváltoztatni, vagy üzenetet küldeni.

A processzusok *processzus típusok* (proctypes) példányosításával jönnek létre. A processzus típusokat a `proctype` kulcsszóval deklarálhatjuk. Ha nem csak deklarálni szeretnénk egy processzus típust, hanem azt rögtön példányosítani is akarjuk, azaz egyből létre is akarunk hozni egy ilyen típusú processzust, akkor ezt az `active proctype` kulcsszóval kell

megtennünk. Például a klasszikus, „Szia világ!" (Hello world) program a Promela nyelven így néz ki:

```
active proctype main()
{
    printf("Hello world\n")
}
```

Ha ezt a pár sort egy `hello.pml` elnevezésű fájlba mentjük, majd a **SPIN** programnak bemenetül adjuk, a következőt láthatjuk ($-t feltételezve prompt karakternek):

```
$ spin hello.pml
Hello world
1 process created
```

A **SPIN**, ha paraméterek nélkül hívjuk meg, a bemenetként kapott modellen szimulációt végez. A szimuláció általában véletlen választások sorozatát jelenti, de ebben az egyszerű példában nincsenek választási lehetőségek, minden determinisztikus. A véletlen szimuláció fogalmát később fogjuk megismerni. A példában a szimuláció során a **SPIN** létrehozott egy `main` nevű processzust, majd végrehajtotta annak egyetlen `print` utasítását és a processzus ezzel befejeződött. A "`1 process created`" üzenet arra is emlékeztet, hogy modellező nyelvről van szó, az üzenet azt jelenti, hogy a szimuláció során egyetlen processzus keletkezett. Bonyolultabb modellekben általában több processzus lesz.

A fájl `.pml` kiterjesztése nem követelmény, bármilyen kiterjesztésű, vagy kiterjesztés nélküli is lehet az a fájl, amit a **SPIN**-nek bemenetként adunk. A `printf` utasítás szinte megegyezik a C-beli megfelelőjével. A legtöbb standard konverziós formátumot ismeri, mint a `%c`, `%d`, `%u`, `%o` és a `%x`, és a speciális karaktereknek is, mint a `\t` és `\n`, megegyezik a jelentésük a C-ben megszokottal.

Ha csak egyetlen processzusunk van, nem szükséges neki nevet adni. A speciális `init` kulcsszóval létrehozhatjuk a kezdeti processzust.

```
init{
        printf("Hello world\n")
    }
```

az előbbivel egyező eredményt ad. Ez azért fontos, mert ha nincs `active` kulcsszóval deklarált processzus, akkor kezdetben csak az `init` processzus létezik. Ezt kényelmesen használhatjuk arra, hogy a többi processzust, akár meghatározott sorrendben, elindítsuk a `run` utasítással. Ezzel a technikával a programunk így nézne ki:

```
proctype main()
{
    printf("Hello world\n")
}
init { run main() }
```

Futása pedig, lépésenként lekérdezve a szimulációt a `-p` opcióval, az alábbit adná:

Típus	tipikus értelmezési tartomány
`bit` vagy `bool`	0, 1
`byte`	0…255
`short`	$-2^{15}\ldots 2^{15}-1$
`int`	$-2^{31}\ldots 2^{31}-1$
`unsigned :n`	$0\ldots 2^{n}-1$
`chan`	1…255
`mtype`	1…255
`pid`	0…255

5.1. táblázat. A **SPIN** *alap adattípusai*

```
  $spin -p hello3.pml
  0:    proc  - (:root:) creates proc  0 (:init:)
Starting main with pid 1
  1:    proc  0 (:init:) creates proc  1 (main)
  1:    proc  0 (:init:) hello3.pml:5 (state 1) [(run main())]
          Hello world
  2:    proc  1 (main) hello3.pml:3 (state 1)   [printf('Hello world\\n')]
  2:    proc  1 (main)           terminates
  2:    proc  0 (:init:)       terminates
2 processes created
```

A kimenetből azt is láthatjuk, hogy minden processzus rendelkezik egy egyedi processzus azonosító számmal, ez egy 0 és 254 közötti egész szám, a neve *pid*. Példánkban az `init` processzus pid száma 0, a `main` létrehozott pédányáé pedig 1. A pid értékeket mindig növekvő sorrendben osztja ki a rendszer, az első processzusé 0, a másodikként elindítotté 1, stb. A létrehozott processzus azonnal megkezdheti, de nem feltétlenül kezdi meg futását. A processzusok csak keletkezésükkel ellentétes sorrendben halhatnak ki. Ha a processzus végrehajtotta utolsó utasítását, akkor véget ér (terminál), de kihalni csak akkor tud, ha már az összes utána létrehozott processzus kihalt. Amennyiben egy processzus kihal, a pid száma felszabadul és az újra kiadható.

5.2.2. Adat objektumok

A **SPIN**-ben használt adattípusok függnek a használt operációs rendszertől, pontosabban magának a **SPIN** programnak a fordításához használt, illetve a verifikációhoz alkalmazott C fordítótól öröklődnek. Az 5.1 ábrán a **SPIN** alap adattípusait soroltuk fel, a tipikus értelmezési tartományokkal egy 32-bites szóhosszal rendelkező rendszert feltételezve.

A `bit` és a `bool` szinonimák, egy bitnyi információ tárolására alkalmasak. A 0 és 1 értékek megadására használhatjuk a `true` és `false` előre definiált konstansokat is. A `byte`, `short` és az `int` a szokásos, egyre nagyobb értelmezési tartományú egész értékű változók. Az `unsigned` típus esetén mi definiálhatjuk, hogy hány biten kívánjuk ábrázolni a változó értékét 1 és 32 között. Például

```
unsigned x : 5 = 17
```

egy olyan x előjel nélküli változót definiál, melyet 5 biten ábrázolunk, kezdeti értéke pedig 17 lesz. A **SPIN** a kifejezéseket mindig a legbővebb, int típusú egész számokra értékeli ki. Azonban értékadáskor és üzenetátadáskor, ha túlságosan nagy a tárolni kívánt érték a változó értékkészletéhez képest, akkor csonkolás történik. Szimuláció során erre a **SPIN** figyelmeztet, de ez nem okoz kivételt. Verifikációs módban viszont csak a végrehajthatatlan típushibákra kapunk hibaüzenetet.

Az operátorok szintén a C nyelvből öröklődtek. Néhány fontos különbséget azért megfigyelhetünk. A legfontosabb, hogy a Promelában a kifejezések kiértékelésének mindig mellékhatás mentesnek kell lennie. Ez azért van, mert kifejezéseket használhatunk egy-egy utasítás végrehajthatóságának tesztelésére. Márpedig, ha nem hajthatjuk végre az adott utasítást, akkor a feltételül szabott kifejezés kiértékelésének nem szeretnénk, hogy bármi nyoma is maradjon. Ebből következik, hogy a Promelában az értékadás nem kifejezés, nincs visszatérési értéke, és a ++ növelő és -- csökkentő operátorokat csak postfix módon alkalmazhatjuk, így: b++, a prefix használat, azaz ++b, nem megengedett.

Végül a chan, mtype és pid speciális, bájt értékekkel reprezentált típusok, melyeket rendre a csatornák, üzenet típusok és pid azonosítók megkülönböztetésére használunk. A csatornákról később lesz szó, a pid azonosítókat pedig már említettük.

Az mtype típusú változók arra szolgálnak, hogy könnyen megjegyezhető szimbolikus értékeket is használhassunk. A kifejezés a message type (üzenet típus) rövidítése, ami arra utal, hogy a protokollokban az üzenetekre ne számokkal hanem szimbolikus nevekkel tudjunk hivatkozni. Persze nem csak az üzeneteknek, hanem bármi másnak adhatunk szimbolikus nevet. Például elnevezhetjük egy közlekedési lámpa állapotait a következő módon

```
mtype = { red, yellow, green }
mtype light = green
```

Azaz először deklaráltunk három szimbolikus értéket, red, yellow és green néven, majd deklaráltunk egy light mtype típusú változót, melynek a green kezdőértéket adtuk. Persze a **SPIN** az mtype típusú változók értékét egész számokkal reprezentálja. Összesen 255 különböző szimbolikus nevet használhatunk. A szimbolikus nevet a %e-vel, vagy a printm függvénnyel lehet kiíratni.

A **SPIN**-ben tömböket és összetett adattípusokat is használhatunk. Például

```
typedef Record {
           byte   a = 3;
           short  b
        };
Record r;
byte t[10] = 5;
init{  r.b = t[2] + r.a;  printf("r.b=%d",r.b)  }
```

futtatásának eredménye r.b=8 lesz. Ugyanis a byte t[10] = 5 a 10 elemű t tömb minden elemét 5-re inicializáltuk, majd ehhez az r rekord a mezőjének 3 értékét adtuk. A összetett adattípusokról és a tömbökről bővebben a hivatkozási kézikönyvben [49] olvashatunk.

5.2.3. Üzenet csatornák

A processzusok egymásnak csatornákon keresztül küldhetnek és fogadhatnak üzenetet. Fontos, hogy más programozási nyelvekkel ellentétben, a Promelában a csatornák globális objektumok, nem csak egy processzuspárhoz kötődnek, hanem bármely processzus küldhet és fogadhat üzenetet bármely csatornán keresztül.

Minden csatornához tartozik egy *üzenet típus* (message type), mely meghatározza, hogy milyen típusú értékek küldhetők a csatornán keresztül. Legfeljebb 255 csatornát hozhatunk létre. A csatorna deklaráció általános alakja a következő:

$$\texttt{chan ch = [capacity] of typename}_1\texttt{, …, typename}_k \quad .$$

Itt `ch` a csatorna neve, `capacity` egy nem negatív egész szám, a csatorna kapacitása, amely nem más mint a csatornában egyszerre elhelyezhető üzenetek maximális száma. Végül $\texttt{typename}_1\texttt{, …, typename}_k$ pedig a csatorna típusa, mely azt mutatja, hogy az üzenet k komponensből állhat, melyeknek rendre $\texttt{typename}_1$, …, $\texttt{typename}_1$ típusúnak kell lenniük.

Például a következő deklarációval: `chan qname = [16] of short, byte, bool` a `qname` nevű csatornát hoztuk létre, mely legfeljebb 16 darab olyan üzenetet tárolhat, mely három komponensből áll.

A csatorna nevéhez a **SPIN** egy 0 és 255 közötti azonosító számot rendel. Megjegyezzük, hogy a csatorna neve lehet globális vagy lokális változó, de maga a csatorna mindig globális objektum.

Az üzenetküldés ebbe a csatornába a `qname!expr1,expr2,expr3` utasítással történik. A `!` utal arra, hogy küldésről van szó. Ha a csatorna nincs tele, akkor ez az utasítás végrehajtható, és hatására az `expr1`, `expr2` és `expr3` kifejezések által meghatározott értékek a csatorna végére kerülnek. Pontosabban a kifejezések értéke a csatorna definíciójában megadott típusokra konvertálódik.

Ha a csatornában már nincs hely, akkor a küldő utasítás nem hajtható végre. Kivéve `-m` opció használata esetén, mert ekkor a küldés mindig végrehajtható, de a tele csatornába küldött üzenet elvész.

Az üzenetfogadás utasítás hasonló, csak felkiáltójel helyett kérdőjelet használunk. A fogadás kétfajta módon történhet: feltétel nélkül, vagy feltételesen.

A feltétel nélküli üzenetfogadás alakja a következő:

```
qname?var1,var2,var3
```

Ez mindig végrehajtható, ha a csatorna nem üres. Hatására a legrégebbi üzenet a felsorolt változókba kerül. Hiba több vagy kevesebb komponenst várni, mint amennyi a csatorna definíciójában szerepelt.

A feltételes üzenetfogadáskor egy vagy több változó helyére konstanst írunk. Például

```
qname?cons1,var1,const2
```

Ez azzal jár, hogy csak az első és a harmadik helyen megadott konstans értéket tartalmazó üzenetet olvashatjuk ki a csatornából. Ha nem ilyen a csatorna legrégebbi üzenete, az utasítás nem hajtható végre. Az utasítás végrehajtásával természetesen `var1` felveszi az üzenetből második komponensként kapott értéket.

Van még egy lehetőségünk: a konstansok helyére írhatunk `eval(`var`)` alakú kifejezést is. Ekkor a var változó értéke lesz az, amit az üzenet megfelelő komponensétől megkövetelünk. A feltételes üzenetfogadással könnyen biztosítható, hogy ha a processzusok közös csatornát használnak is, minden processzus csak a neki szóló üzenetet olvashassa ki a közös csatornából.

Egy alternatív, de ekvivalens szintaxist is használhatunk a küldő és fogadó utasítások megadásakor. Az üzenetek első komponense általában az üzenetet típusának megadására szolgál, és ezért `mtype` típusú. Az üzenetet megadásakor úgy is eljárhatunk, hogy nem vesszővel elválasztva soroljuk fel a komponenseket, hanem az első komponenst névként használva utána argumentumként, zárójelek között adjuk meg a többi komponenst. Így például `qname!expr1, expr2, expr3` helyett írhatunk `qname expr1(expr2,expr3)`-at, és `qname?var1,var2,var3` helyett `qname?var1(var2,var3)`-at, ahol természetesen `expr1`, `expr2`, `expr3`, és `var1`, `var2`, `var3` a csatorna komponenseinek megfelelő típusú kifejezés, illetve változó.

Az alábbi példában a `P` processzus a `csat` csatornán keresztül egy kérést küld, mire egy egész értéket vár egy másik processzustól válaszként. A csatorna üzenet típusának első komponense biztosítja, hogy a kérő és válaszoló üzeneteket meg tudjuk különböztetni.

```
mtype {ker, kuld}
chan csat = [8] of {mtype, int};
int valasz;
active proctype P() {
    csat!ker(0);
    csat?kuld(valasz);
}
```

5.2.4. Szinkron és aszinkron üzenetküldés, randevú csatornák

Amennyiben a csatorna kapacitás nagyobb 0-nál, aszinkron üzenetküldésről van szó. Az üzenetek, amíg férnek, egy pufferbe kerülnek, és a küldő processzus ez után tovább léphet. Ettől függetlenül férhet hozzá a csatornához a fogadó processzus. Az egyetlen szabály, hogy tele csatornába nem tudunk üzenetet küldeni, üres csatornából pedig nem tudunk olvasni.

Ezzel szemben, ha a csatorna kapacitása 0, szinkron üzenetküldésről és randevú csatornáról beszélünk. Ilyenkor az üzenetek nem tárolódnak, a küldő vagy a fogadó processzus kénytelen várakozni mindaddig, amíg egy, az üzenetet fogadni vagy küldeni képes partnert nem talál. Ha ez megvan, a küldés és a fogadás egyszerre megy végbe, ez kézfogás vagy randevú típusú szinkronizáció. Tehát, ha a csatorna kapacitása 0, a küldő utasítás csak akkor hajtható végre, ha van az esetleges konstansoknak is megfelelő fogadó utasítás, ami végrehajtásra vár. Például, ha

```
chan cs = [0] of  bit, byte ,
```

és, ha az egyik processzus ajánlja a `cs!1, 3+7` utasítást, amit egy más processzus el tud fogadni a `cs?1, x` utasítással, akkor a randevú létrejöhet, mert az első paraméter (1) egyezik, és a randevú után `x` értéke 10 lesz.

5.3. A Promela alap utasításai

A Promela 6 alapvető utasítástípusát az 5.2 táblázat foglalja össze.

	típus	példa
1.	értékadás	`x++, x--, x = x+1, x=run P()`
2.	kiírás	`print("x=%d\n", x)`
3.	feltételezés	`assert(x+1 == 2)`
4.	kifejezések	`(x), (1), run P(), skip, true, else, timeout`
5.	üzenetküldés	`q!ack(m)`
6.	üzenetfogadás	`q?ack,2,var`

5.2. táblázat. A **SPIN** *alap utasítás típusai*

Az értékadó utasítások a C nyelvben megszokottak. Az `x=run P()` azon túl, hogy a `P` típusú processzus egy újabb példányát elindítja, az `x` változóba az új processzus azonosítóját teszi. Kivéve, ha már 254 processzus fut és nincs lehetőség egy újabb elindítására.

A `print` utasítás nem szorul különösebb magyarázatra, csak szimuláció során van hatása, verifikációkor a **SPIN** figyelmen kívül hagyja.

A feltételezések (assertions) az ellenőrzés legalapvetőbb eszközei. Egyszerűen hibát jelent, ha az `assert` utasításban szereplő kifejezés hamis, azaz 0 értéket vesz fel. Ez az egyetlen verifikációs eszköz, ami még szimulációs módban is működik.

A kifejezések, a programozási nyelvektől eltérő módon, önmagukban is teljes értékű utasítások. A Promela lényeges vonása, hogy az utasítások nem mindig hajthatók végre. Egy kifejezés pontosan akkor hajtható végre, ha kiértékelve igaz értéket ad. A kifejezés mint utasítás hatása abban áll, hogy ha a feltételt sikerül "végrehajtanunk", akkor legközelebb a feltételt követő utasításra kerülhet a vezérlés. Ellenben, ha a feltétel hamis, addig nem léphetünk tovább, míg igazzá nem válik.

A két utolsó utasítástípust: az üzenetküldést és az üzenetfogadást már említettük.

Összefoglalva, az első három utasítás típus mindig végrehajtható, a kifejezések akkor, ha igaz, azaz nem nulla az értékük, az üzenetküldés, akkor, ha nem tele a csatorna, melybe üzenetet akarunk küldeni. Kivéve az `-m` opció használatakor, mely engedélyezi a tele csatornába történő üzenetküldés végrehajtását, de ekkor az üzenet elvész. Az üzenetfogadás pedig általában akkor hajtható végre, ha van a csatornában üzenet, és a legrégebbi üzenet illeszkedik az esetleges konstansokhoz. Kivételt képez a szinkron üzenetfogadás, mely csak szinkron üzenetküldéssel hajtható végre egyidejűleg.

5.4. Összetett utasítások

A processzusok utasításai általában szekvenciálisan, sorra egymás után hajtódnak végre. Kivéve a `goto` és a `break` parancsokat, bár ezek igazából nem önálló utasítások, csak a következő vezérlési pontot jelzik, valamint az ebben a részben ismertetésre kerülő összetett utasításokat. Ide tartoznak a nemdeterminisztikus választások (`if..fi`, `do..od`;) és a megszakíthatatlan lépések (`atomic{...}`, `d_step{...}`).

5.4.1. Esetválasztás

Az esetválasztás általános alakja a következő:

```
if
::őrfeltétel1 -> ut11;ut12;ut13;...
::őrfeltétel2 -> ut21;ut22;ut23;...
...
::őrfeltételn -> utn1;utn2;utn3;...
fi
```

Az utasítás akkor hajtható végre, ha van igaz értékű az őrfeltételek közt. Ha több is van, akkor a **SPIN** nemdeterminisztikusan választ közülük, ha egy sincs, az utasítás blokkol. Őrfeltétel bármilyen alap- vagy összetett utasítás lehet, bár leggyakrabban kifejezéseket használunk e célra. A `->` (nyíl) jelölést csak a modellek olvashatóságának megkönnyítésére szoktuk használni, írhatnánk helyére pontosvesszőt is. A két utasítást elválasztó jel ekvivalens.

Az alábbi programrészlettel `x` és `y` maximumát tehetjük az `m` változóba.

```
if
:: x >= y -> m = x
:: x <= y -> m = y
fi
```

Amennyiben `x` és `y` egyenlő, a **SPIN** nemdeterminisztikusan választ a két ág közül, ami most `m` értékének szempontjából nem jelent különbséget.

5.4.2. Ciklikus esetválasztás

A ciklikus esetválasztás általános alakja nagyon hasonló az előzőhöz, mindössze az `if-fi` párt kell `do-od`-ra cserélni.

```
do
::őrfeltétel1 -> ut11;ut12;ut13;...
::őrfeltétel2 -> ut21;ut22;ut23;...
...
::őrfeltételn -> utn1;utn2;utn3;...
od
```

Az előzőhöz képest a különbség csupán annyi, hogy míg az `if-fi` esetválasztás csak egyszer hajtódik végre, a `do-od` elágazásai végtelen sokáig ismétlődnek. Ebből a végtelen ismétlődésből csak a `break` vagy `goto` utasítással léphetünk ki, mely a következő utasításra, illetve a `goto` után álló címkére adja a vezérlést. Például az alábbi processzus `x` és `y` legnagyobb közös osztóját adja:

```
proctype Euclid(int x, y)
{
do
```

```
:: (x >  y) -> x = x - y
:: (x <  y) -> y = y - x
:: (x == y) -> goto DONE
od;
  DONE: printf("GCD: %d",x);
}
```

Mivel a `done` címke közvetlenül a ciklikus esetválasztás után következik, `goto DONE` helyett használhattuk volna a `break` utasítást is.

5.4.3. `Else` és `timeout`

Van két speciális őrfeltétel, az `else` és a `timeout`, melyeket mind a két esetválasztásban használhatunk. Az `else`, mint őrfeltétel pontosan akkor igaz, ha a processzusban, melyben szerepel, az összes többi ág őrfeltétele hamis. Ezzel szemben a `timeout` akkor hajtható végre, ha az egész rendszerben, azaz a többi processzusban sincs más végrehajtható utasítás. Tehát az `else` a processzusra nézve lokális, a `timeout` pedig az egész modellre vonatkozik, vagyis globális. A `timeout` segítségével könnyen elkerülhetjük, hogy a rendszerben holtpont alakuljon ki. Például, az alábbi processzus egy üzenetre vár a `q` csatornán keresztül. Ám, ha a végrehajtás során olyan szituáció alakul ki, hogy azt sohasem kapná meg, mert a küldő processzus is blokkolt, akkor a `timeout` ágon ki tud lépni a ciklusból:

```
do
:: q?message -> ...
:: timeout -> break
od
```

Megjegyezzük, hogy a `timeout` és az `else` tulajdonképpen belső, csak olvasható logikai változóknak is tekinthetők, melyek annak megfelelően vesznek fel értéket, hogy processzus, illetve modell szinten van-e végrehajtható utasítás. Azaz hamisak mindaddig, amíg van más végrehajtható utasítás, és ekkor nem léphetők át. Ezért állhatnak magukban is, nincsenek feltétlenül esetválasztáshoz kötve. Ennek persze csak `timeout` esetében van értelme, hiszen ha az `else` nem egy esetválasztás őrfeltétele, akkor mindig végrehajtható, így helyettesíthető az üres `skip` utasítással.

Egy dologra azonban még ügyelnünk kell, bármely vezérlési ponton együttesen is legfeljebb egy `else` vagy `timeout` ág szerepelhet. Például az alábbi modell részlet hibás:

```
A: do
   :: if
      :: x > 0 -> x--
      :: else  -> break
      fi
   :: else -> x = 10
   od
```

A hiba oka, hogy a két `else` utasítást ugyanazon (az `A` címkével jelölt) vezérlési ponton kellene kiértékelni.

5.4.4. Megszakíthatatlan (atomi) lépések

A megszakíthatatlan vagy atomi lépések alakja a következő:

```
atomic { őrfeltétel -> ut1; ut2; ...; utn }
```

Az értelme az, hogy amíg az összes utasítás végre nem hajtódik, más processzus nem léphet. Pontosan akkor hajtható végre, ha az őrfeltétel igaz.

A megszakíthatatlan lépések utasításai közt lehetnek nemdeterminisztikus utasítások. De, ha valamelyik belső utasítás blokkol, akkor a vezérlést megkaphatja más processzus, vagyis a „megszakíthatatlanság” megszűnik. Később, ha a blokkolás megszűnik, akkor (nem feltétlenül azonnal) megszakítás nélkül folytatódik a blokk.

Az atomi lépések alkalmazásával elkerülhető a klasszikus tesztelés és értékadás (test and set) probléma. Azzal, hogy egy változó értékét atomi lépéseken keresztül olvassuk majd változtatjuk meg, megakadályozhatjuk a többi processzust abban, hogy az olvasás után, de még az értékadás előtt a változóhoz hozzáférjenek. Például:

```
atomic{ (mutex==free) -> mutex=busy }
```

Vagy például sokszor atomi lépésekben célszerű két változó értékét megcserélni:

```
atomic { tmp = b, b = a, a = tmp }
```

Továbbá így érhetjük el azt, hogy két processzust egyszerre indítsunk el, azaz az első ne kezdhesse meg futását, míg a második létre nem jött:

```
init { atomic { run A(1,2); run B(3,4) } }
```

A másik nagyon hasonló konstrukció a determinisztikus lépések, vagyis a `d_step` utasítás. Alakja:

```
d_step { őrfeltétel -> ut1; ut2; ...; utn }
```

A különbség az atomi lépésekkel szemben az, hogy determinisztikusnak kell lennie. Egész pontosan, lehetnek ugyan benne nemdeterminisztikus utasítások, de ha nemdeterminisztikusság áll elő, akkor azt fix módon oldja fel a rendszer, például mindig a legelső lehetőséget választja. De hogy pontosan hogyan választ a **SPIN**, az nincs definiálva, nem lehet rá építeni.

Mivel biztosan megszakítás nélkül hajtódik végre, nem lehetnek az őrfeltételen kívül blokkoló utasításai, továbbá tilos bele vagy belőle goto-val be- vagy kiugrani.

Segítségével nagy mértékben csökkenthető a modell állapotainak a száma, de ügyelnünk kell a használatára. Verifikációkor a `d_step` utasítás lépéseinek végrehajtása közben a rendszer nem végez mentést és ellenőrzést, ha végtelen ciklusba kerülünk egy `d_step` utasítás sorozaton belül, a modell-ellenőrzővel is ez történik.

5.5. Verifikáció a SPIN segítségével

Az előző fejezetben a modellek megadását vizsgáltuk a **SPIN**-ben, most az ellenőrizendő specifikáció megadásának lehetőségeit ismertetjük.

A hatékonysága érdekében a verifikáció végrehajtása a következőképpen zajlik. Először a **SPIN**-nel a modellből a `-a` opcióval egy C forrásnyelvű programot kell generálni, mely a modell-ellenőrzést a megadott speciális feladatra végre fogja hajtani. Ennek a programnak a neve `pan` (Process Analyser). Ezt követően a generál forrást egy C fordítóval le kell fordítanunk, majd futtatnunk. Például a `pelda.pml` modellt a következőképpen verifikálhatjuk, ha `gcc` a rendszerben használatos C fordító:

```
$ spin -a pelda.pml
$ gcc -o pan pan.c
$ ./pan
```

A **SPIN** a következő ellenőrzési lehetőségeket nyújtja:

- holtpont mentesség (invalid endstates),
- feltételezések (assertions),
- elérhetetlen kód (dead code),
- üresjárati ciklusok (non-progress cycles, livelocks),
- tetszőleges LTL logikában felírt tulajdonságok.

Persze a **SPIN** nem tudja, mi a „jó” és „rossz” tulajdonság, a modell-ellenőrzés szempontjából csak azt mondhatjuk egy rendszerről, hogy mi az, ami lehetséges, és mi az, ami nem.

Az ellenőrizendő tulajdonságokat a negyedik fejezetben ismertetett két nagy kategóriába sorolhatjuk. *Állapot tulajdonságok:* elérhető vagy elérhetetlen egy adott tulajdonságú állapot; és *út tulajdonságok:* lehetséges vagy lehetetlen adott tulajdonságú (véges vagy végtelen) végrehajtási sorozat. A helyességi kritériumok megadására az alábbi eszközök szolgálnak.

- Állapot tulajdonságok:
  - alapfeltételezések (basic assertions),
  - végállapot-címkék (end-state labels).
- Út tulajdonságok:
  - előrehaladási címkék (progress-state labels),
  - elfogadási címkék (accept-state labels),
  - soha-állítások (never claims)
    (ezek automatikusan generálhatók LTL formulákból),
  - út-feltételezések (trace assertions).

A továbbiakban röviden ezeket ismertetjük.

5.5.1. Alapfeltételezések

A verifikáció legegyszerűbb módja az alapfeltételezések használata. Segítségükkel azt a helyességi kritériumot fejezzük ki, hogy egy kifejezésnek a végrehajtás pillanatában igaznak kell lennie. Ez a következő formában történik:

`assert(` *kifejezés* `).`

Fentebb *kifejezés* tetszőleges, a Promela nyelvben szabályos, mellékhatás mentes kifejezés. Az utasítás végrehajtásakor a kifejezés kiértékelődik, és ha értéke 0, hibát kapunk. Ezzel kényelmesen ellenőrizhetünk egyszerű biztonsági tulajdonságokat a modell különböző pontjain. Ha pedig a rendszer egészére nézve valamilyen invariáns tulajdonságot szeretnénk ellenőrizni, azt egy monitor processzus segítségével célszerű megtennünk:

```
active proctype monitor()
{
atomic { !invariant -> assert(false) }
}
```

Az `assert` utasítás hasonló a `skip` üres utasításhoz abban, hogy mindig végrehajtható, és a rendszer állapotát nem változtatja meg, kivéve persze, hogy az utasítást végrehajtó processzus vezérlési pontját az `assert` utasítás utánra helyezi.

Az alapfeltételezések az egyetlen olyan verifikációs eszköz, mely már szimulációs módban is működik. Ellenőrzésük a **SPIN** `-A` opciójával kapcsolható ki.

5.5.2. Címkék

Bármely utasítás elé írható (egy vagy több) címke, mely az utasítás végrehajtása előtti vezérlési pontot azonosítja. A címkék kis és nagybetűből, valamint számjegyekből és aláhúzás karakterekből állhatnak, de nem kezdődhetnek aláhúzással. A címkék a processzustípuson belül egyediek kell, hogy legyenek.

A címkék nem csak a `goto` utasítások céljainak megadásához szükségesek, hanem három típusuk az ellenőrzést is szolgálja, ezek az

- `end..` kezdetű végállapot-címkék, a
- `progress..` kezdetű előrehaladási címkék, és az
- `accept..` kezdetű elfogadási címkék.

Mivel a modell értelmezésekor az utasításokból a processzus automatájának átmenetei lesznek, ezért az utasítások elé írt címke mindig azt az állapotot jelöli, ahonnan az utasításhoz rendelt átmenet kiindul. Ez esetválasztásnál jelentős, mert így a végrehajtási ágak elé írt címkék ugyanazt a vezérlési pontot jelölik.

5.5.3. Végállapot címkék

Végállapot címkéknek nevezzük az `end` kezdetű címkéket. Például `end`, `end0`, `end_State`.

A rendszer végállapotba ér, ha már egyik processzusnak sincs végrehajtható utasítása, azaz minden processzus véget ért vagy blokkol. Alapértelmezés szerint a rendszernek csak azok a végállapotai megengedettek, melyekben minden processzus befejeződött, vagyis elérte a záró kapcsos zárójelet. Különben holtpont (deadlock) állt elő. Erre a verifikáció *invalid endstate* (hibás végállapot) jelzéssel figyelmeztet.

De ez sokszor nem valódi hiba. Általában megengedhetjük, hogy néhány végtelen ciklusban működő processzus várakozó állapotban maradjon. Erre megoldást a végállapot címkék jelentenek. Velük mi mondhatjuk meg, hogy mely állapotban való várakozás megengedett, azaz nem okoz holtpontot. Persze ettől még a rendszer kerülhet úgy holtpont állapotba, hogy az egyik processzus végállapot címkénél áll, de ekkor valamelyik másik processzus akadt el nem végállapotban.

Az alábbi példa egy tipikus helyzetet mutat be a végállapot címkék használatára:

```
     mtype { p, v };
     chan sema = [0] of { mtype };
     active [3] proctype user()
     {
          sema?p;                /* enter */
crit:     skip;                 /* leave */
          sema!v;
     }
     active proctype semaphore()
     {
     byte count = 1;
            do
            :: (count == 1) ->
end:              sema!p; count = 0
            :: (count == 0) ->
                  sema?v; count = 1
            od
     }
```

A példában három `user` típusú processzust láthatunk, melyek közt a kölcsönös kizárást a `semaphore` nevű processzus valósítja meg a `sema` csatornán keresztül szinkronizáció segítségével. Ha az `end` végállapot címkét nem használnánk, akkor, miután a `user` processzusok lefutottak, *invalid endstate* hibajelzést kapnánk, hiszen a `semaphore` processzus az `end` címkével jelzett utasítása blokkolna. Ez elkerülhető a végállapot címke használatával, hiszen a megfigyelt viselkedés nem hibás, a `semaphore` processzust eleve végtelen ciklusban futónak terveztük. Ugyanakkor vegyük észre, hogy ha a `user` processzust is végtelen ciklussal modelleznénk, akkor a rendszerben egyáltalán nem lenne végállapot, se megengedett, se nem megengedett.

Mint említettük, fontos, hogy esetválasztásnál és ciklikus esetválasztásnál nem valamelyik őrfeltétel elé kell a címkét írni, mert az őrfeltételek átmenetek, melyek a választás kezdőpont-

jából az ágak kezdetére mutatnak. Ehelyett az elágazás előtti állapotot kell megjelölni, azaz az `if` vagy a `do` utasítás elé kell tenni a címkét.

Alapértelmezés szerint a rendszer ellenőrzi a hibás végállapot jelenlétét a modellben. Továbbá a `pan -q` opciójának használata esetén nem csak azt ellenőrizhetjük, hogy megálláskor minden processzus végállapotban van-e, hanem azt is, hogy minden csatorna üres-e. Végül a `-E` opcióval az összes végállapot ellenőrzés kikapcsolható.

5.5.4. Előrehaladási címkék

Az előrehaladási címkék a `progress` szóval kezdődnek. Hasonlóak a végállapot-címkékhez, de azt jelzik, hogy az általuk jelölt állapotban előrehaladás történt, például beléptünk a kritikus állapotba, üzenetet fogadtunk stb.

Valamely előrehaladási címkével jelölt állapotnak mindig végtelen sokszor elő kell fordulni minden végtelen futás során. Tehát, az előrehaladási címkékkel ellenőrzésnél azt kérdezhetjük, hogy van-e olyan „üresjárati” ciklusa a rendszernek, melyben már sohasem történik előrehaladás. Másként fogalmazva, előfordulhat-e, hogy az egész futás alatt csak véges sokszor (a ciklusba lépés előtt) érintünk előrehaladási címkét. Az ilyen nem kívánatos futások hiánya a rendszer egy élőségi tulajdonságát igazolja.

Az üresjárati ciklusok hiánya a C-fordító `-DNP` és a `pan -l` opciójával ellenőrizhető. Technikailag a **SPIN** fordításkor soha-állítást készít a címkékből, és azt ellenőrzi.

Az előrehaladási címkékkel még egy másik trükköt is csinálhatunk: Ha egy nem kívánatos eseményt, pl. üzenetvesztést látunk el progress címkével, és ezután csak „nonprogress” ciklusokat keresünk, akkor a rendszernek csak azon futásához jutunk, melyben a nem kívánatos esemény csak véges sokszor fordul elő. Sokszor ésszerű ezt feltennünk, hogy a rendszernek csak ilyen megszorításokkal kell helyesen működnie, pl. az adatátvitelnek megvalósulnia.

5.5.5. Elfogadási címkék

Az elfogadási címkék az `accept` karaktersorozattal kezdődnek.

Az előrehaladási címkék „szimmetrikus párjai”. Most a kérdés pont fordított: van-e olyan ciklusa a rendszernek, melyben végtelen sokszor halad át elfogadási címkével jelöl állapoton, vagy végtelen sokáig ilyen állapotban tartózkodik.

Ilyen „rossz” vagy csak számunkra valamiért érdekes ciklusok létezését kérdezzük. Önmagukban ritkán használjuk őket, a soha-állításokban kapnak kiemelkedő szerepet. Használatához a `pan -a` opciója kell.

Például, ha az alábbi modellben elfogadó ciklusokat (acceptance cycle) keresünk, akkor csak olyan futásokat vizsgálunk, melyben a `B` folyam végtelen sokszor jut szóhoz, míg az `A` esetleg csak véges sokszor.

```
byte x = 2;
active proctype A()
{
do
:: x = 3 - x
od
```

```
        }

        active proctype B()
        {
        do
        :: x = 3 - x;
accept:    skip
        od
        }
```

5.5.6. Fair ciklusok

A verifikáció esetén gyakran ki akarunk zárni olyan szélsőséges eseteket, melyek azért vezetnek hibához, mert valamelyik processzus egy idő után sohasem került végrehajtásra. Erre szolgál a *pártatlanság* (fairness).

Kétfajta fair tulajdonságot ismerünk:

- *Gyenge fair tulajdonság (Weak fairness)*: Ha egy processzus egy olyan utasításánál áll, amely végtelen sokáig (megszakítás nélkül) végrehajtható, akkor azt az utasítást a processzus előbb utóbb végre is tudja hajtani.
- *Erős fair tulajdonság (Strong fairness)*: Ha egy processzus egy olyan utasításánál áll, amely végtelen sokszor (esetleg megszakításokkal) végrehajtható, akkor azt az utasítást a processzus előbb utóbb végre is tudja hajtani.

A költségnövekedés gyenge fair tulajdonság ellenőrzése esetén lineáris, erős fair tulajdonság esetén kvadratikus szorzó az összes processzus számában. A **SPIN** közvetlenül csak a gyenge fair tulajdonságot támogatja, az erős fair tulajdonság soha-állításokkal valósítható meg.

Használata: `pan -f`. Ekkor a modell-ellenőrző csak gyengén fair végrehajtásokat keres.

5.5.7. Soha állítások

Eddig alapfeltételezésekkel és címkék segítségével megfogalmazott helyességi kritériumokat tekintettünk. Ezek mindig egy-egy processzushoz, és a processzus lépéseinek végrehajtásához kötöttek. Nem tudunk velük olyan tulajdonságokat megfogalmazni, hogy, pl. ha a rendszer egy `p` tulajdonságú állapotban van, akkor előbb-utóbb biztosan eljut egy `q` tulajdonságú állapotba. Ehhez a rendszer futásával szinkronban kell ellenőrzést végezni. Olyan ellenőrzésre van szükségünk, mely a rendszer minden lépésének végrehajtása előtt ellenőrzi, hogy `p`, illetve aztán `q` teljesül-e. Erre szolgálnak a *soha-állítások* (never claims). Például:

```
byte x;
#define p (x==2)
#define q (x==5)
active proctype novel(){
do
```

```
      ::if
      :: (x < 10) ->
            x = x + 2
      :: else -> break;
      fi;
      od;
      };

      never {
         do
         :: true
         :: p -> break
         od;
accept:
         do
         :: !q
         od
         }
```

A soha-állítások speciális processzusok, csak feltétel utasításokat tartalmazhatnak. Ugyanúgy automatákat definiálnak, mint a többi processzus, de nem változtathatják meg a rendszer állapotát, hanem lépésenként megfigyelik (ellenőrzik) azt.

A rendszer minden lépése előtt a soha állítás egy-egy lépése hajtódik végre. Így az 1. 3. 5. stb. lépés a soha-állításé, a 2. 4. 6. a rendszer processzusaié. Amennyiben a rendszer működése véget ért, a soha állítás még önmagában is további lépéseket tehet. Minden modellben egyszerre csak egy soha-állítást tudunk ellenőrizni. De többet is megadhatunk, és az LTL formulákhoz hasonlóan, a verifikáció elindításakor határozhatjuk meg név vagy sorszám alapján, hogy melyiket akarjuk használni.

Az ellenőrzés azt jelenti, hogy keressük a rendszernek olyan végrehajtási sorozatait, melyek a soha állításban megfogalmazott tulajdonságot teljesítik. Ezekre a **SPIN** hibát jelez. Mert soha-állítással pontosan az általa lehetetlennek definiált működéseket keressük. Azaz hiba, ha a soha-állítás véget ér (azaz eléri a záró kapcsos zárójelet) vagy elfogadó állapotot is tartalmazó végtelen ciklusba jut. A soha-állítás nem blokkol, ha nem tud lépni, akkor nincs hibás működés, a rendszer ezen a végrehajtási ágon nem keres tovább hibát, hanem másik ágat próbál.

Példaként bemutatjuk, hogy a nem előrehaladási ciklusok (non progress cycles) létének tiltása hogyan kerül ellenőrzésre egy soha-állítás segítségével.

```
      never { /* non progress cycle detector */
              do
              :: true
              :: np_ -> break
              od;
      accept:
              do
```

```
        :: np_
        od
    }
```

Fentebb `np_` egy csak olvasható Boole változó, mely pontosan akkor igaz, ha egyetlen processzus sincs előrehaladási (progress) állapotban. A bemutatott soha-állítás a `<>[]np_` alakú LTL formulát kielégítő futások keresését jelenti.

5.5.8. LTL formulák

Sokszor nehézkes a rendszertől megkívánt, bonyolult időbeli viszonyokat tükröző helyességi kritériumokat közvetlenül soha-állítással leírni. Ráadásul a kézzel írt soha állítások nem biztos, hogy kompatibilisek a **SPIN** egyik fő optimalizálási technikájával, a részleges rendezési redukcióval (partial order reduction). Ezért sokszor kényelmesebb LTL (lineáris temporális logikai) formulákat használni, melyeket a **SPIN** automatikusan soha-állításokká konvertál és ellenőriz.

A **SPIN** 6.0-ás verziójától kezdve a megkövetelt működést leíró formulákat közvetlenül a modellel együtt is megadhatjuk. A korábbi verziókban ez nem volt lehetséges, hanem erre a `-f` opciót kellett használnunk. Az LTL formulákat a következő alakban adhatjuk meg:

`ltl` *név* { *formula* }

A név megadása nem kötelező, de hasznos, ha több formula helyességét szeretnénk ellenőrizni. Ha névvel látjuk el a formulákat, akkor a `pan` modell-ellenőrzőnek a `-N` opció segítségével mondhatjuk meg név vagy sorszám alapján, hogy melyiket ellenőrizze. Alapértelmezés az első formula verifikációja. A formula lehet egy kifejezés, melyet egy kisbetűvel kezdődő szimbolikus névvel is helyettesíthetünk. Például:

```
#define p     (a > b)
#define q     (len(q) < 5)
#define r     (root@Label)
```

A második formula azt jelenti, hogy a `q` csatornában kevesebb mint 5 üzenet van, a harmadik pedig akkor igaz, ha a `root` processzusnak a következő végrehajtandó utasítása a `Label` címkéjű utasítás. Ez a fajta rövidítés, bár sokszor hasznos, a 6.0-ás verziótól kezdve nem kötelező, magukat a feltételeket is szerepeltethetjük a formulákban.

A formulákban a következő operátorokat használhatjuk:

- Unáris (egyváltozós) operátorok
  - `[]`: `always` (mindig)
  - `<>`: `eventually` (végül is)
  - `!`: `not` (negáció)
- Binér (kétváltozós) operátorok
  - `U`: `strong until` (erős until)

- `W`: `weak until` (gyenge until)
- `V`: `relase` (az U duálisa, azaz `p V q`≡`!(!p U!q)`)
- `&&` vagy `/\`: `and` (és)
- `||` vagy `\/`: `or` (és)
- `->`: implikáció
- `<->`: ekvivalencia

Az operátorok jele helyett az angol nevüket is használhatjuk. Például

```
ltl p1 { []<> p },
ltl p2 { always eventually p }
```

ekvivalens formulák.

5.6. Példák

Ebben a fejezetben két példát mutatunk a jegyzetben korábban szereplő két algoritmus, az alternáló bit protokoll és a Peterson kölcsönös kizárás algoritmus **SPIN**-ben történő verifikációjára.

5.6.1. Az alternáló bit protokoll modellje

A 2.8.2 fejezetben bemutatott alternáló bit protokollnak most csak azt a változatát modellezzük, melyben a vevőtől az adóig hibátlan a kommunikáció, az adótól a vevőig viszont nem. Ennek, a korábban 3. esethez hasonló szituációnak egy lehetséges Promela modellje az alábbi. Lényeges különbség, hogy itt nem szinkron módon zajlik a kommunikáció, hanem egy-egy, két üzenet tárolására alkalmas csatornán keresztül.

```
mtype = { msg, ack };

chan to_sndr = [2] of { mtype, bit };
chan to_rcvr = [2] of { mtype, bit };

active proctype Sender()
{ bool seq_out, seq_in;

/* obtain first message */
do
::  to_rcvr!msg(seq_out) ->
        to_sndr?ack(seq_in);
        if
        :: seq_in == seq_out ->
            /* obtain new message */
```

```
            seq_out = 1 - seq_out;
        :: else
        fi
    od
}

active proctype Receiver()
{ bool seq_in;

do
:: to_rcvr?msg(seq_in) ->
   to_sndr!ack(seq_in)
:: timeout ->/* recover from msg loss */
   to_sndr!ack(seq_in)
od
}
```

5.6.2. A Peterson algoritmus modellje

Másik példánk a 2.3.5 fejezetben ismertetett Peterson algoritmus:

```
       bool turn, d[2];
       byte cnt;

       active [2] proctype P()
       {  pid i, j;

          i = _pid;
          j = 1 - _pid;
again:
          d[i] = true;
          turn = j;
          (!d[j] || turn == i) ->
CS:       cnt++; assert(cnt == 1); cnt--;
          d[i] = false;
          goto again
        }
```

5.7. UPPAAL

Az **UPPAAL** [50] valós idejű rendszerek modellezésére, szimulációjára és verifikációjára szolgáló modell-ellenőrző rendszer, melyben a modellezés kibővített időzített automaták hálózataival történik.

Az eszközt az **Upp**salai Egyetemen (Svédország) és az **Aal**borgi Egyetemen (Dánia) közösen fejlesztették, illetve fejlesztik. A városok nevének összevonásából származik a neve.

Az első verzió 1995-ben jelent meg, jelenleg a 4.0-s a legfrissebb változat. Az eszközt számos ipari projektben alkalmazták sikerrel, és amellett, hogy oktatási célokra továbbra is ingyenes, van kereskedelmi változata is.

Használatához segítséget és irodalmat az [50] hivatalos weboldalakon találunk. Kezdetnek a fejlesztők, G. Behrmann, A. David, és Kim G. Larsen által írt oktatási segédletet [13] ajánljuk, melynek aktuális változata az eszköz weboldaláról letölthető.

Animált ábra. Az 5. animált ábra röviden bemutatja az **UPPAAL** rendszert.

5.7.1. Modellek megadása az UPPAAL-ban

Az **UPPAAL**-ban a modell megadásához kiterjesztett időzített automata hálózatokat használunk, melyeket a 2.9 és 2.10 fejezetben mutattunk be. Emlékeztetőül, ebben a kifejezésben az *időzített* szó arra vonatkozik, hogy az automatákban valós idejű órák használhatók, melyek újraindíthatók és tesztelhetők. A *kiterjesztett* szó arra utal, hogy korlátos egész értékű változókat is használhatunk. Végül a *hálózat* azt jelenti, hogy több automatát tekintünk, melyek csatornákon keresztül szinkronizálnak egymással. A szinkronizáció úgy valósul meg, hogy egy automatában egy `üzenet!` címkéjű (küldő) átmenet csak egy másik automata `üzenet?` (fogadó) átmenetével egyszerre hajtható végre.

Egy **UPPAAL** modell a következő elemekből épül fel:

- globális és lokális deklarációk:
  - konstansok,
  - változók,
  - csatornák,
  - órák,
  - tömbök,
  - rekordok,
  - skalárok,
  - metaváltozók,
  - felhasználói típusok,
- automata sablonok (automata templates),
- rendszer definíció (system definition).

Az automata sablonok tartalmazzák az egyes processzusok modelljeit, ide tartoznak a lokális deklarációk is. A rendszer definíció pedig a komponensek megadásából áll. A komponensek az automata sablonok példányosításaival jönnek létre. Például egy rendszer definíció lehet az alábbi, ahol `lampa` és `embertípus` egy-egy automata sablon, előbbi paraméter nélküli, az utóbbi egy egész típusú paraméterrel rendelkezik:

```
ember1:=embertipus(1000);
ember2:=embertipus(2000);
system lampa, ember1, ember2;
```

5.7.2. Deklarációk

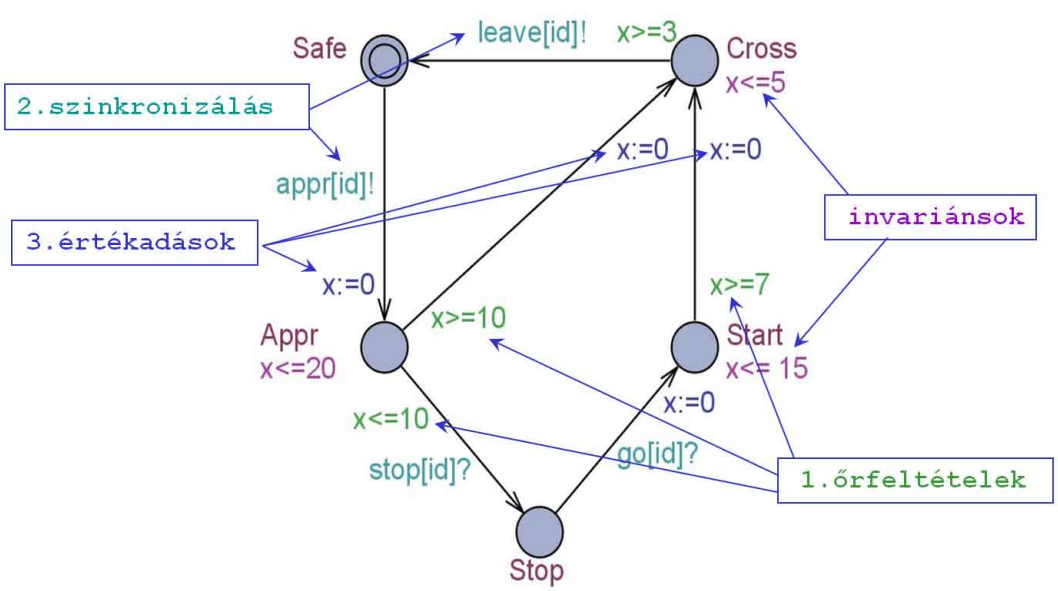
Az **UPPAAL** deklarációi a C nyelvhez hasonlóak. Lássunk néhány tipikus példát:

- `const int a = 1;` a egész típusú konstans, kezdeti értéke 1, az egészek alapértelmezett értékkészlete [-32768, 32767];
- `bool b[8], c[4];` b és c két 8 illetve 4 elemű tömb, melyek Boole értékeket tartalmaznak;
- `int[0,100] a=5;` a egy korlátozott egész, értéke 1 és 100 közötti lehet, kezdetben 5;
- `int a[2][3]=1,2,3,4,5,6;` a kétdimenziós egész értékeket tartalmazó tömb, kezdeti értékadással;
- `clock x, y;` x és y két óra;
- `chan d;` d egy csatorna neve, `d?` és `d!` használható a szinkronizáláshoz;
- `urgent chan e;` e egy sürgős (urgent) csatorna, ha rajta keresztül az üzenet küldés-fogadás létrejöhet, akkor nem választhatunk helyette várakozó átmenetet;
- `struct{int a; bool b;} s1={2,true};` s1 egy olyan rekord, melynek a komponense egész (kezdetben 2), második, b komponense Boole típusú (kezdetben igaz).
- `meta int swap;` swap metaváltozó, a modell állapotait nem növeli, használata például: `int a; int b; assign swap=a; a=b; b=swap;`

5.7.3. Automata sablonok szerkesztése

Az időzített automaták, pontosabban azok sablonjainak megadására az **UPPAAL** kényelmes felhasználói felületet biztosít. Egy kattintással adhatunk az automatához új helyet vagy átmenetet, lehetőségünk van a helyek és átmenetek paramétereinek megadására, valamint azok kényelmes mozgatására és színezésére is. Ez utóbbiak a modell működését nem befolyásolják, de az áttekinthetőség szempontjából nagy segítséget jelentenek.

A bal egér gombbal a helyeknél a kezdő (initial), sürgős (urgent) és az elkötelezett (committed) lehetőségek közül választhatunk. Ezekről később lesz szó. A szerkesztés (edit) menüponttal pedig az átmenetnél az *őrfeltétel* (guard), *szinkronizáció* (sync.) és *értékadás* (update), a helyeknél pedig az *invariáns* (invariant) jellemzőt adhatjuk meg. Ezeket az **UPPAAL** különböző színnel jelöli, lásd az 5.1 ábrát. A továbbiakban röviden ezeket tekintjük át.



5.1. ábra. Időzített automata sablon az **UPPAAL**-*ban*

5.7.4. Őrfeltételek

Az őrfeltételek szerepe az átmenetek engedélyezése, azaz egy átmenet csak akkor hajtható végre, ha a hozzá tartozó őrfeltétel igaz a rendszer aktuális állapotában.

Az őrfeltételeknek mindig Boole értékre kell kiértékelődniük. Csak óra változókra, egészekre és konstansokra (valamint azok tömbjeinek elemeire) hivatkozhatunk bennük. Az operandusok megfelelő típusúak kell, hogy legyenek, kivéve az `int` – `bool` konverziót. További megszorítások, hogy az óra változók (és azok különbsége) csak egészekkel hasonlíthatók össze, ezért az órákat egész értékűnek gondolhatjuk. Az őrfeltételek mellékhatás mentesek (side-effect free) kell, hogy legyenek, és a különböző órákra vonatkozó feltételeket csak konjunkcióval köthetjük össze, diszjunkcióval nem. Például: `x <100 or y==50` nem megengedett.

5.7.5. Szinkronizációk

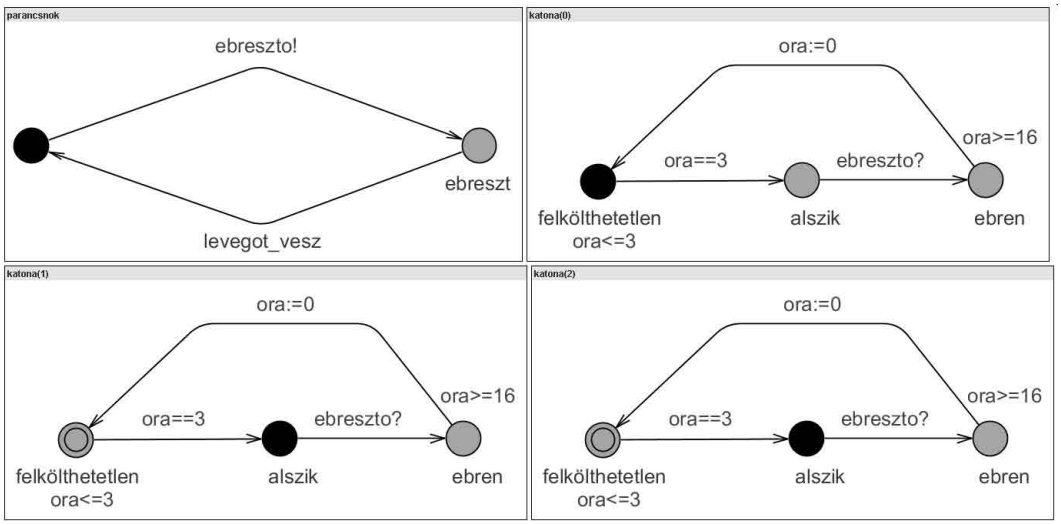
A csatornákat globálisan kell deklarálni. Lokális csatornáknak nincs értelme, mert nem látná a másik automata, amivel szinkronizálni akarunk. A csatornákból szabad viszont tömböt szervezni, azaz egész értékekkel paraméterezni őket. Például szabályos deklaráció `chan csat[3]`. Ekkor `csat[2]!` alatt gondolhatjuk azt, hogy a 2 értéket küldjük el a `csat` csatornába, melyet ezzel szinkronban egy másik automata `csat[2]?` segítségével tud kiolvasni, de gondolhatjuk azt is, hogy 3 különböző (`csat[0]`, `csat[1]`, `csat[2]`) cselekvéssel tudunk szinkronizálni. A speciális sürgős (urgent) csatornákról később lesz szó.

Kétfajta szinkronizáció lehetséges: *bináris szinkronizáció* és az *üzenetszórásos szinkronizáció* (brodcast syncronization).

A bináris szinkronizáció mindig két automata között jön létre. Az egyik a küldő, a másik a fogadó fél. Például: `chan c[3]` esetén, ha `t` értéke 2, akkor az egyik automatában a `c[t]!`

küldés csak a másik automatában a `c[2]?` fogadással egyszerre hajtható végre.

Fontos, hogy a bináris szinkronizáció küldés átmenete csak akkor hajtható végre, ha van fogadó fél, különben az átmenet blokkol.



5.2. ábra. Üzenetszórásos szinkronizáció az **UPPAAL***-ban*

A üzenetszórásos szinkronizáció általában nem csak két automata között zajlik. Ez a fajta szinkronizáció egy küldő és tetszőleges számú (esetleg nulla!) fogadó fél közt jön létre. Ha a csatornát a `broadcast` előtaggal deklaráltuk, például így: `broadcast chan a`, akkor az `a!` küldés, az összes pillanatnyilag végrehajtható `a?` fogadással szinkronizál. A bináris szinkronizációval ellentétben itt az `a!` küldés mindig végrehajtható. Legfeljebb, ha a rendszerben nincs olyan automata, mely az `a?` fogadó átmenetet végre tudná hajtani, akkor az üzenetküldés a többi automatára nincs hatással. Olyan, mintha a küldő egy szinkronizálással nem rendelkező átmenetet hajtana végre.

A kétfajta szinkronizáció különbségét mutatja az 5.2 ábra. Ha az `ébresztő` csatorna üzenetszórásos csatorna, akkor a bal felső `parancsnok` automata mind a bal alsó `katona(1)`, mind a jobb alsó `katona(2)` automatával szinkronizál. Amennyiben viszont egyszerű csatornáról van szó, akkor egyszerre csak az egyik alsó automatával jöhet létre szinkronizáció.

5.7.6. Értékadások

Az értékadások az átmenet végrehajtásának sorrendben harmadik lépését adják. Az őrfeltételekkel szemben megváltoztatják a modell állapotát, itt éppen a mellékhatás a lényeg. A típusokat itt is egyeztetni kell, és itt is csak konstansra, órára, egész változóra (és ezek tömbjeire) hivatkozhatunk. Az **UPPAAL**-ban nem csak az órák újraindítására van mód, hanem az óraváltozóknak tetszőleges, de csak egész érték adható. Ezen felül természetesen értéket adhatunk a többi változónak is.

5.7.7. Invariánsok

Az invariánsok mindig a helyekhez és nem az átmenetekhez kapcsolódnak. Mindig Boole értékre értékelődnek ki és mellékhatás mentesek. Azt a megszorítást fejezik ki, hogy az időzített automata csak addig tartózkodhat bármely helyen, amíg a helyhez tartozó invariáns feltétel igaz. Az invariáns feltétel nem válhat hamissá. Ha az automata nem tud helyet váltani, és az idő múlásával az invariáns feltétel hamissá válna, nincs következő rendszer állapot és a futás véget ér. Az invariánsok csak órákra, illetve órák különbségére vonatkozó egyszerű kifejezések konjunkciói, illetve olyan Boole kifejezések lehetnek, melyek órákat nem tartalmaznak. Az egyszerű kifejezés azt jelenti, hogy csak felső korlát használható, mely egész értékű kifejezéssel van meghatározva. Így például órákra vonatkozó alsó korlátokat nem használhatunk. Fontos, hogy az invariánsokat megkülönböztessük a helyességi kritériumoktól. A helyekhez tartozó invariánsok a rendszer leírásának részei, és olyan futások, melyek ezeket sértik, egyszerűen nem léteznek.

5.7.8. Sablonok (templates)

A sablonok használata kényelmes módszer több azonos vagy hasonló időzített automata definiálására. Hasonló, mint a **SPIN**-ben a `proctype` definíció. Az automata sablonokat korlátozott egész értékű paraméterekkel láthatjuk el. Ezeket a sablon neve utáni `Parameters` (paraméterek) mezőben kell megadni a változó deklarációkhoz hasonlóan. A sablonokat a rendszerdeklarációkban példányosíthatjuk a konkrét paraméterek megadásával, ekkor érték szerinti (call by value), vagy & használata esetén hivatkozás szerinti (call by reference) paraméterátadás történik. Egy másik lehetőség, hogy nem adjuk meg a paramétereket, és ekkor az **UPPAAL** önműködően létrehozza a példányokat, a paramétereket az összes lehetséges módon megválasztva az értékkészletből. Ez különösen kényelmes több azonos típusú automata megadására.

5.7.9. Sürgősség és elkötelezettség

Az **UPPAAL** helyei három különböző tulajdonsággal bírhatnak: kezdő (initial), sürgős (urgent) és az elkötelezett (committed). Ezek közül az utóbbi kettő kizárja egymást. Kezdő állapotból minden időzített automatában pontosan egynek kell lennie. A kezdőállapotot duplán bekarikázott kör jelöli. A sürgős állapot jele a karikán belül egy „U” betűhöz hasonló, az elkötelezett állapot jele pedig egy „C” betűhöz hasonló szimbólum.

A modellezéshez sokszor szükségünk van arra, hogy a rendszer ne várakozhasson bizonyos helyen, hanem haladás (progress) történjen. Például egy szinkronizáció azonnal létrejöjjön, mihelyt mindkét fél készen áll rá. Az erre szolgáló eszközök a következők:

- sürgős csatornák (urgent channels),
- sürgős helyek (urgent locations),
- elkötelezett helyek (committed locations).

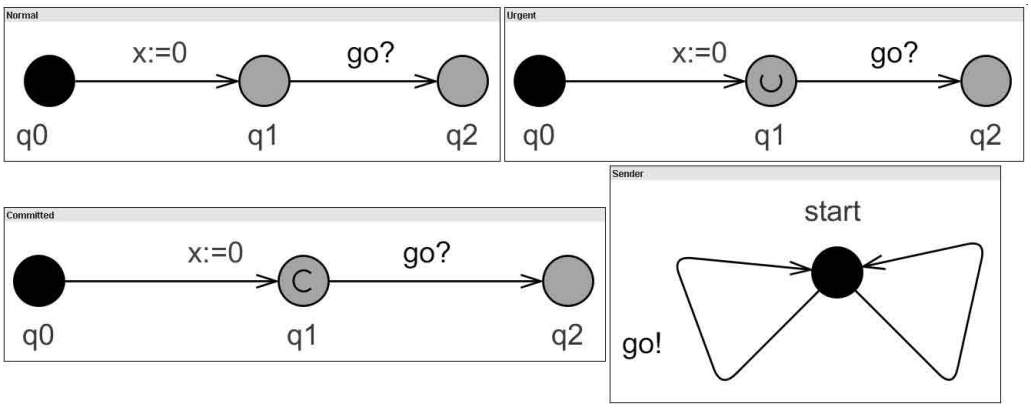
Megjegyezzük, hogy ezek az eszközök jelentősen csökkenthetik a szükséges órák számát és az állapottér méretét is.

Sürgős csatornák deklarációja az `urgent` előtaggal történik. Például: `urgent chan a, b[12];`. Ennek jelentése az, hogy ha a sürgős csatornán keresztül szinkronizálni lehet, sem a küldő sem a fogadó fél nem tehet várakozó átmenetet. De természetesen egymást meg kell várniuk.

A sürgős helyek (urgent locations) is hasonlóan a rendszer előrehaladását kényszerítik ki, ugyanis ezeken a helyeken nem telhet az idő. Ezt úgy képzelhetjük el, hogy amint a rendszer valamelyik automatája sürgős helyre jut, az összes órát befagyasztjuk, amíg a sürgős helyet el nem hagyjuk. Ám addig is, míg az automata a sürgős helyet el nem hagyja, a többi automata várakozástól különböző, azaz időbe nem kerülő átmeneteket tehet. Szemantikailag minden sürgős hely egyenértékű egy olyan normál hellyel, amelyhez egy új `x` órát veszünk fel, melyet minden bemenő él mentén újraindítunk, továbbá a hely invariánsaihoz az `x <= 0` feltételt is hozzáadjuk.

Az elkötelezett helyek még szigorúbb feltételt fogalmaznak meg. Ezeken a helyeken sem telhet az idő, sőt a következő átmenetek valamelyikének innen (vagy ha több is van, egy másik elkötelezett helyről) kell indulnia. Tehát az elkötelezett helyekről induló átmenetek a normál helyekről induló átmenetekkel szemben mindig elsőbbséget élveznek.

A fent említett fogalmak megértését megkönnyíti, ha az 5.3 ábrán látható modellel végzünk néhány szimulációt, mind sima, mind sürgős csatornát használva.



5.3. ábra. A sürgős és elkötelezett helyek viselkedését bemutató modell

5.7.10. Specifikáció az UPPAAL-ban

A helyességi kritériumok megadása az **UPPAAL**-ban a CTL logika megszorított változatával történik. A használható formulák az alábbiak:

- `E<>P`: van olyan futás, mely során valamikor igaz lesz P (**EF** P, lehetségesség, possibility).
- `A[]P`: minden futás minden állapotában igaz P (**AG** P, invariáns tulajdonság).

- `E[]P`: van olyan futás, melynek minden állapotában igaz P (**EG**P, lehetséges invariáns).
- `A<>P`: minden futás során valamikor igaz lesz P (**AF**P, előbb-utóbb biztosan P).
- `P-->Q`: minden úton, ha P teljesül, valamikor Q is teljesülni fog (válasz).

Az előzőekben P és Q már nem lehetnek összetett formulák, csak olyan kifejezések, melyek mellékhatás mentesek és Boole értékűek. A CTL-lel szemben az **UPPAAL** nem engedi meg a temporális operátorok egymásba ágyazását, például `A[](E<>P)`-t nem írhatunk.

A kifejezésekben hivatkozhatunk konstansokra, egész értékű változókra, órák értékeire és az automaták aktuális állapotaira. Például az alábbiak megengedettek:

- `A[] 1<2`: ez az invariáns mindig igaz.
- `E<> p1.cs and p2.cs`: igaz, ha a rendszer valamikor elérhet egy olyan állapotot, melyben mind $p1$, mind $p2$ a `cs` helyen van.
- `A[] p1.cs imply not p2.cs`: invariáns tulajdonság; valahányszor $p1$ a `cs` helyen van, mindannyiszor $p2$ nincs a `cs` helyen. (Nem keverendő a `-->` operátorral, mely megengedi az időbeli eltolódást.
- `A[] not deadlock`: a rendszerben nincs holtpont (`deadlock` beépített kulcsszó).

Mint láthattuk, a formulákban a `not`, `or`, `and` és `imply` Boole műveleteket is használhatjuk, a szokásos értelmezéssel. Végül felsorolunk néhány azonosságot, melyek azt mutatják, hogy bizonyos operátorok kifejezhetők a többi segítségével:

$$\texttt{not A[]}\,P = \texttt{E<> not}\,P$$
$$\texttt{not A<>}\,P = \texttt{E[] not}\,P$$
$$P\,\texttt{--> }Q = \texttt{A[]}(P\ \texttt{imply A<>}\,Q)$$

5.7.11. Példák

Bevezető gyakorló példaként az alábbi modellek [51] vizsgálatát ajánljuk, melyek a 2.9.2 fejezetben bemutatott időzítésre érzékeny kapcsoló és lámpa **UPPAAL** modelljét mutatják be és fejlesztik tovább.

- `lampa1.xml`: az időzítés nélküli modell azt mutatja meg, hogyan kell definiálni állapotokat és átmeneteket, hogyan működik a szimuláció, mit jelent a szinkronizálás.
- `lampa2.xml`: az időzítés bevezetésével kapott modell felépítése, őrfeltételek, óra újraindítások, szimulációk.
- `lampa3.xml`: az egy perc után magától kikapcsoló lámpa modellje, invariánsok használata, verifikáció.
- `lampa3.q`: CTL formulák megadása, a specifikációk ellenőrzése, a hiba trace elemzése a szimulátorban.
- `lampa4.xml`: sablonok (templates) használata, paraméterezés, rendszer deklarációja, további tulajdonságok ellenőrzése.

Irodalomjegyzék

[1] L. Aceto, A. Ingólfsdóttir, K. G. Larsen, J. Srba, Reactive Systems: Modelling, Specification and Verification, Cambridge University Press, 2007.

[2] J. B. Almeida, M. J. Frade, J. S. Pinto and S. Melo de Sousa, An Overview of Formal Methods Tools and Techniques, in: Rigorous Software Development, Undergraduate Topics in Computer Science, 15–44, Springer-Verlag, 2011.

[3] R. Alur, D. L. Dill, A theory of timed automata, Theoretical Computer Science 126(2), 183–235, 1994.

[4] A. Arnold: Finite Transition Systems: Semantics of Communicating Systems, Prentice-Hall International Series in Dynamics, Prentice-Hall, 1994.

[5] Ch. Baier and J–P. Katoen, Principles of Model Checking, MIT Press, 2008.

[6] K. A. Bartlett, R. A. Scantlebury and P. T. Wilkinson, A note on reliable full duplex transmission over half duplex links, Communications of the ACM, 12(5), 260–261, 1969.

[7] M. Ben-Ari, Principles of the Spin Model Checker, Springer, London, 2008.

[8] Barry W. Boehm, *Software Engineering Economics*, Prentice-Hall, Inc., Eaglewood Cliffs, New Jersey, 1981.

[9] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, S. Yovine, Kronos: a model-checking tool for real-time systems, in A. J. Hu and M. Y. Vardi (eds.), Proc. Computer Aided Verification, CAV 98, Lecture Notes in Computer Science 1427, Springer-Verlag, 546–550, 1998.

[10] IEEE 1076 Standard VHDL Language Reference Manual, 2008.

[11] IEEE 1364 Standard for Verilog Hardware Description Language, 2006.

[12] IEEE 1850 standard for PSL - property specication language, 2005, revised in 2010.

[13] G. Behrmann, A. David, K. G. Larsen: A Tutorial on Uppaal, in: Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04), LNCS 3185, revised and extended version: www.uppaal.com.

[14] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen: Systems and Software Verification, Springer, 2001.

[15] E. M. Clarke and E. A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in Logic of Programs, Lecture Notes in Computer Science, vol. 131, 52–71, Springer-Verlag, 1981.

[16] E. Clarke, O. Grumberg, and D. Peled, Model Checking, MIT Publishers, 1999.

[17] J. N. Buxton and B. Randell, eds, Software Engineering Techniques, April 1970, p. 16. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969.

[18] O. Grumberg, H. Veith (Eds.), 25 Years of Model Checking - History, Achievements, Perspectives, Lecture Notes in Computer Science 5000, Springer, 2008.

[19] K. Havelund, M. R. Lowry, J. Penix, Formal analysis of a space-craft controller using SPIN, IEEE Trans. Software Eng., 27(8),749–765, 2001.

[20] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[21] G. J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2003.

[22] G. Holzmann, M. H. Smith, Automating software feature verification, Bell Labs Technical Journal, Issue on Software Complexity, 5(2), 72–87, 2000.

[23] P. Kars, The Application of Promela and SPIN in the BOS Project, in: J.-C. Grégoire, G. J. Holzmann and D. Peled (eds), The Second Workshop on the SPIN Verification System; Proceedings of a DIMACS workshop, August 5, 1996, DIMACS, vol. 32, 51–63, AMS, 1997.

[24] T. Kropf, Introduction to Formal Hardware Verification, Springer-Verlag, 1999.

[25] A. Pataricza (szerk.): Formális módszerek az informatikában, Typotex kiadó, 2004

[26] G. L. Peterson, Myths about the mutual exclusion problem, Information Processing Letters, 12(3):15–116, 1981.

[27] J. L. Peterson, Petri Net Theory and the Modeling of Systems, Prentice-Hall, 1981.

[28] J.-P. Queille and J. Sifakis, Specification and verification of concurrent systems in CESAR, In 5th International Symposium on Programming, Lecture Notes in Computer Science, vol. 137, 337–351, Springer-Verlag, 1982.

[29] V. D'Silva, D. Kroening, G. Weissenbacher, A Survey of Automated Techniques for Formal Software Verification, *IEEE Trans. on CAD*, vol. 27, 1165–1178, 2008.

Elektronikus hivatkozások

[30] Az Astrée Static Analyzer weboldala, http://www.astree.ens.fr/

[31] M. Ben-Ari, Development Environments for Spin and Erigone, http://stwww.weizmann.ac.il/g-cs/benari/jspin/index.html

[32] A BLAST weboldala, http://mtc.epfl.ch/software-tools/blast/

[33] A CBMC weboldala, http://www.cprover.org/cbmc/

[34] A CMC weboldala, http://www.pst.ifi.lmu.de/~hammer/cmc/

[35] A CodeSonar weboldala, http://www.grammatech.com/products/codesonar/

[36] A Coverity weboldala, http://coverity.com/

[37] N. Dershowitz, „Software horror stories”, http://www.cs.tau.ac.il/~nachumd/horror.html

[38] Az F-Soft projekt weboldala, http://www.nec-labs.com/research/system/systems_SAV--website/projects.php#FSOFT

[39] A Java Pathfinder weboldala http://babelfish.arc.nasa.gov/trac/jpf

[40] A KRONOS weboldala http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/kronos/

[41] A Magic weboldala http://www.cs.cmu.edu/~chaki/magic/

[42] A NuSMV weboldala http://nusmv.fbk.eu/

[43] A Polyspace weboldala, http://www.mathworks.com/products/polyspace/

[44] A PRISM weboldala http://www.prismmodelchecker.org/

[45] A SAL weboldala http://sal.csl.sri.com/

[46] A Saturn weboldala http://saturn.stanford.edu/

[47] A SLAM weboldala http://research.microsoft.com/en-us/projects/slam/

[48] A SPIN weboldala, http://spinroot.com/

[49] A SPIN Online References, http://spinroot.com/spin/Man/

[50] UPPAAL hivatalos weboldalak, http://www.uppaal.org/ (akadémiai verzió), http://www.uppaal.com/ (kereskedelmi verzió).

[51] UPPAAL modellek, http://www.inf.u-szeged.hu/~zlnemeth/modell2008/UPPAAL/

[52] A Zing weboldala, http://research.microsoft.com/en-us/projects/zing/