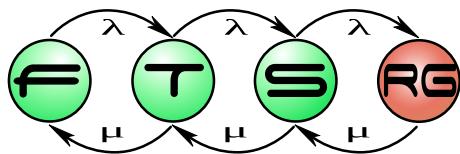


Rendszermodellezés

Hibatűrő Rendszerek Kutatócsoporthoz



Bergmann Gábor

Darvas Dániel

Molnár Vince

Szárnyas Gábor

Tóth Tamás

2017. augusztus 11.

Tartalomjegyzék

1. fejezet

Bevezető

Vajon mennyi programkódot képes átlátni egy programozó? Hány sorból áll a Windows 7 forráskódja?¹ Hogyan lehetséges mégis kézben tartani ekkora szoftverek fejlesztését? Hogyan garantálhatjuk, hogy a végső termék nem lesz hibás, megfelelő lesz a teljesítménye, és a rajta dolgozó mérnökökön kívül később más is képes lesz megérteni és módosítani a programot? Ezekre a kérdésre ad választ a Rendszermodellezés tárgy a komplex rendszerek modellezésének alapvető aspektusainak bemutatásával.

A *modellezés* egy jól bevált technika a bonyolult rendszerek komplexitásának kezelésében. A tárgy célja ezért a hallgatók megismertetése a szoftver- és hardverrendszerek modellezésének eszközeivel és módszereivel. A modellezés alapfogalmainak bevezetése után a modellek két nagy csoportjával, a struktúra-alapú és viselkedés-alapú modellekkel foglalkozunk. Szó lesz a modellek fejlesztéséről, illetve az elkészült modellek ellenőrzéséről, különböző felhasználásairól is.

¹Körülbelül 40 millió sorból [?].

Példa. Vállalatunk a kezdeti sikereken felbuzdulva úgy dönt, hogy belép a közösségi taxizás piacára egy saját szolgáltatás indításával. Ehhez ki kell fejleszteni egy nagyméretű, elosztott rendszert, amelynek részét képezi majd a cég belső gépeire telepített központi szerver és a felhasználók okostelefonjain futó mobilalkalmazás, valamint kapcsolatban fog állni egy bank rendszerével is a fizetések lebonyolításához.

A szenior szoftvermérnök a kezdetektől modellalapú fejlesztésben gondolkodik, hiszen a rendszer mérete és összetettsége több szempontból is indokolja a modellek használatát. Először is a tervezés első lépései során nem lenne szerencsés alacsony szintű technikai problémákkal foglalkozni. Egy megfelelően *absztrakt* modell segítségével a részletkérdések kezdetben elhanyagolhatók, így a mérnökök elsőként a legfontosabb jellemzők kialakítására fókuszálhatnak. Mivel a megcélzott terület a cég számára új, mindenekelőtt a kapcsolódó fogaalmak és a köztük húzódó összefüggések feltárása lesz a cél. A rendszer elosztottsága miatt szükség lesz az egyes összetevők (szerver, mobiltelefon, bank, hálózat stb.) kapcsolatainak modellezésére is. Az ezekhez szükséges eszközöket a *struktúra alapú modellezés* nyújtja.

Amint elkészül a rendszer architektúrája, a mérnökök nekiláthatnak az egyes komponensek viselkedésének tervezéséhez. Itt két kapcsolódó feladat is felmerül: meg kell tervezni a rendszer *folyamatait* (pl. autórendelés, regisztráció, egyenleg feltöltése stb.), illetve ennek megvalósításához az egyes komponensek belső *állapotait* és lehetséges állapotváltozásait. Az ezekhez szükséges eszközöket az *viselkedés alapú modellezés* nyújtja.

Természetesen a vezetőség szeretné elkerülni a tervezési hibák miatti többletköltségeket. A szenior mérnök javaslatára a megszokott *tesztelésen* kívül már a rendszer modelljein is *ellenőrzéseket* fognak végrehajtani, így a hibák az előtt kiderülhetnek, hogy a tényleges implementációt (drága emberi erőforrás felhasználásával) elkészítenék.

A szoftverek hibátlan működése mellett legalább ilyen fontos, hogy a rendszer *teljesítménye* megfelelő legyen. Ha a szolgáltatásunk nem elérhető vagy lassan válaszol, a felhasználók minden bizonnal a konkurenciához menekülnek majd, a vállalkozásunk pedig kudarcba fullad. Szerencsére a leendő rendszer teljesítménye is jól becsültető a modellek alapján, ehhez a *teljesítménymodellezés* nyújt majd eszköztárát.

Részben ide tartozik az is, hogy a legtöbb viselkedésmóddal *szimulálható*. Ez több szempontból is a hasznunkra válik majd, hiszen egrészt már a fejlesztés korai fázisában ki lehet próbálni az alapfunkciókat, másrészt a szimulációk eredményét is felhasználhatjuk a teljesítmény becslésére.

Akár a szimulációkból, akár később az elkészült rendszer monitorozásából rengeteg információ is a rendelkezésünkre áll majd. Ahhoz, hogy ezekből használható tudást állítsunk elő, és adott esetben problémákat, vagy a rendszer rejtegett tulajdonságait felfedezhessük, a *felderítő adatelemzés* eszköztárát is bevethetjük majd.

Ebben a jegyzetben a tárgyhoz tartozó legfontosabb témaikat mutatjuk be. Az alapfogalmak definícióit releváns mérnöki példákkal illusztráljuk és magyarázzuk, illetve gyakorló feladatokkal segítjük a bemutatott módszerek megértését. Ehhez különböző színű keretes írásokat fogunk használni. Amint az előzőekben látszott, a példák és a definíciók keretben kerülnek kiemelésre, míg a kiegészítő megjegyzések kisebb betűmérettel szedtük. A jegyzetben szerepelnek majd feladatok is kék háttérrel, valamint hasznos tippek zöld keretben.

Definíció. A *definíció* egy fogalom pontos, precíz meghatározása.

Megjegyzés. Időnként megjegyzésekkel fogjuk árnyalni a bemutatott anyagot.

Feladat. Gyakorlásclépp oldja meg az egyes fejezetekhez tartozó feladatokat!

Tipp. A feladatokhoz hasonló kérdések előfordulhatnak a számonkéréseken is.

A jegyzethez tárgymutató is tartozik, amely tartalmazza az előforduló kifejezések magyar és angol megfelelőjét, továbbá egyes angol kifejezések brit angol szerinti kiejtését az IPA jelölésrendszerével.

A jegyzetről

A jegyzet LATEX nyelvű forráskódja elérhető a <https://github.com/FTSRG/remo-jegyzet> oldalon.

A jegyzet jelenleg fejlesztés alatt áll. Amennyiben az elkészült részekben hibát vagy hiányosságot találnak, kérjük jelezze a GitHubon egy új *issue*² felvételével. A javítási ötleteket szívesen fogadjuk *pull requestek*³ formájában is.

²<https://help.github.com/articles/about-issues/>

³<https://help.github.com/articles/using-pull-requests/>

2. fejezet

Modellezés és metamodellezés

Ebben a fejezetben a modellezés alapfogalmaival fogunk megismerkedni. Az itt bevezetett fogalmak újra és újra megjelennek majd a későbbi fejezetekben, ahol részletesen ki fogunk térti az adott területen történő értelmezésükre.

2.1. Modellezés

Mi értelme van modellezni? Fejben szinte minden modellezünk, bármilyen probléma kerül előnk. Nincs ez másképp egy szoftver fejlesztésekor sem. Lássuk hát, miről beszélünk, amikor a modell szót használjuk.

Definíció. *Modell:* egy valós vagy hipotetikus világ (a „*rendszer*”) egy részének egyszerűsített képe, amely a rendszert helyettesíti bizonyos megfontolásokban. Egy modell elkészítésének mindenkor egy kérdés megválaszolása a célja.

A modell tehát egy többnyire bonyolult rendszer egyszerűsített reprezentációja, amiben csak az aktuális probléma szempontjából lényeges vonásokat szerepeltetjük. Egy adott rendszer modellé történő leképezésének általában két fontos előnye van:

- A modell az eredeti rendszernél *kisebb*, hiszen az aktuális problémához nem (vagy lazábban) kapcsolódó, elhanyagolható információk nem jelennek meg benne.
- A modell az eredeti rendszernél *áttekinthetőbb*, hiszen csak az adott probléma szempontjából érdekes, releváns információkat és kapcsolatokat kell vizsgálni.

Példa. Modellekre sok példát láthatunk a hétköznapokban is. Nem csak gyerekek körében népszerű játék a modellvasút. Itt valóban modellről beszélhetünk, hiszen a járatok számos tekintetben hűen reprezentálják a valódi vonatokat, azonban például „szemet hunyunk” a méretükkel, tömegükkel, a bennük lévő villanymotor paramétereivel és még sok egyéb hasonló tulajdonságukkal kapcsolatban.

Az informatika egyik leggazdagabb modellforrása a matematika. Amikor gráfelméletet tanulunk, valójában rengeteg, bizonyos szempontból hasonló probléma közös modelljét vizsgáljuk. Valóban, a matematika egyik célja az ilyen modellek azonosítása és minél hatékonyabb eszköztárak kidolgozása a rajtuk megfogalmazott feladatok megoldására. A gráfoknál maradva (gráfokról bővebben a *Struktúra alapú modellezés* c. segédanyagban és a *Bevezetés a számításelméletbe 2.* c. tárgyban lehet tájékozódni) egy város úthálózata jól reprezentálható egy élsílyokkal ellátott gráffal, ahol a csomópontok a kereszteződések, az élek az útszakaszok, az élsílyok pedig a szakaszok hosszait jelölik. Ez a modell kiválóan alkalmas legrövidebb útvonalak tervezésére (mi kellene még a *leggyorsabb* útvonal tervezéséhez?), de figyelmen kívül hagy számos egyéb paramétert, például az utakon lévő kanyarulatokat, a legnagyobb megengedett sebességet stb.

Megjegyzés. Az, hogy a modell kisebb, nem minden „kényelmi” szempont. Gyakran lehet olyan problémával találkozni, aminek a mérete bizonyos szempontból *végtelen* (például végtelen sok állapota van, folytonos változók vannak benne, esetleg részét képezi az *idő*), viszont egy alkalmas *véges* modellen a számunkra releváns tulajdonságok továbbra is jól vizsgálhatók maradnak. Egy autót fizikai szempontból modellezhet a pillanatnyi sebességvektora, ami három valós szám, tehát a modellünk így végtelen. Ha viszont feltételezzük, hogy a sebesség nagysága nem lehet 200 km/h-nál több, és két tizedesjegy pontossággal adjuk meg a vektor koordinátáit (tehát *diszkrét értékekkel* jellemezzük a rendszert), akkor márás véges modellt kapunk. Természetesen ez a modell kicsit torzítani fog a valósághoz képest, de számos problémánál ez a hiba elhanyagolhatóan kicsi lesz (ráadásul végtelen modellen lehet, hogy nem is tudnánk megoldást adni).

Fontos kérdés, hogy hogyan lehet ábrázolni egy modellt. Maga a modell ugyanis többnyire egy hipotetikus struktúra, nem kézzel fogható és nem is minden célszerű teljes részletességgel ábrázolni.

Definíció. *Diagram:* a modell egy nézete, amely a modell bizonyos aspektusait grafikusan ábrázolja.

Megjegyzés. Fontos megjegyezni, hogy nem minden modell, ami modellnek látszik.

- *A modell nem a valóság:* az általunk definiált modellen bizonyos állítások igazak lehetnek, amik a valóságban nem állják meg a helyüket.
- *A modell nem a diagram:* a diagram csak egy ábrázolás módja a modellnek, amivel olvashatóvá tesszük.

2.2. Modellezési nyelvek

A modellek leírásához nem elég, ha diagramokat rajzolunk. Egy modell precíz megadásához szükség van egy *modellezési nyelvre*. A modellezési nyelvek legfőbb célja a kommunikáció gép-gép, ember-gép, vagy akár ember-ember között is. Egy modellezési nyelv lehet *szöveges* (pl. Verilog, VHDL, Java stb.) vagy *grafikus* (pl. webes konyhatervező, UML diagramok stb.). Gyakori, hogy egy modellezési nyelv szöveges és grafikus jelölésrendszert is definiál, ugyanis mindenek megvannak a saját előnyei és hátrányai: jellemzően modellt építeni könnyebb szöveges nyelv használataval, míg a modell olvasása grafikus nyelvek (diagramok) segítségével egyszerűbb.

Definíció. Egy *modellezési nyelv* négy részből áll:

- *Absztrakt szintaxis* (más néven *metamodell*): meghatározza, hogy a nyelvnek milyen típusú elemei vannak, az elemek milyen kapcsolatban állhatnak egymással, és a típusoknak mi a viszonya egymáshoz. Ezt a reprezentációt használják gépek a modell belső tárolására.
- *Konkrét szintaxis*: az absztrakt szintaxis által definiált elemtípusokhoz és kapcsolatokhoz szöveges vagy grafikus jelölésrendszert definiál. Ezt a reprezentációt használják az emberek a modell olvasására és szerkesztésére.
- *Jólformáltsági kényszer*: Megadja, hogy egy érvényes modellnek milyen egyéb követelményeknek kell megfelelnie.
- *Szemantika*: az absztrakt szintaxis által megadott nyelvi elemek jelentését definiáló szabályrendszer.

Egy modellezési nyelvhez több konkrét szintaxis is adható.

Az absztrakt szintaxis tekinthető a modellezési nyelv modelljének, ezért hívjuk *metamodellnek* is. A konkrét szintaxis a metamodellhez képest annyival több, hogy a definiált elemekhez megjeleníthető reprezentációkat rendel, ezáltal válik olvashatóvá és szerkeszthetővé egy modell leírása. Ha nem okoz félreértest, akkor a szintaxis szót leggyakrabban az absztrakt és konkrét szintaxis együttes jelölésére használjuk (tehát az elemkészletre és a jelölésekre együtt). A jólformáltsági kényszerek tovább szűkítik a lehetséges modellek körét olyan szabályokkal, mint például az azonos nevű elemek tiltása. Végül a szemantika megadja, hogy az így leírt modell pontosan mit is jelent, vagy hogyan „működik”. A szemantika fogalmát strukturális és viselkedési modellek esetén kissé eltérően értelmezzük majd, erre a későbbi fejezetekben térünk vissza.

Példa. A C nyelv egy szöveges nyelv, melyben elemeknek tekinthetjük az **if**, **for**, **switch**, **változónevek**, **metódusnevek**, **típusok** stb. részeket. Ezek azonban nem tetszőleges sorrendben állhatnak egymás mellett, hanem egy jól meghatározott szabályrendszer szerint. Ez a szintaxis. Azt, hogy az **if** kulcsszóval elágazást, a **for** kulcsszóval pedig egy ciklust definiálunk és nem pedig valami teljesen más dolgot, a szemantika határozza meg. Vagyis a szemantika jelentéssel tölti meg a nyelvi elemeket.

Példa. A YAKINDU Statechart Tools^a egy állapotgépek specifikálását és fejlesztését lehetővé tévő nyílt forráskódú eszköz, mely az Eclipse IDE alapjaira épül. Kényelmi funkcióihoz tartozik az elkészített modell validációja, szimulálásának lehetősége, ill. a modellalapú kódgenerálás is. A szöveges (XML) nyelv mellett egy könnyen használható grafikus felületet is biztosít, ahol lehetőségünk van állapotokat és átmeneteket definiálni. Ezek nyelvi elemként, mint dobozok és nyilak jelennek meg. A grafikai szintaxis meghatározza, hogy a dobozokból csak nyilakkal lehet összekötni más dobozokat. Ekkor igazából azt adjuk meg, hogy egy adott állapotból alkalmazásunk milyen másik állapotba kerülhet. Ez a szemantikai jelentés. Az állapot alapú modellek leírásával részletesen a 4. fejezetben fogunk foglalkozni.

^a<http://statecharts.org>

Megjegyzés. A metamodell ugyanúgy modell, mint a segítségével leírható modellek. Ebből kifolyólag ugyanúgy adható hozzá modellezési nyelv, aminek szintén lesz egy metamodellje. A gyakorlatban ez addig folytatódik, amíg az egyik metamodellt már egy szabványos modellezési nyelv segítségével írjuk fel. Érdekesség, hogy az UML metamodell-hierarchiának gyökerében egy *önleíró modell* található: önmaga a saját metamodellje is egyben.

Példa. Modellekre és metamodellekre számos példa kerül majd elő a későbbi tanulmányok során:

- Az UML [?] objektum diagramján ábrázolt objektummodelleket az osztálydiagramokon ábrázolt metamodelljeik szerint adhatjuk meg (*Szoftvertechnológia* tárgy).
- Az XML dokumentumok metamodelljét az XML sémák adják (*Szoftvertechnológia* tárgy).
- Egy adatbázis tábla metamodellje a relációs adatbázisséma (*Adatbázisok* tárgy).

A konkrét szintaxist (jelölésrendszert) mindegyik esetben külön határozzák meg, az UML esetében többféle grafikus és szöveges szintaxis is definiált.

Tipp. minden fejezetben, ahol modellezési formalizmusokkal ismerkedünk meg, szerepelni fog az adott modelltípus metamodellje is.

2.3. Nyílt és zárt világ feltételezés

Egy modell szemantikájának definiálása során több feltételezésből is kiindulhatunk. Ezeket rögzíteni kell, és a modell értelmezéskor döntő szerepük van a különböző állítások igazságtartalmának eldöntésekor.

Definíció. *Zárt világ feltételezés:* minden állítás, amiről nem ismert, hogy igaz, hamis.

Definíció. *Nyílt világ feltételezés:* egy állítás annak ellenére is lehet igaz, hogy ez a tény nem ismert.

A két feltételezés között az egyik legfontosabb különbség, hogy a nyílt világ feltételezés elismeri és használja az *ismeretlen* fogalmát, míg a zárt világban minden tudás ismertnek tekintett. Mindkét feltételezésnek megvan a maga szerepe és helye, ahogy azt a következő két példa is illusztrálja.

Példa. A zárt világ feltételezést olyankor alkalmazzuk, amikor egy rendszernek a teljes szükséges információ a rendelkezésére áll. Erre példa egy tömegközlekedési adatbázis: ha megkérdezzük, hogy közlekedik-e közvetlen járat a Magyar Tudósok körútja és a Gellért tér között, és ilyen járat nincs az adatbázisban, akkor a válasz egyértelmű „nem”. Ebben az esetben valóban ez az elvárt és helyes válasz.

A nyílt világ szemantika olyankor alkalmazandó, amikor nem feltételezhetjük, hogy minden információ a rendelkezésünkre áll. Például egy orvosi nyilvántartó rendszerben előfordulhat (sőt, számítani kell rá), hogy egy beteg annak ellenére allergiás valamire, hogy ez a nyilvántartásban nem szerepel.

2.4. Absztrakció és finomítás

A modell definíciójának szerves része az *egyszerűsítés*, az adott probléma szempontjából irreleváns információk elhanyagolása. Ugyanannak a rendszernek több modelljét is elkészíthetjük más-más részletek kihagyásával. Ráadásul egy elkészített modellből további részleteket hagyhatunk el, vagy éppen újabb részleteket vehetünk bele, így a modellek a részletgazdagság szerint egyfajta hierarchiába rendezhetők.

Különbséget kell tennünk azonban egy modell részletekkel gazdagítása és megváltoztatása között. Ehhez segít, ha megkülönböztetjük a modellt (illetve a modellezett rendszert) és a környezetét.

Definíció. *Rendszer* és *környezete*: a környezet (vagy kontextus) a rendszerre ható tényezők összessége. Modellezéskor a modellezett rendszernek minden egyértelműen definiálni kell a *határait*. A határon belül eső dolgokat a rendszer részének tekintjük, az azon kívül esők adják a környezetet. Szokás még a környezet elemeit két csoportba sorolni: *releváns* környezeti elemek azok a dolgok, amik a rendszerrel közvetve vagy közvetett módon kapcsolatban állnak, míg *irreleváns* környezeti elem, ami a rendszerrel nincs kapcsolatban.

Megjegyzés. Gyakori, hogy egy rendszer környezetéről (is) készítünk modelleket. Ennek hasznára a rendszer és a környezet interakcióinak pontosabb megértése, tervezése és analízise. Az elkészült rendszerek tesztelésekor például gyakran nem az éles környezetben tesztelünk, hanem a környezet modellje alapján szimuláljuk az interakciókat.

A környezet szempontjából nézve a rendszert tekinthetjük *fekete doboznak* vagy *fehér doboznak*. Előbbi esetben a rendszer belső felépítését és viselkedését nem ismerjük, míg utóbbi esetben ezek az információk rendelkezésre állnak. Ez a két fogalom főleg tesztek tervezésékor érdekes, erről bővebben a 8. fejezetben fejezetben lesz szó.

Példa. Sportautók tervezésekor a rendszer jól körülhatárolható. Aerodinamikai szempontból releváns külvilágna tekinthető az autó körül áramló levegő. Tervezéskor célszerű a karosszéria (rendszer) és a légáramlás (környezet) modelljét is elkészíteni, hogy minél pontosabb szimulációkkal megfelelő jóslatokat tudunk tenni a terv minőségét illetően. Később, amikor az elkészült prototípus tesztelése folyik, a szélcsatorna tekinthető a légáramlás egy pontosabb, fizikai modelljének.

A környezet segítségével definiálhatjuk, hogy mit jelent a modell részletekkel gazdagítása.

Definíció. *Finomítás*: a modell olyan részletezése (Pontosítása), hogy a környezet szempontjából nézve a finomított modell (valamilyen tekintetben) helyettesíteni tudja az eredeti modellt.

A finomítás minden többletismeret bevitelét jelenti a modellbe. A finomítás konkrét módja és a helyettesíthetőség pontos definíciója problémafüggő, amikre sok példát láthatunk majd a későbbi fejezetekben. Természetesen a finomításnak ellentétes művelete is van.

Definíció. *Absztrakció*: a finomítás inverz művelete, vagyis a modell részletezettségének csökkentése, a modellezett ismeretek egyszerűsítése.

A szabályos finomítás után az eredeti modell minden visszakapható egy szabályos absztrakcióval. Érdemes megjegyezni, hogy finomítás során többnyire tervezői döntést hozunk annak tekintetében, hogy egy adott részletet hogyan illesztünk a modellbe, míg az absztrakció során a részlet elhagyása egyértelmű. Következésképpen egy finomítási lépésnek általában sokféle eredménye lehet, de egy adott absztrakciós lépésnek mindenkor csak egy.

Példa. Egy közlekedési lámpának megadható egy absztract és egy részletes modellje is. Az absztract modellben a lámpának két állása van: *tilos* és *szabad*. A részletesebb modellben a *szabad* állapotnak megfelelhet a *zöld* állapot, a *tilos* állapotban pedig a *sárga*, *piros* és *piros-sárga*.

Figyeljük meg, hogy az absztract modellből a részletesebbé lépés (finomítás) során többlet ismeretet vittünk a modellbe, mégpedig a lámpákkal kapcsolatban. Ráadásul mivel az új állapotok mindegyike egyértelműen megfeleltethető egy réginek, a környezet számára a rendszer továbbra is leírható az absztract modellel.

Azt is láthatjuk, hogy a finomítási lépésnek („bontsuk szét a *tilos* állapotot”) több megoldása is lehetett volna, míg az ennek megfelelő absztrakciós lépésnek („vonjuk össze a *sárga*, *piros* és *piros-sárga* állapotokat”) csak egyetlen megoldása van (az elnevezésekkel eltekintve).

Tipp. A későbbi fejezetekben az adott témakör vonatkozásában számos példát adunk majd absztrakcióra és finomításra.

3. fejezet

Struktúra alapú modellezés

Hogyan épül fel egy rendszer? Milyen részekre bontható és ezek között milyen kapcsolat van? Ahhoz, hogy a rendszerünkkel kapcsolatos problémákat megválaszoljuk, fontos, hogy ezekre a kérdésekre tudjuk a választ. Ebben a fejezetben az egyes rendszerek *struktúrájának* jellemzésével foglalkozunk. Bemutatjuk a strukturális modellezés motivációját, a leggyakrabban alkalmazott formalizmusokat és azok matematikai alapjait.

3.1. A strukturális modellezés alkalmazásai

Mind a természetben, mind az ember alkotta rendszerekben fellelhetők bizonyos szabályszerűségek: egyesek a rendszer elemei közötti kapcsolatokat, míg mások magukat az elemeket jellemzik. Bár sok mindenben eltér egy közlekedési hálózat, egy számítógép-hálózat, egy épület vagy egy város felépítése, a strukturális modellezés eszköztárával olyan „nyelvet” kapunk, amivel ezeket hasonlóan modellezhetjük.

3.1.1. Hálózatok

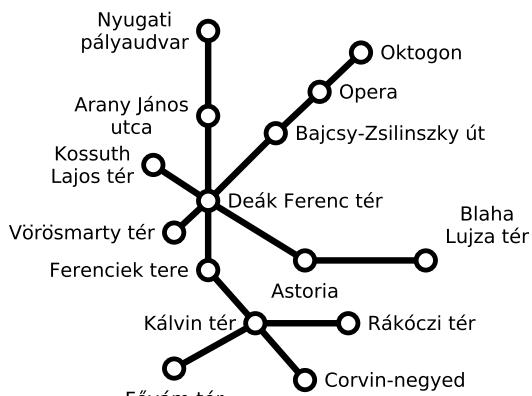
Egy rendszert gyakran úgy jellemzhetünk a legjobban, ha bizonyos elemeit megkülönböztetjük és leírjuk az ezek közötti *kapcsolatokat*.

Példa. Egy nagyváros közlekedése az autóutak és sínhálózatok, a százezernyi járműnek és a rajta utazó embereknek szövevényes rendszere. A közlekedők folyamatosan mozgása mellett az infrastruktúra is rendszeresen változik a különböző fejlesztések, átalakítások és karbantartások miatt.

Egy ilyen rendszer precíz modellezése lehetetlen vállalkozás lenne, ezért helyette tipikusan olyan absztrakciókkal dolgozunk, amik az adott probléma megoldásához szükséges információkat tartalmazzák. Például az útvonalkereső alkalmazások ismerik a helyi tömegközlekedés járatait és segítséget nyújtanak abban, hogy az indulási pontunktól a célállomásig közlekedő járműveket és a közöttük lehetséges átszállásokat listázzák.

Vizsgáljuk meg például Budapest metróhálózatát! Az egyszerűség kedvéért a példában a hálózatnak csak a Nagykörúton belüli részével foglalkozunk.

A metróhálózat egyszerűsített térképe a 3.1(a) ábrán látható. A térképből is látható, hogy a hálózat könnyen ábrázolható egy gráffal, ahol a gráf minden *csmópontja* egy-egy metróállomást jelöl. A csomópontok címkeje a metrómegálló neve. Két csomópont között akkor fut *él*, ha a két megálló közvetlenül össze van kötve metróval (azaz nincs közöttük másik megálló). A metróhálózatot modellező gráf a 3.1(b) ábrán látható.



(b) A metróhálózat egyszerű gráfként ábrázolva

3.1. ábra. Budapest metróhálózata a Nagykörúton belül

A modellünk segítségével választ kaphatunk például a következő kérdésekre:

1. Milyen megállók érhetők el a Vörösmarty térről indulva?

Vizsgáljuk meg, hogy a Vörösmarty teret reprezentáló csomópontból indulva milyen csomópontok érhetők el. Ehhez elemi gráfbejáró algoritmusok használunk.

Megjegyzés. Elemi útkereső algoritmusok pl. a *szélességi keresés* vagy *méliségi keresés*.

2. Legalább hány megállót kell utaznunk a Kossuth Lajos tér és a Kálvin tér között?

A legrövidebb utat szintén kereséssel határozhatjuk meg.

Megjegyzés. Például egy *szélességi kereséssel*.

Vannak azonban olyan metróközlekedéssel kapcsolatos kérdések, amelyek megválasztásához a modell nem tartalmaz elég információt:

1. Milyen megállók érhetők el a Fővám térről indulva *legfeljebb egy átszállással*?
2. A menetrend szerint *hány percig tart* az út a Kossuth Lajos tér és az Astoria között?

Ezeknek a kérdéseknek megválasztásához egészítsük ki a gráfot! Az első kérdéshez szükséges, hogy az egyes metróvonalakat meg tudjuk különböztetni, amit például az élek címkézésével érhetünk el. A 3.2(a) ábrán színekkel jelöltük a különböző élcímkéket. Indulunk ki a Fővám térről: átszállás nélkül a Kálvin tér és a Rákóczi tér megállókat érhetjük el, míg egy átszállással elérhetjük az M3-as metró vonalán található megállókat is.



(a) A gráf elcímekkel kiegészítve



(b) A gráf élsúlyokkal kiegészítve

3.2. ábra. A metróhálózatot ábrázoló gráf kiterjesztései

A második kérdés megválasztásához az egyes megállók közötti utazási időt kell jellemeznünk. Ehhez vegyük fel *élsúlyokat* a gráfba. A 3.2(b) ábrán élsúlyokkal jelöltük az egyes megállók között menetidőt. Ezek ismeretében meghatározható a Kossuth Lajos tér és a Kálvin tér közötti út menetrend szerinti időtartama. Ez a modell arra is alkalmas, hogy meghatározzuk a legrövidebb utat a két csomópont között, például a *Dijkstra algoritmus* segítségével.

Példa. Melyik metróvonalak között tudunk átszállni? A kérdésre választ kapunk a fenti modell segítségével. Nagyméretű hálózat esetén azonban már sokáig tarthat a válasz meghatározása, ezért érdemes olyan modellt készíteni, ami a válaszhoz nem szükséges részeket *absztrahálja*.



3.3. ábra. Átszállási lehetőségek a metróvonalak között a Nagykörúton belül

A 3.3. ábrán látható modellből azonnal kiderül, hogy mely metróvonalak között van átszállási lehetőség. A modell segítségével tehát hatékonyabban tudunk válaszolni erre a kérdésre, de a korábbi kérdésekhez szükséges információt elvesztettük az absztrakció során.

A közlekedési hálózathoz hasonlóan sok rendszer jól modellezhető gráffal: az élőlények táplálkozási lánca, a közösségi hálók, az úthálózat, telekommunikációs hálózatok stb.

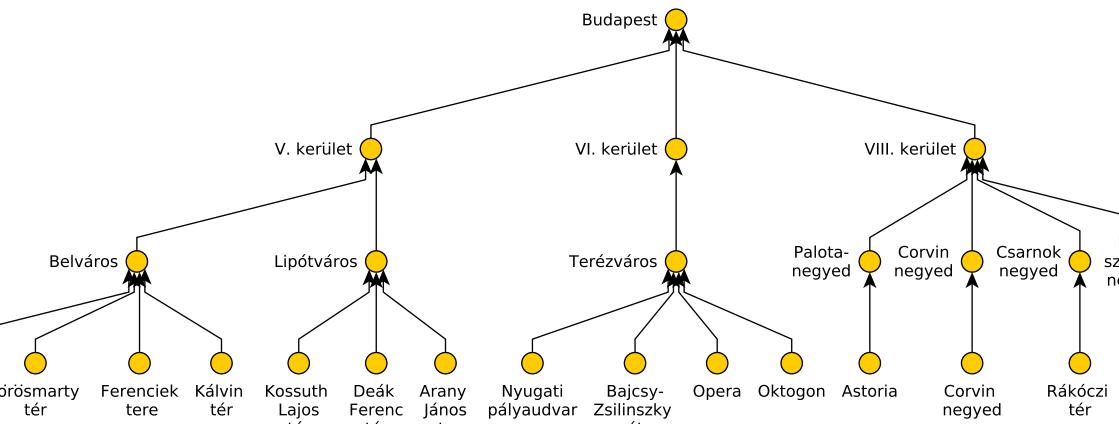
3.1.2. Hierarchikus rendszerek

Példa. Budapestnek 23 kerülete van, amelyek további városrészekből állnak. Melyik városrészben van az Opera metrómegállója? Melyik városrészben van a legtöbb metrómegálló? Ezekhez hasonló kérdésekre úgy tudunk hatékonyan válaszolni, ha készítünk egy *hierarchikus modellt* a problémáról.

Készítünk modellt, amely ábrázolja Budapest, a kerületek, a városrészek és a metrómegállók viszonyát! A 3.4. ábra modellje négy szintet tartalmaz:

1. A hierarchia legfelső szintjén Budapest szerepel.
2. A második szinten a város kerületei találhatók.
3. A harmadik szinten az egyes városrészek vannak.
4. A negyedik szinten a metrómegállók találhatók.

Látható, hogy a hierarchikus modellt is ábrázolhatjuk gráfként. A csomópontok a modell különböző szintű elemeit reprezentálják, míg az élek a része viszonyt fejezik ki, például az Opera megálló a VI. kerület része. A gráf *gyökér csomópontja* a hierarchiában legmagasabban szereplő elem, Budapest.

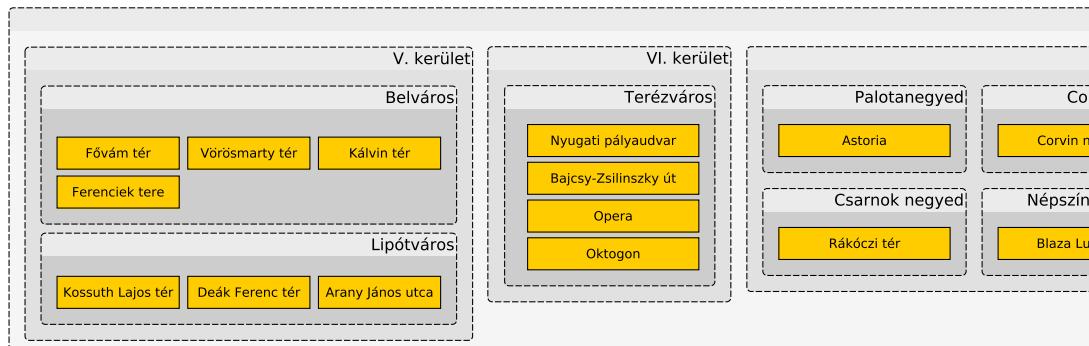


3.4. ábra. Budapest kerületei, városrészei és metrómegállói (részleges modell)

Amennyiben egy rendszert hierarchikusan részekre bontunk, nem fordulhat elő, hogy egy elem tartalmazza a szülő elemét, ezért a hierarchikus modelleket reprezentáló gráfok körmentesek. A gyökér elemet leszámítva minden elemnek van szülője, tehát a gráf összefüggő is, így a hierarchikus modellek gráfjai egyúttal fák.

Megjegyzés. A fa struktúrában a gráf élei *explicit módon* jelölik a tartalmazási hierarchiát. A gyakorlatban nem minden jelenítjük meg explicit módon a tartalmazási viszonyokat.

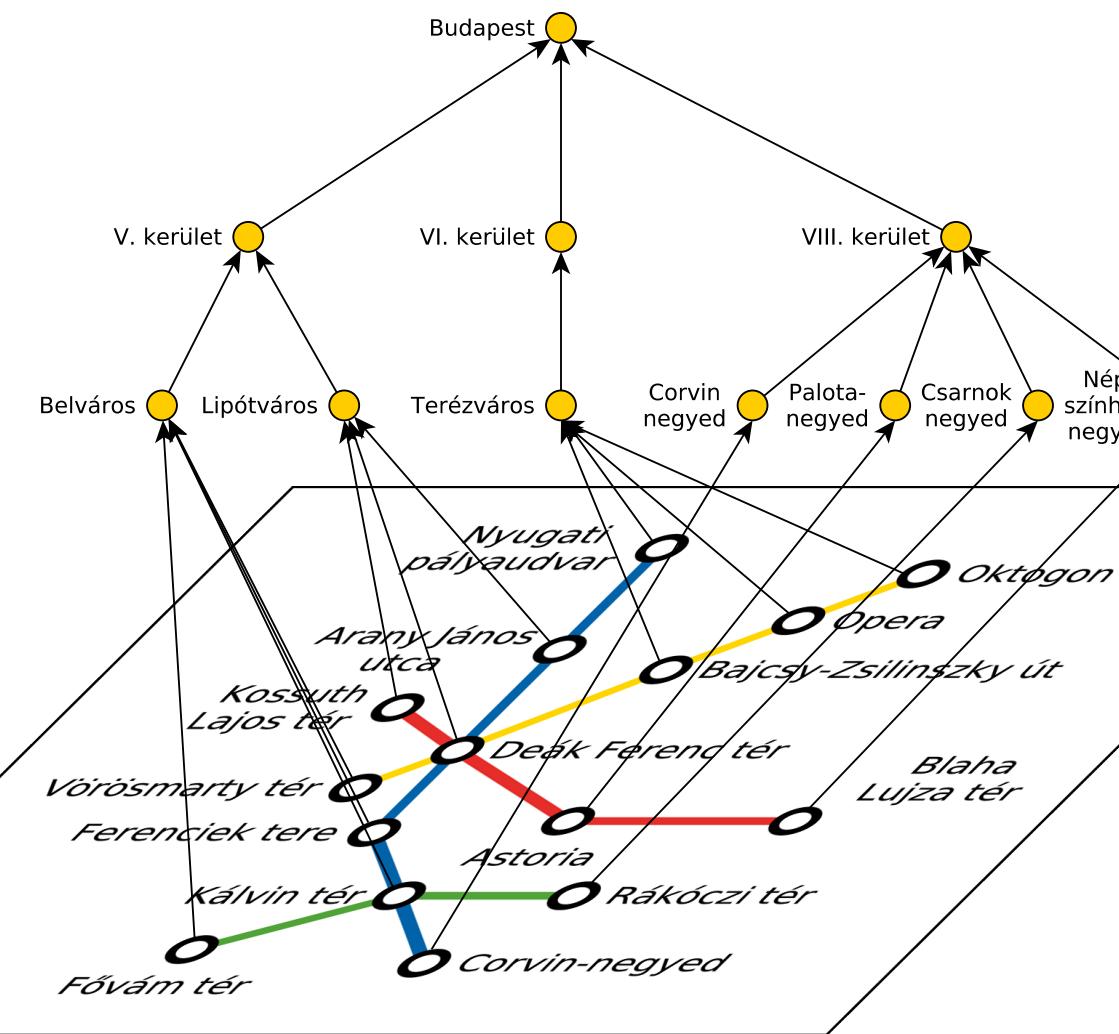
A tartalmazási hierarchia ábrázolható a tartalmazási viszonyok *implicit megjelenítésével* is:



3.5. ábra. A tartalmazási viszonyok implicit módon jelölve

A 3.5. ábrán egy olyan diagramot látunk, amelyben az egyes síkidomok közötti bennfoglaló ábrázolása reprezentálja a modellben szereplő tartalmazási viszonyt.

Láthatunk tehát, hogy mind a modellelemek közötti kapcsolat, mind a modellhierarchia hatékonyan ábrázolható gráfként. A 3.6. ábrán látható modell a 3.2(a) és a 3.4. ábrákon található metróhálózatot és a területek hierarchiáját is tartalmazza.



3.6. ábra. Budapest metróhálózata és a városrészek, kerületek hierarchiája

3.1.3. Tulajdonságok

Példa. A közlekedési vállalat üzemen kívüli járművei telephelyeken parkolnak és itt végzik rajtuk a szükséges karbantartásokat. A metrók telephelyeinek neve metró járműtelep, a buszoké az autóbuszgarázs.

Mennyi üzemanyag tárolható a legnagyobb autóbusz garázsban? Milyen hosszú a Kőér utcai metró járműtelep vágányhálózata? Ezek a kérdések a modellünk elemeinek tulajdonságaival kapcsolatban merülnek fel. Építsünk modellt a problémára!

A telephelyeket például az alábbi modellel írhatjuk le:

azonosító	helyszín	funkció	kapacitás	vágányhossz	max. üzemanyag
T1	Mexikói út	metró járműtelep	24	8 500 m	
T2	Kőér utcai	metró járműtelep	60	16 512 m	
T3	Cinkota	autóbuszgarázs	265		250 000 liter
T4	Kelenföld	autóbuszgarázs	322		200 000 liter

3.1. táblázat. A BKK telephelyei (részleges modell)

A modellünk elemei a táblázat sorai. A táblázat fejlécében definiált *tulajdonságokból* (pl. helyszín, vágányhossz) az egyes modellelemek eltérő értékeit vehetnek fel. Látható, hogy a táblázatnak nem minden celláját töltöttük ki, mivel az egyes jellemzők nincsenek mindenhol értelmezve.

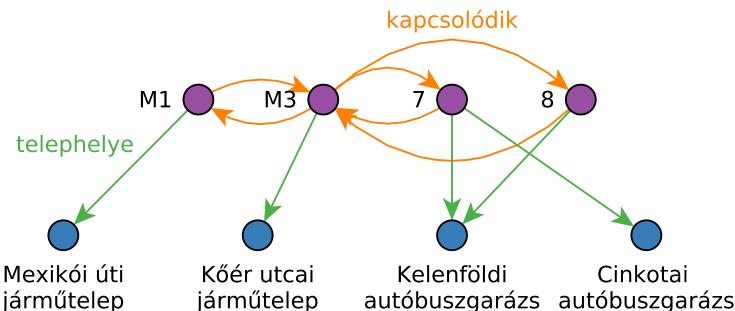
Megfigyelhetjük azonban, hogy az azonos funkciójával rendelkező modellelemek azonos tulajdonságokra vesznek fel értéket. Mindkét típusú telephelynek van *azonosító*, *helyszín*, *funkció* és *kapacitás* attribútuma. Az *autóbuszgarázs* esetén a *max. üzemanyag*, a *metró járműtelep* esetében a *vágányhossz* attribútumot rögzítjük.

3.1.4. Típusok

A 3.1. táblázatban láthattuk, hogy a *funkció* jellemzővel megkülönböztethetjük az egyes telephelyeket. Ha a *funkció* jellemzőt a modellelemek *típusának* tekintjük, akkor úgy is fogalmazhatunk, hogy a *vágányhossz* jellemző csak a *metró járműtelep* típusú elemekre értelmezett, a *max. üzemanyag* jellemző csak az *autóbuszgarázs* típusra. A típusokat felhasználhatjuk a gráfok jellemzésére is.

Példa. A korábbi metróhálózatot szeretnénk kiegészíteni a buszhálózatra vonatkozó információval. Szeretnénk tárolni azt is, hogy az egyes vonalakon közlekedő járművek melyik telephelyen parkolnak. Az autóbuszvonalon közelekedő járművek autóbuszgarázsban, a metróvonalon közlekedő járművek metró járműtelepen parkolnak.

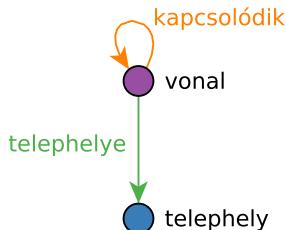
Készítsünk egy példánygráfot, amelyen ábrázoljuk az M1 és M3-as metróvonal, valamint a 7-es és a 8-as buszvonalak kapcsolódásait (*kapcsolódik*) élek, valamint azt, hogy az egyes vonalak melyik telephelyhez tartoznak (*telephelye*) élek.



3.7. ábra. Közlekedési vonalak és telephelyek

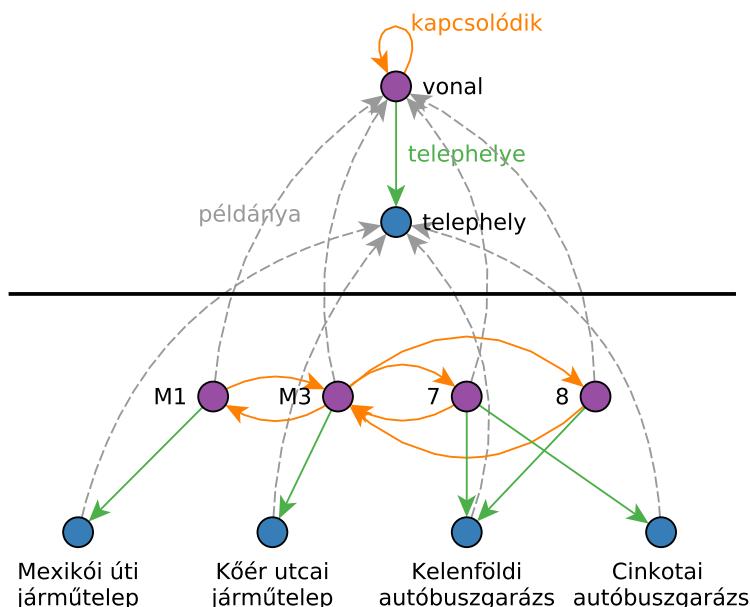
A példánygráfot követve a problémához tartozó *típusgráfban* meg kell jelennie a *vonal* és a *telephely* fogalmaknak. Az egyes vonalak kapcsolódhatnak egymáshoz a *kapcsolódik* él mentén, míg a vonal telephelyét a *telephelye* él jelzi.

Megjegyzés. Az egyes telephelyek tulajdonságait nem ábrázoltuk az ábrán.



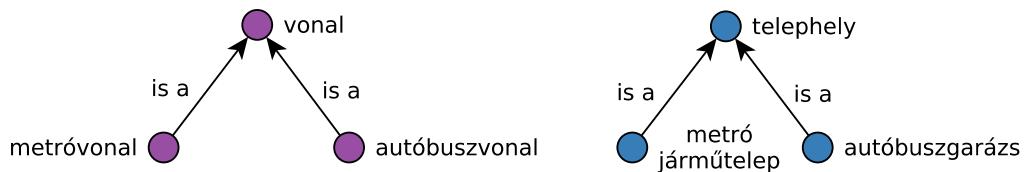
3.8. ábra. Közlekedési vonalak és telephelyek típusgráfja

A példánygráfot és a típusgráfot egy gráfban is jelölhetjük. Ekkor az egyes példányok és a típusok között egy *példánya* él fut.



3.9. ábra. Közlekedési vonalak és telephelyek példány- és típusgráfja

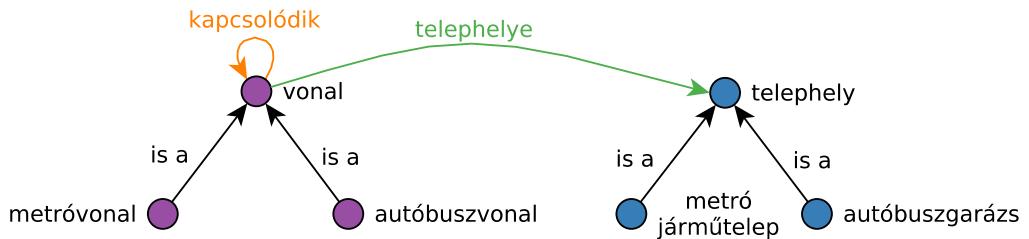
A 3.10. ábra bemutatja a problémához tartozó *típushierarchiát*. A típushierarchia egy hierarchikus modell, amely az egyes típusok leszármazási (*is a*) viszonyait ábrázolja.



3.10. ábra. Közlekedési vonalak és telephelyek típushierarchiája

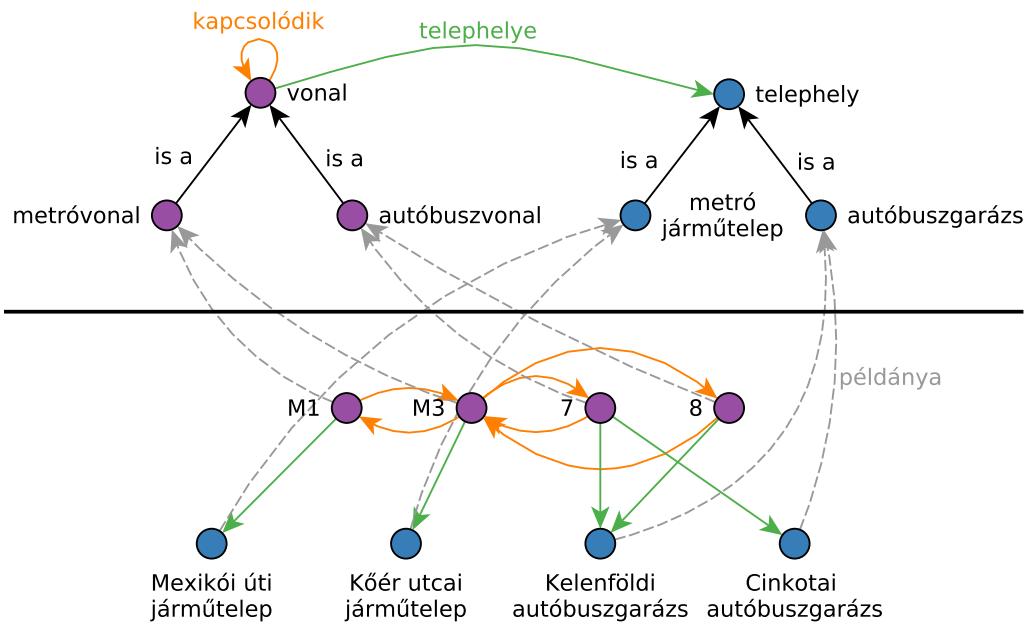
A rendszer fejlesztésekor gyakran fontos, hogy a típusgráfot és a típushierarchiát együtt lássuk. A *metamodell* tartalmazza a típusgráfot, a típusok hierarchiáját, a tulajdonságmodellt (ill. további, itt nem részletezett szabályokat, pl. multiplicitási kényszerek, jóformáltsági kényszerek).

A típusgráfban és a típushierarchiában tartalmazott információt, valamint tulajdonságmodellt együtt modellezve megkapjuk a terület metamodelljét. A metamodell részlete a 3.11. ábrán látható. Bár az ábrán a csomópontok elrendezése alapján következtethetünk arra, hogy melyik vonalakat látjuk, ezt az információt elvesztettük: ebben a reprezentációban az egyes metróvonalakat nem tudjuk megkülönböztetni.



3.11. ábra. Közlekedési vonalak és telephelyek (részleges) metamodellje

A példánymodellt és a metamodellt ábrázolhatjuk egyszerre is, a két modell elemei között a *példánya* viszony teremt kapcsolatot (szürke szaggatott nyíl).



3.12. ábra. Közlekedési vonalak és telephelyek példánymodellje és (részleges) metamodellje egy gráfon ábrázolva

Feladat. Az egyes járműtelepek is valamelyik városrészhez tartoznak. Képzítsük olyan metamodellt, amelyben ábrázolható az egyes járműtelepek és városrészek kapcsolata is!

3.2. A strukturális modellezés elmélete

Ahogy az eddigi példákban láttuk, a strukturális modellezés célja, hogy a rendszer felépítését jellemesse, beleérte az egyes elemek típusát, a közöttük lévő kapcsolatokat és hierarchiát, valamint az elemek tulajdonságait. Egy rendszer strukturális modellje tehát alkalmas arra, hogy az alábbi kérdésekre (nem feltétlenül kimerítő) választ nyújtson:

- Milyen elemekből áll a rendszer?
- Hogyan kapcsolódnak egymáshoz az elemek?
- Milyen hierarchia szerint épül fel a rendszer?
- Milyen tulajdonságúak a rendszer elemei?

A strukturális modellre az alábbi tömör definíciót adhatjuk.

Definíció. A *strukturális modell* a rendszer felépítésére (*struktúrájára*) vonatkozó tudás. A strukturális modell a rendszer alkotórészeire, ezek tulajdonságaira és egymással való viszonyaira vonatkozó statikus (tehát változást nem leíró) tudást reprezentál.

Megjegyzés. Fontos, hogy maga a strukturális modell változhat az idő során (pl. a metróhálózat fejlődik), de maga a modell nem ír le időbeli változásokat (pl. hogy miként mozognak a szerelvények).

A következőkben a korábbi példákra építve bemutatjuk a strukturális modellezés elméleti hátterét és precízen definiáljuk a szükséges fogalmakat.

3.2.1. Tulajdonságmodell

Definíció. A *jellemző* egy, a modell által megadott *parciális függvény*, amelyet a modellelemeken értelmezünk.

Megjegyzés. A H halmazon értelmezett *parciális függvény* nem jelent mást, mint a H valamely (nem megnevezett) részhalmazán értelmezett *függvény*. Konkrét esetünkben az egyes jellemzőket a modellelemek egy-egy részére értelmezzük (nem feltétlenül az összesre).

A jellemzőket gyakran táblázatos formában ábrázoljuk. Vizsgáljuk meg például a 3.1. táblázat jellemzőit:

azonosító	helyszín	funkció	kapacitás	vágányhossz	max. üzemany.
T1	Mexikói út	metró járműtelep	24	8 500 m	
T2	Kőér utcai	metró járműtelep	60	16 512 m	
T3	Cinkota	autóbuszgarázs	265		250 000 lit.
T4	Kelenföld	autóbuszgarázs	322		200 000 lit.

Megjegyzés. A tulajdonságmodell mögötti matematikai struktúra az ún. *reláció*. A reláció pontos halmazelméleti definíciójával és a relációkon értelmezett műveleteket definiáló *relációalgebrával* bővebben az *Adatbázisok* tárgy foglalkozik.

A modellelemeket az *azonosító* jellemzőjükkel különböztetjük meg egymástól. A *kapacitás* jellemzőhöz tartozó függvény $\text{kapacitás}(e) \rightarrow n$, ahol e egy modellelem azonosítója, n egy nemnegatív egész szám. Például

$$\text{kapacitás(T2)} = 60, \quad \text{kapacitás(T3)} = 265.$$

A *funkció* jellemzőhöz tartozó függvény $\text{funkció}(e) \rightarrow t$, ahol e egy modellelem azonosítója, t a {metró járműtelep, autóbuszgarázs} halmaz eleme. Például

$$\text{funkció(T2)} = \text{metró járműtelep}, \quad \text{funkció(T3)} = \text{autóbuszgarázs}.$$

A fentiekhez hasonlóan, a *vagányhossz* jellemzőhöz tartozó parciális függvény $\text{vagányhossz}(e) \rightarrow n$, ahol e egy modellelem azonosítója, n egy nemnegatív egész szám. Fontos különbség viszont az eddigiekhez képest, hogy ezt jellemzőt csak bizonyos modellelemekre értelmeztünk, másokra nem: a metró járműtelepek esetén vesz fel értéket, az autóbuszgarázsok esetén viszont nem. Vagyis látható, hogy a *vagányhossz* jellemző csak akkor vesz fel értéket, ha a *funkció* attribútum értéke metró járműtelep.

Ha tehát a *funkció* attribútumot a telephely *típusának* tekintjük, akkor úgy is fogalmazhatunk, hogy a *vagányhossz* jellemző csak a metró járműtelep típusú elemekre értelmezett. Általánosan:

Definíció. A *típus* egy kitüntetett jellemző, amely meghatározza, hogy milyen más jellemzők lehetnek értelmezettek az adott modellelemre, illetve milyen más modellelemekkel lehet kapcsolatban. A többi jellemzőt *tulajdonságának* hívjuk.

Megjegyzés. Jelen anyagban az egyszerűség kedvéért feltételezzük, hogy a modellelemeknek pontosan egy típusa van.

A modellünkben két típus van: az autóbuszgarázs és a járműtelep. Például a cinkotai telephely az autóbuszgarázs, míg a mexikói úti telephely a metró járműtelep típus példánya.

Definíció. Egy adott t típus *példányainak* nevezzük azon modellelemeket, amelyek típusa t .

A modellezési gyakorlatban elterjedt (de nem univerzális) megkötés, hogy az egyes elemek típusát állandónak tekintjük. Ha az elem típusa a rendszer működése során nem változhat meg, akkor olyan típust kell hozzárendelni, amely az elem teljes

életciklusa során előforduló összes lehetséges jellemzővel, kapcsolattal összhangban van.

3.2.2. Gráfmodellek

Formális definíciók (*). A definíciók támaszkodnak a gráfelmélet alapjaira, ezért röviden összefoglaljuk a felhasznált definíciókat.

A *gráf* egy olyan $G = (V, E)$ struktúra, ahol V halmaz a csomópontok, E az élek halmaza. Az élek csomópontok között futnak, *irányítatlan gráf* esetén az E halmaz csomópontok rendezetlen $\{v_1, v_2\}$ párjaiból áll (tehát nem különböztetjük meg a $\{v_1, v_2\}$ és a $\{v_2, v_1\}$) párokat, míg *irányított gráf* esetén csomópontok rendezett (v_1, v_2) párjaiból.

Címkézett gráf esetén a gráf elemeit (csomópontjait és/vagy éleit) *címkékkel* láthatjuk el. A címkézés célja lehet *egyedi azonosító* hozzárendelése vagy bizonyos tulajdonság leírása (pl. a csomópontok kapacitása, élek típusa). Ha precízen szeretnénk jellemzni a gráfot, a következő terminológiát használhatjuk: ha csak a csomópontokhoz rendelünk címkéket, *csúcscímkézett gráfról* beszélünk, míg ha csak a gráf éleihez rendelünk címkéket, *élcímkézett gráfról* beszélünk.

A típusok gráf jellegű modellek esetén is fontos szerepet játszanak. A gráfmodell elemeit (külön-külön a csomópontokat és az éleket is) típusokba sorolhatjuk. A típusok meghatározzák, hogy az egyes csomópontok milyen élekkel köthetők össze.

Az egyes csomópont- és éltípusok viszonyát gráfként is ábrázolhatjuk.

Definíció. A *típusgráf* egy olyan gráf, amelyben minden csomóponttípushoz egy típuscsomópont, minden éltípushoz egy típusél tartozik.

A gráfmodellekben is értelmezzük a *példány* fogalmát.

Definíció. Egy adott típusgráf *példánygráfja* alatt olyan gráfmodellt értünk, amelynek elemei a típusgráf csomópont- és éltípusainak példányai, valamint minden él forrása és célja rendre az éltípus forrásának és céljának példánya.

A típusgráfot és példánygráfot egy gráfon ábrázolva a típus-példány viszonyok is megjeleníthetők: a példánygráf csomópontjaiból a típusgráf csomópontjaira *instance of* (példánya) élek mutatnak. Szintén *instance of* viszony áll fenn a példánygráf élei és a típusgráf élei között.

Megjegyzés. Az *instance of* viszony helyett gyakran annak inverzét, a *type of* (x típusa y-nak) viszonyt ábrázoljuk. Gráfban ábrázolva az *instance of* és a *type of* élek adott csomópontok között egymással ellentétes irányúak.



3.13. ábra. Példány és típusgráf *type of* élekkel

Definíció. Egy rendszer *metamodellje* tartalmazza a típusgráfot, az egyes típusok közötti kapcsolatokat, ill. további megkötésekét is.

Megjegyzés. A további megkötések között szerepelhetnek jólformáltsági kényszerek, multiplicitási kényszerek stb. Ezekkel most nem foglalkozunk részletesen.

3.2.3. Hierarchia

Formális definíciók (*). A séta szomszédos csúcsok és élek váltakozó sorozata, mely csúccsal kezdődik és csúcsban végződik. Az út olyan séta, amely nem metszi önmagát, valamint első és utolsó csúcsa különbözik. A kör olyan séta, amely nem metszi önmagát, valamint első és utolsó csúcsa megegyezik. A körmentes, összefüggő gráfokat *fának* nevezzük. (A körmentes gráfokat *erdőnek* nevezzük.) A fák esetén gyakran kiemelt szerepet tulajdonítunk egy csomópontnak: a *gyökér csomópont* a fa egy megkülönböztetett csomópontja. A *gyökérfa* olyan fa, ami rendelkezik gyökér csomóponttal. *Gyökérfa*, *színtezett fa* esetén a fa csomópontjaihoz hozzárendeljük a gyökértől vett távolságukat is.

A hierarchikus modellezésben kiemelt szerepet játszanak az irányított körmentes gráfok. Egy gráf DAG (*directed acyclic graph*), ha nem tartalmaz irányított kört.

Egy rendszer hierarchiája a rendszer dekompozíciójával állítható elő.

Definíció. A *dekompozíció (faktoring)* egy rendszer kisebb *komponensekre* bontása, amelyek könnyebben érhetők, fejleszthetők és karbantarthatók.

A rendszer így kapott egyes komponensei (részrendszer) gyakran további dekompozícióval még kisebb részekre bonthatóak. Természetesen a dekompozíció során ügyelnünk kell arra, hogy az egyes részekből visszaállítható legyen az eredeti rendszer, különben a kapott strukturális modellünk hiányos.

Definíció. Egy dekompozíció *helyes*, ha a dekompozícióval kapott rendszer minden elemének megfeleltethető az eredeti rendszer valamelyik eleme, és az eredeti rendszer minden eleméhez hozzárendelhető a dekompozícióval kapott rendszer egy vagy több eleme.

3.3. Nézetek

A struktúramodellekből különböző *nézeteket* állíthatunk elő.

Tulajdonságmodellekben a leggyakrabban használt műveletek a *szűrés* és a *vetítés*. Ezek során úgy *absztraháljuk* a modellt, hogy bizonyos modellelemeket és/vagy azok egyes jellemzőit elhagyjuk.

Megjegyzés. A relációalgebrában a *szűrés* művelet neve *szelekció*, a *vetítés* művelet neve *projektíó*.

3.3.1. Szűrt nézet

Ismét idézzük fel a 3.1. táblázat telephelyeket tartalmazó tulajdonságmodelljét.

azonosító	helyszín	funkció	kapacitás	vágányhossz	max. üzemanyag
T1	Mexikói út	metró járműtelep	24	8 500 m	
T2	Kőér utcai	metró járműtelep	60	16 512 m	
T3	Cinkota	autóbuszgarázs	265		250 000 liter
T4	Kelenföld	autóbuszgarázs	322		200 000 liter

Definíció. A *szűrés* művelet során a modell elemein kiértékelünk egy feltételt és azokat tartjuk meg, amelyek megfelelnek a feltételnek.

Tulajdonságmodellek esetén a szűrés az elhagyott modellelemek a modell sorai lehetnek, gráf jellegű modellek esetén a gráf csúcsai vagy élei.

Tulajdonságmodell szűrése

Példa. Szeretnénk megtudni, hogy mely telephely alkalmas legalább 100 jármű befogására.

Azokra az elemekre szűrünk, amelyeknél a *kapacitás* jellemző értéke 100-nál nagyobb vagy egyenlő.

Az eredmény:

azonosító	helyszín	funkció	kapacitás	vágányhossz	max. üzemanyag
T3	Cinkota	autóbuszgarázs	265		250 000 liter
T4	Kelenföld	autóbuszgarázs	322		200 000 liter

Példa. Szeretnénk megtudni, hogy mely metró járműtelep alkalmas legalább 50 jármű befogására.

Végezzünk szűrést azokra a modellelemekre, ahol a *funkció* attribútum értéke *metró járműtelep*, a *kapacitás* attribútum értéke nagyobb vagy egyenlő 50-nél.

Az eredmény:

azonosító	helyszín	funkció	kapacitás	vágányhossz	max. üzemany.
T2	Kőér utcai	metró járműtelep	60	16 512 m	

Gráfmodell szűrése

Egy gráfmodellből a szűrés a modell egy részgráfját állítja elő.

Példa. Soroljuk fel az M2-es és az M4-es metró megállói.

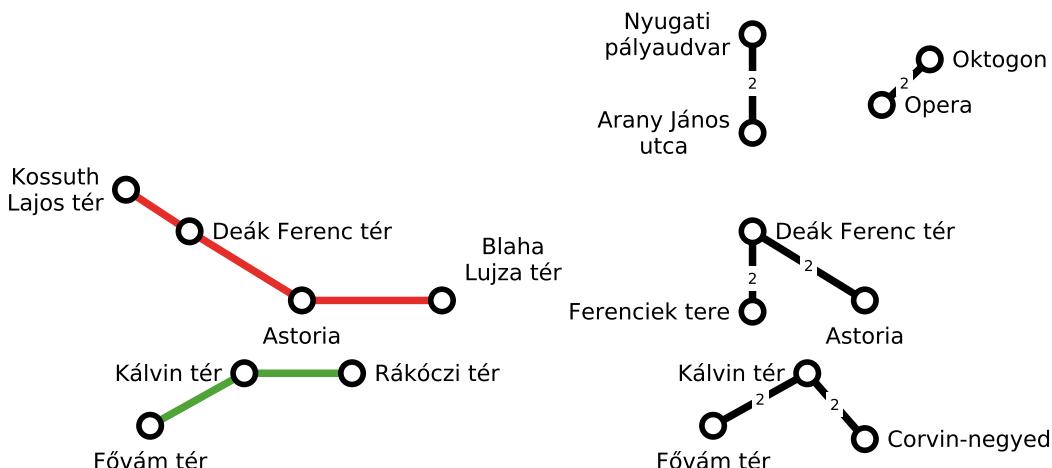
A modellen szűrést végzünk, ami csak azokat a csomópontokat tartja meg, amelyekhez tartozik olyan él, amelynek a címkéje M2 vagy M4. A kapott részgráf a 3.14(a) ábrán látható.

Példa. Határozzuk meg, hogy mely szomszédos metrómegállók között hosszabb egy percnél a menetidő.

A modellen szűrést végzünk, ami

- csak azokat a csomópontokat tartja meg, amelyekhez tartozik 1-nél nagyobb súlyú él,
- csak az 1-nél nagyobb súlyú éleket tartja meg.

A kapott részgráf a 3.14(b) ábrán látható.



3.14. ábra. Szűrések a metróhálózatot tartalmazó gráfon

3.3.2. Vetített nézet

Definíció. Vetítés során a modell egyes jellemzőit kiválasztjuk és a többet elhagyjuk a táblázatból.

Megjegyzés. Érvényes vetítés művelet az is, ha a tulajdonságmodell összes jellemzőjét tartjuk.

Tulajdonságmodell vetítése

Példa. Olyan kimutatásra van szükségünk, ami csak az egyes telephelyek helyszínét, funkcióját és kapacitását tartalmazza.

Végezzünk szűrést a tulajdonságmodellekben a *helyszín*, *funkció* és a *kapacitás* attribútumokra.

<i>helyszín</i>	<i>funkció</i>	<i>kapacitás</i>
Mexikói út	metró járműtelep	24
Kőér utcai	metró járműtelep	60
Cinkota	autóbuszgarázs	265
Kelenföld	autóbuszgarázs	322

3.4. Strukturális modellezési technikák

A modellezési formalizmusok után bemutatunk néhány strukturális modellezési technikát.

3.4.1. Hierarchia modellezése

Korábban láttuk, hogy a modellelemek közti szigorú hierarchia kifejezhető fa (ill. erdő) gráfokkal. Ezek a fajta modellek képesek kifejezni a (rész)rendszer és alkotóelemeik közti tartalmazási viszonyt, akár többszintű tartalmazással is (a részrendszer is további részeket tartalmaznak). Azonban a gyakorlatban a modellnek sokszor ennél jóval több információt kell tartalmaznia; egy-egy adott modellelemmel kapcsolatban nem csak a tartalmazó és tartalmazott komponenseivel való tartalmazási viszonyát kell ismerni, hanem egyéb modellelemekkel való kapcsolatait.

Ilyenkor a strukturális modell (gráf jellegű része) két rétegre tagozódik: egyrészt a modell szerkezeti vázát alkotó tartalmazási hierarchiára (fa/erdő), amely az alkotóelemek rész-egész viszonyait reprezentálja, másrészt ezen felüli *kereszthivatkozás* élekre, amelyek a tartalmazási rendtől függetlenül, a körmentesség korlátozása nélkül köthetnek össze elemeket. Ennek megfelelően egy metamodell megmondhatja,

hogy mely éltípusok példányait fogjuk az említett szerkezeti vázat alkotó tartalma- zási éleknek tekinteni.

A hierarchikus modellek megalkotása, illetve az összetett rendszerek tervezése során többféleképp választható meg az egyes elemek elkészítésének sorrendje. Jól illusztrálja a választási szabadságot két polárisan ellentétes megközelítés: a *top-down* és a *bottom-up* modellezés.

Definíció. *Top-down* modellezés során a rendszert felülről lefelé (összetett rendszerből az alkotóelemei felé haladva) építjük. A modellezés alaplépése a *dekompozíció*.

Egy top-down modellezési / tervezési folyamatot úgy kell tehát elképzelni, hogy a kezdetektől fogva az egész rendszer modelljét építjük; azonban eleinte hiányoznak még a részletek. Idővel a modellt finomítjuk: tartalmazott alkotóelemekre bontjuk a rendszert, megadva azok tulajdonságait és kereszthivatkozásait is; majd később magukat az alkotóelemeket ugyanúgy strukturális dekompozíciót vetjük alá.

A top-down modellezés fontos jellemzői:

- ⊕ Részrendszer tervezéskor a szerepe már ismert
- ⊖ „Félidezőben” még nincsenek működő (teljes részletességgel elkészített) részek
- ⊖ Részek problémái, igényei későn derülnek ki

Definíció. *Bottom-up* modellezés során a rendszert alulról felfelé (elszigetelt alkotóelemkből az összetett rendszer felé haladva) építjük. A modellezés alaplépése a *kompozíció*: az egész rendszer összeszerkesztése külön modellezett vagy fejlesztett részrendszerkből.

Egy bottom-up modellezési / tervezési folyamatot úgy kell tehát elképzelni, hogy a kezdetektől fogva részletes modelleket építünk; azonban eleinte csak a rendszer izolált, egyszerű komponenseivel foglalkozunk. Ahogy fokozatosan egyre több komponens készül el, nagyobb részrendszerük foglalhatjuk őket össze, az egymáshoz való kapcsolódásukat is tisztázva. Idővel az így kapott összetettebb részrendszereket is további kompozíciót vethetjük alá.

A bottom-up modellezés fontos jellemzői:

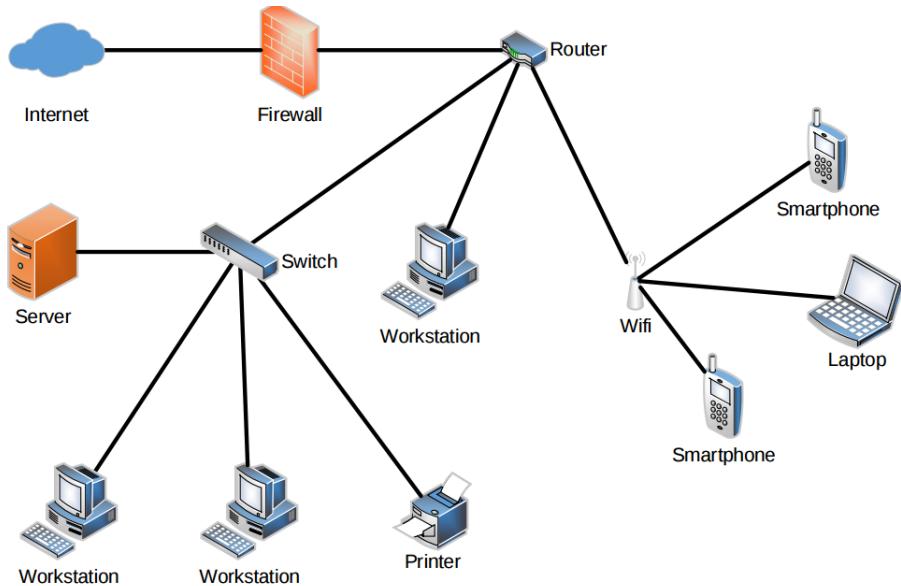
- ⊕ A rendszer részei önmagukban kipróbálhatók, tesztelhetők
- ⊕ Részleges készültségnél könnyebben előállítható a rendszer prototípusa
- ⊖ Nem látszik előre a rész szerepe az egészben

Természetesen a gyakorlatban a kétféle szélsőséges megközelítés közti kompro- misszum is elképzelhető.

3.5. Gyakorlati alkalmazások

3.5.1. Számítógéphálózat modellezése

Számítógép-hálózatok kiválóan modellezhetők gráfokkal, amelyben a gráf csomópontjai a hálózat elemei (pl. számítógép, router, tűzfal), a kapcsolatok pedig ezek összeköttetései.



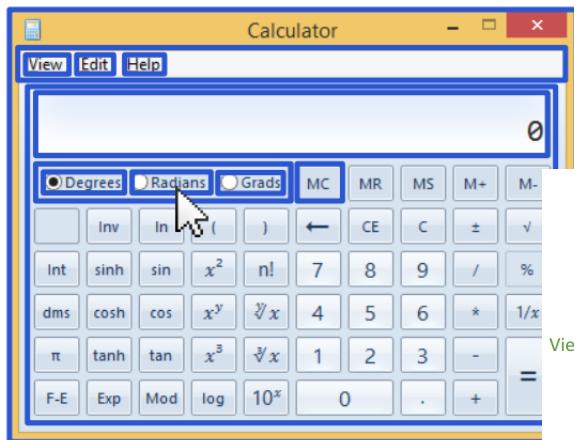
3.15. ábra. Hálózat

- Milyen elemekből áll a rendszer, milyen kapcsolatok lehetségesek?
- Van-e egyszeres hibapont a rendszerben?
- Milyen hosszú úton, milyen típusú elemeket érintve lehet elérni az internetet?
- Hány gép van a wifi hálózaton?
- Egy elem hibája meddig terjedhet?
- Elérhető-e az internet?

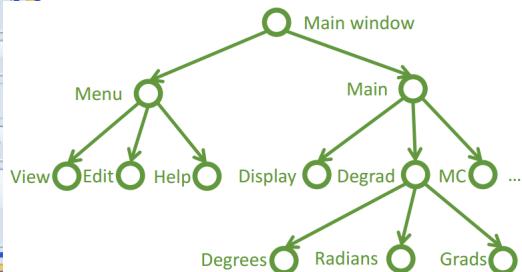
Feladat. Milyen típushierarchiát készíthetünk egy számítógép-hálózathoz?

3.5.2. Grafikus felhasználói felület

Egy szoftver alkalmazás *grafikus felhasználói felülete* (GUI) is egy hierarchikus modell.



(a) A számológép alkalmazás grafikus felülete



(b) A komponensek hierarchiája

3.16. ábra. A metróhálózatot ábrázoló gráf kiterjesztései

Feladat. Mi történik, amikor egy alkalmazás ablakán kattintunk – hogyan határozza meg a rendszer, hogy melyik komponensre kattintottunk?

3.6. Összefoglalás

A fejezetben bemutattuk *struktúra alapú modellezés* motivációját, a használt formalizmusokat és azok alkalmazásait. Ismertettük *típusok* fontosságát és a típusrendszer ábrázolásának lehetőségeit.

A következő fejezetekben a *viselkedés alapú modellezést* és annak formalizmusait mutatjuk be.

3.7. Kitekintés: technológiák és technikák*

Az alábbiakban bemutatunk néhány, a strukturális modellezés téma-köréhez kapcsolódó gyakorlati technológiát, specifikációt és eszközt. Az itt felsorolt fogalmak nem részei a számonkérésnek, gyakran előkerülnek viszont a későbbi tanulmányok és munkák során, ezért mindenképpen érdemes legalább névről ismerni őket.

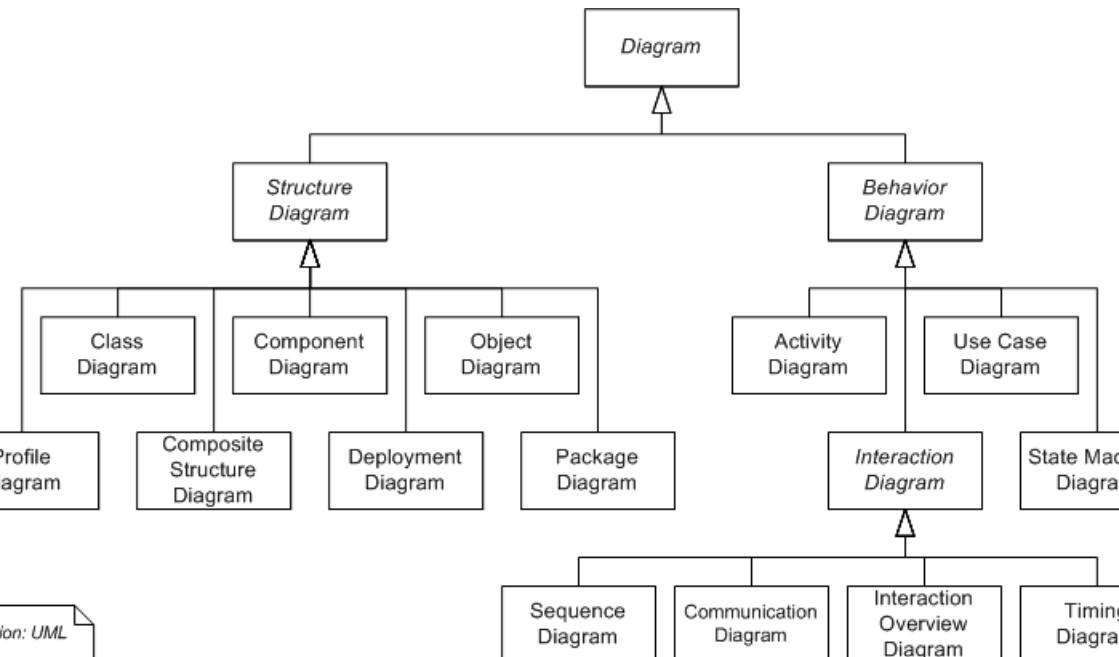
3.7.1. Technológiák

A gyakorlatban sokféle modellezési nyelvet és technológiát használnak. Ezek közül mutatunk be most azokat, amelyek a későbbi tanulmányok során előkerülnek.

UML

Az UML (*Unified Modeling Language*) egy általános célú modellezési nyelv az [?]. Az UML három fő diagramtípus definiál:

- *Structure Diagram*: strukturális modellek leírására. A *Class Diagram* az osztályok (metamodell), míg az *Object diagram* a példányok (modell) leírására alkalmas. A *Composite Structure Diagram* egy rendszer struktúráját és a rendszer komponenseinek kapcsolatát mutatja be.
- *Behaviour Diagram*: viselkedési modellek leírására, pl. a *State Machine Diagram* segítségével állapotgépek az *Activity Diagramon* folyamatok ábrázolhatók. A *Behaviour Diagramok* között megkülönböztetjük az *Interaction Diagramokat*. Ezeknek szintén a viselkedés leírása a célja, de a hangsúly a vezérlés- és adatáramlás bemutatásán van. Ilyen pl. a *Sequence Diagram* (szekvenciadiagram), amely az egyes objektumok közötti interakciót mutatja be üzenetek formájában.



3.17. ábra. UML diagramok típusai és a közöttük lévő viszony osztálydiagramként ábrázolva [?]

Az UML nyelvvel részletesen foglalkozik a *Szoftvertechnológia* tárgy. A nyelvről egy jó összefoglalót a [?] oldal.

EMF

Az Eclipse fejlesztőkörnyezet¹ saját modellezési keretrendszerrel rendelkezik, ez az EMF (*Eclipse Modeling Framework*). Az EMF metamodellező nyelve, az Ecore lehetővé teszi saját, ún. szakterület-specifikus nyelv (*domain-specific language*, DSL) definiálását. Az EMF mára több területen is *de facto* modellezési keretrendszer.

Az Eclipse fejlesztőkörnyezettel és az EMF keretrendszerrel foglalkozik az *Eclipse* alapú technológiák szabadon választható tárgy.

3.7.2. Haladó strukturális modellezési technikák

A struktúramodellek definiálására és fejlesztésére különböző technikák léteznek. Korábban tárgyaltuk a *top-down* és a *bottom-up* tervezést. Itt két további, általánosan alkalmazható technikát mutatunk be.

Tervezési minták

Az *objektum-orientált* tervezés során gyakran előforduló problémákra különböző tervezési minták (*design patterns*) léteznek. A tervezési minták között külön szerepet kapnak a rendszer struktúráját leíró szerkezeti minták (*structural patterns*). A tervezési mintákkal bővebben a *Szoftvertechnikák* tárgy foglalkozik.

Refactoring

A *dekompozícióhoz*, azaz *faktoringhoz* szorosan kapcsolódik a *refactoring* (*refactoring*) fogalma [?]. Refactoringon egy rendszert definiáló programkód vagy modell átalakítását értjük. A refactoring lényege, hogy az átalakítás során a rendszer megfigyelhető működése változatlan marad, de a kapott programkód vagy modell érthetőbb, karbantarthatóbb lesz. Tipikus refactoring műveletek pl. változók átnevezése, ismétlődő programrészletek megszüntetése (pl. külön metódusba vagy osztályba kiszervezéssel).

3.7.3. Struktúramodellező eszközök és vizualizáció

Gráfok automatikus megjelenítésére alkalmas pl. a GraphViz² programcsomag. Gráfok feldolgozására gyakran alkalmazzák az igraph³ programcsomagot. Manapság több gráfadatbázis-kezelő rendszer is elterjedt, pl. a Neo4j⁴ és a Titan⁵ rendszerek.

¹<http://www.eclipse.org/>

²<http://www.graphviz.org/>

³<http://igraph.org/>

⁴<http://neo4j.com/>

⁵<http://thinkaurelius.github.io/titan/>

Gráfok manuális rajzolására szintén több eszköz elterjedt. Egyszerűen használható online felületet biztosít a draw.io⁶ és az Arrow Tools⁷. A jegyzetben szereplő ábrák a yEd eszközzel⁸ készültek. Sok információt tartalmazó gráf esetén érdemes lehet vektorgrafikus rajzoló-, ill. prezentáló eszközök, pl. a Microsoft Visio vagy Microsoft PowerPoint alkalmazás.

3.8. Elméleti kitekintés*

A strukturális modellezésnek komoly matematikai eszköztára is van. Az alábbiakban ezekből mutatunk be néhány részletet.

3.8.1. Bináris relációk tulajdonságai

Egy (D_1, D_2) halmazpáron értelmezett R bináris relációt úgy definiálhatunk, mint ezen halmazok Descartes-szorzatának egy részhalmazát: $R \subseteq D_1 \times D_2$.

Hasznos ismerni a kétváltozós relációkon értelmezett tulajdonságokat [?].

Tipp. Az elnevezések egyezése nem véletlen, a kétváltozós reláció egy alesete a reláció fogalomnak.

Definíció. Egy S halmazon értelmezett r kétváltozós reláció *reflexív*, ha bár-mely $a \in S$ -re $r(a, a)$ teljesül.

Definíció. Egy S halmazon értelmezett r kétváltozós reláció *szimmetrikus*, ha bármely $a, b \in S$ -re $r(a, b)$ teljesülése esetén $r(b, a)$ is teljesül. A nem szimmetrikus relációkat *aszimmetrikusnak* nevezzük.

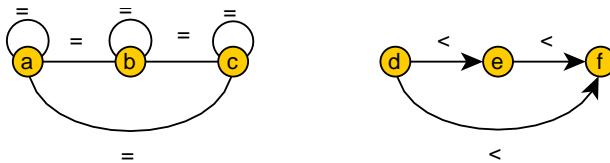
Definíció. Egy S halmazon értelmezett r kétváltozós reláció *tranzitív*, ha bár-mely $a, b, c \in S$ -re $r(a, b)$ és $r(b, c)$ teljesülése esetén $r(a, c)$ is teljesül.

⁶<http://draw.io/>

⁷<http://www.apcjones.com/arrows/>

⁸<https://www.yworks.com/products/yed>

Példa.



3.18. ábra. Az *egyenlő* (=) és a *kisebb* (<) relációk gráfban ábrázolva

Tranzitív relációk:

- Az *egyenlő* (=) és a *kisebb* (<) relációk tranzitívak, mert
 - o $a = b$ és $b = c$ esetén $a = c$,
 - o $d < e$ és $e < f$ esetén $e < f$.

A 3.18. ábrán ábrázoltuk a fenti relációkat. Az = reláció szimmetrikus, ezért iránytalan gráffal, a < reláció aszimmetrikus, ezért irányított gráf-fal reprezentálható.

Nem tranzitív relációk:

- A *nemegyenlő* reláció (\neq) nem tranzitív, mert
 - o $a \neq b$ és $b \neq c$ esetén $a \neq c$ nem mindenkorán áll fenn, például
 - o $1 \neq 2$ és $2 \neq 1$ esetén $1 \neq 1$ nem teljesül.
- Személyek közötti ōse reláció tranzitív, mert $\text{óse}(a, b)$ és $\text{óse}(b, c)$ esetén $\text{óse}(a, c)$ is fennáll.
- Személyek közötti *ismerőse* reláció nem tranzitív, mert $\text{ismerőse}(a, b)$ és $\text{ismerőse}(b, c)$ esetén nem garantált, hogy $\text{ismerőse}(a, c)$ fennáll.

3.9. Ajánlott irodalom

A gráfelmélettel behatóan foglalkozik a *Bevezetés a számításelméletbe 2.* tantárgy és Fleiner Tamás jegyzete [?]. Különböző gráfalgoritmusokat mutat be – pl. *legrövidebb út* és minimális összsúlyú *feszítőfa* keresésére – az *Algoritmuselmélet* tárgy. További keresőalgoritmusok a *Mesterséges intelligencia* tárgyban szerepelnek.

Olvasmányos összefoglalót nyújt az UML nyelvről Martin Fowler „UML distilled” című könyve [?].

A tulajdonsággráfokról egy jól érthető tudományos cikk Marko Rodriguez és Peter Neubauer munkája [?]. Rodriguez a Titan elosztott gráfadatbázis-kezelő rendszer egyik fő fejlesztője, míg Neubauer a Neo4j gráfadatbázis-kezelőt fejlesztő cég alapítója (mindkét rendszert említettük a 3.7.3. szakaszban). Az elosztott gráfadatbázis alkalmazását, elméleti és gyakorlati kihívásait kiváló prezentációkban mutatja be [?, ?].

Barabási-Albert László magyar fizikus nemzetközileg elismert kutatója a komplex hálózatok elméletének. Barabási „Behálózva” című könyve közérthető stílusban mutatja be a hálózatok elemzésének elméleti kihívásait a kutatási eredmények gyakorlati jelentőségét [?]. A szerzővel több interjú is készült.^{9,10,11}

Az osztályok és prototípusok közötti elvi különbséget mutatja be Antero Taivalsaari, a Nokia Research fejlesztőjének cikke [?].

⁹<http://index.hu/tudomany/bal080429/>

¹⁰http://index.hu/tudomany/2010/06/02/az_elso_cikk_utan_majdnem_leharaptak_a_fejunket/

¹¹http://index.hu/tudomany/2015/02/20/az_nsa_primitiven_hasznalta_a_begyujtott_adatokat/

4. fejezet

Állapotálapú modellezés

Az alábbi dokumentum egy háromfényű közúti jelzőlámpa fokozatosan kidolgozott modelljén keresztül mutatja be az állapot alapú modellezés alapfogalmait.

A bemutatott modellek a Yakindu Statechart Tools¹ eszközzel készültek.

4.1. Egyszerű állapotgépek

A példa kidolgozását azzal az egyszerű esettel kezdjük, amikor a modellező nyelv lényegében a *Digitális technika* tárgyból megismert *Mealy-automata* formalizmus.

4.1.1. Állapottér

Ehhez első lépéseként meg kell határozni a rendszer *állapotterét*. Az állapottér elemeit *állapotoknak* nevezzük. Az állapottérnek az alábbi két kritériumnak kell megfelelnie:

Definíció. *Teljesség.* minden időpontban az állapottér legalább egy eleme jellemzi a rendszert.

Definíció. *Kizárolagosság.* minden időpontban az állapottér legfeljebb egy eleme jellemzi a rendszert.

Egy adott időpontban a rendszer *pillanatnyi állapota* az állapottér azon egyetlen eleme, amelyik abban az időpontban jellemző a rendszerre. A rendszer egy *kezdőállapota* olyan állapot, amely a vizsgálatunk kezdetekor (például a $t = 0$ időpillanatban) pillanatnyi állapot lehet.

¹<http://statecharts.org>

Példa. Határozzuk meg egy jelzőlámpa egyszerű állapotterét!

- Off: kikapcsolt állapot
- Stop: piros jelzés
- Prepare: piros-sárga jelzés
- Go: zöld jelzés
- Continue: sárga jelzés

Kezdetben a rendszer legyen kikapcsolva, vagyis a rendszer egyetlen kezdőállapota legyen Off.

4.1.2. Állapotátmenet, esemény

A rendszer időbeli viselkedésében kulcsfontosságú, hogy a pillanatnyi állapot hogyan változik az idővel. Bizonyos mérnöki diszciplínákban ez a változás folytonos függvényel jellemezhető (ilyen rendszerekkel a Rendszerelméletcímű tárgy foglalkozik részletesebben). Például egy repülő állapota lehet a tengerszint feletti magasság, amely egy időben folytonosan változó mennyisége. Azonban az informatikai gyakorlatban a *diszkrét állapottereknek* van kiemelt jelentősége, ahol nem létezik folytonos átmenet az állapotok között, tehát a rendszer pillanatnyi állapota mindenkor addig állandó, amíg egy pillanatszerű esemény hatására egy másik állapotba át nem megy.

Az ilyen diszkrét rendszerek viselkedése *állapotátmenetekkel* (más néven *tranzíciókkal*) jól modellezhető. Egy állapotátmenet megengedi, hogy a rendszer állapotot váltszon egy forrás- és egy célállapot között. Amennyiben a rendszer pillanatnyi állapota a forrásállapot, az állapotátmenet *tüzelését* követően a rendszer új állapota a célállapot lesz. (Az, hogy a tüzelés pillanatában a rendszer pillanatnyi állapota a forrás- vagy a célállapot-e, megállapodás kérdése.)

Egy állapotátmenet tüzelését egy adott *esemény* bekövetkezte váltja ki. (Ennek speciális esete a spontán állapotátmenet, amikor a kiváltó esemény kívülről nem megfigyelhető.) Ezen felül állapotátmenetek *akciókat* hajthatnak végre, például maguk is válhatnak ki eseményeket. Sokszor praktikus egy adott rendszer szempontjából bemenet és kimeneti események elkülönítése.

Példa. Definiáljuk a jelzőlámpa modelljéhez az alábbi bemeneti eseményeket:

- onOff: ki- és bekapcsolás kérése
- switchPhase: jelzésváltás kérése

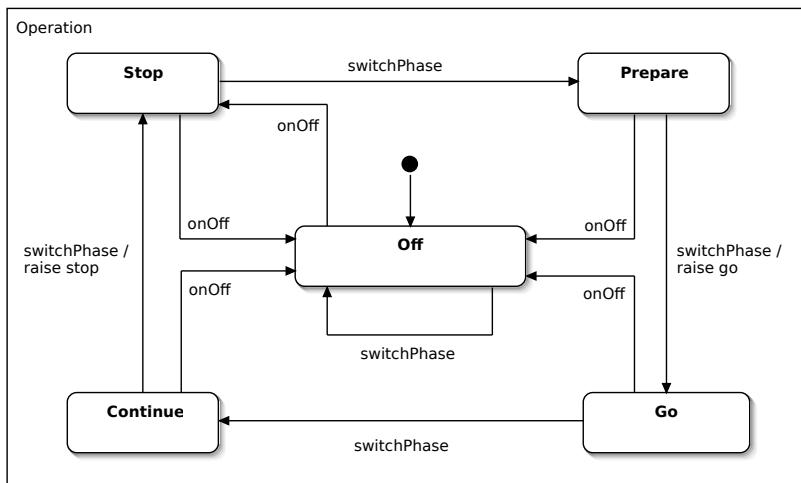
A jelzőlámpa a működését a balesetek reprodukálását segítendő folyamatosan naplózza egy külső fekete dobozba. Ennek megfelelően legyenek a rendszer output eseményei a következők:

- Stop: a rendszer sárga jelzésből vörös jelzésbe váltott
- Go: a rendszer zöld jelzésbe váltott

Az események definíciója a Yakindu szerkesztőjében a következőképpen adható meg:

```
interface:  
  in event onOff  
  in event switchPhase  
  out event stop  
  out event go
```

Ekkor a jelzőlámpa modellezhető a 4.1. ábra állapotgépével.



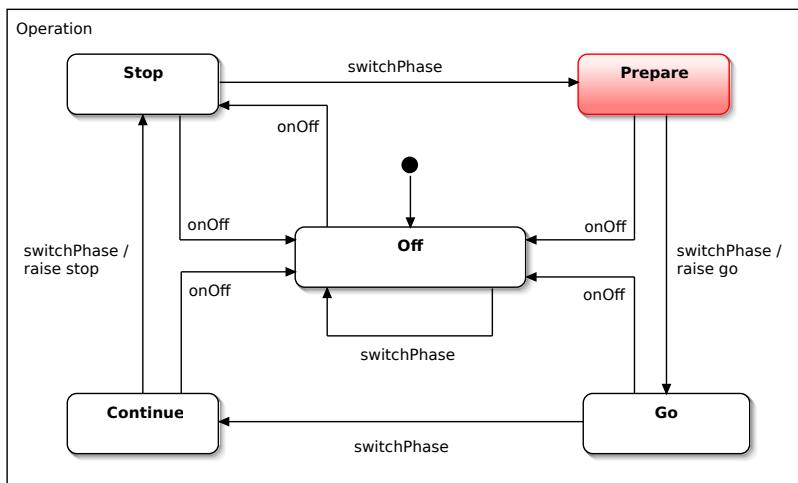
4.1. ábra. Jelzőlámpa egyszerű állapotgépe

A diagramon a rendszer állapotait (Off, Stop, Prepare, Go, Continue) lekerekített téglalapok jelölik. Az állapotgép kezdőállapotát (Off) a tömör fekete korongból húzott nyíl jelöli ki. A téglalapok között húzott nyílak a megfelelő állapotok közötti tranzíciókat jelképezik. A nyílakra írt címek a tranzíciót kiváltó, illetve a kiváltott eseményekre hivatkoznak (Yakinduban az esemény kiváltását `raise` kulcsszó jelöli).

Amint azt a fenti definíciókból láthatjuk, az „állapot” kifejezés két különböző jelentéssel bír:

- *Szintaktikai jelentés*. Az állapotgráf egy csomópontja, melyet lekerekített tég-lalap jelöl (*állapotcsomópont*).
- *Szemantikai jelentés*. Az állapottér egy eleme.

Egyszerű állapotgépek esetében elmondható, hogy a két fogalom által jelölt objektumok megfeleltethetőek egymásnak. Az állapotgép formalizmus új szintaktikai elemekkel történő kiterjesztésével (változók, összetett állapotok, ortogonális állapotok – ld. később) ugyanakkor ez a kapcsolat a továbbiakban nem áll fenn.



4.2. ábra. Pillanatnyi állapot nyomonkövetése szimulációval

Tipp. A Yakindu eszköz lehetőséget biztosít az állapotgép *szimulációjára*. Szimuláció során nyomon tudjuk követni, hogy a adott események hatására időben hogyan alakul a rendszer pillanatnyi állapota.

Például az **onOff**, majd **switchPhase** események a szimulátor felületén történő kiváltását követően a rendszer **Prepare** állapotba kerül, amit a Yakindu az állapotot jelképező téglalap átszínezésével ábrázol (ld. 4.2. ábra).

A fenti modell két fontos tulajdonsággal bír:

Definíció. Determinisztikus. Az állapotgépnek legfeljebb egy kezdőállapota van, valamint bármely állapotban, bármely bemeneti esemény bekövetkeztekor legfeljebb egy tranzíció tüzelhet.

Megjegyzés. A Yakindu csak determinisztikus modellek létrehozását támogatja.

Definíció. Teljesen specifikált. Az állapotgépnek legalább egy kezdőállapota van, valamint bármely állapotban, bármely bemeneti esemény bekövetkeztekor legalább egy tranzíció tüzelhet.

Megjegyzés. Ennek egyik következménye, hogy a rendszer *holtpontmentes*, azaz nem tartalmaz olyan állapotot, amelyből nem vezet ki tranzíció.

4.1.3. Végrehajtási szekvencia

A rendszer időbeli viselkedését annak *végrehajtási szekvenciái* jellemzik. Egy végrehajtási szekvencia állapotok és események egy

$$s_0 \xrightarrow{i_0/o_0} s_1 \xrightarrow{i_1/o_1} \dots$$

(véges vagy végtelen) alternáló sorozata, ahol s_0 a rendszer egy kezdőállapota, $s_j \xrightarrow{i_j/o_j} s_{j+1}$ pedig a rendszer egy állapotátmenete minden j -re. Egy állapot *elérhető*, ha a rendszernek létezik véges végrehajtási szekvenciája az állapotba.

Példa. Az $Off \xrightarrow{onOff} Stop \xrightarrow{switchPhase} Prepare \xrightarrow{switchPhase/go} Go$ sorozat a jelző egy véges végrehajtási szekvenciája; ennek megfelelően biztosan tudjuk, hogy például a *Go* állapot elérhető állapot. Az $Off \xrightarrow{onOff} Stop \xrightarrow{onOff} Off \xrightarrow{onOff} \dots$ egy végtelen végrehajtási szekvencia.

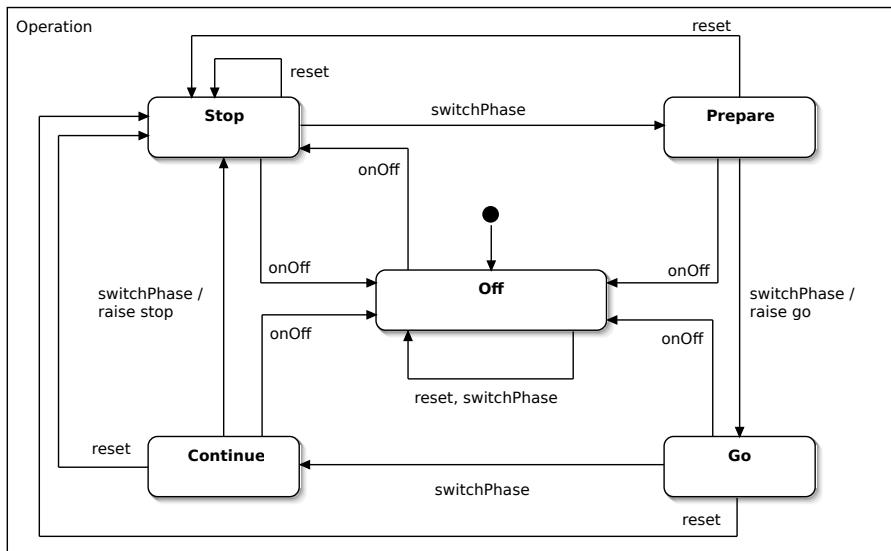
Az állapotgép végrehajtási szekvenciái vezérelten bejárhatóak szimuláció segítségével, ami jó módöt ad az állapotgép ellenőrzésére. A Yakindu beépített szimulátora erre lehetőséget biztosít.

4.2. Hierarchia

Példa. Egészítük ki a fenti modellt egy *reset* bemeneti eseménnyel, melynek hatására bekapcsolt állapotban a jelzőlámpa alaphelyzetbe (*Stop* állapotba) áll.

Az eddig megismert eszközökkel elkészített modellt szemlélteti a 4.3. ábra.

Vegyük észre, hogy a rendszer a *reset* eseményre a *Stop*, *Prepare*, *Go* és *Continue* állapotok mindegyikében egyformán viselkedik. Ebben az esetben ez annak köszönhető, hogy ezen állapotok mindegyikében a rendszer bekapcsolt állapotban van. Ezt a kapcsolatot a modellben explicit módon meg lehet jeleníteni egy *összetett állapot* bevezetésével, amely a négy állapot közös tulajdonságait és viselkedését általánosítja.



4.3. ábra. Jelzőlámpa állapotgépe reset eseménnyel 💀

4.2.1. Összetett állapot, állapotrégió

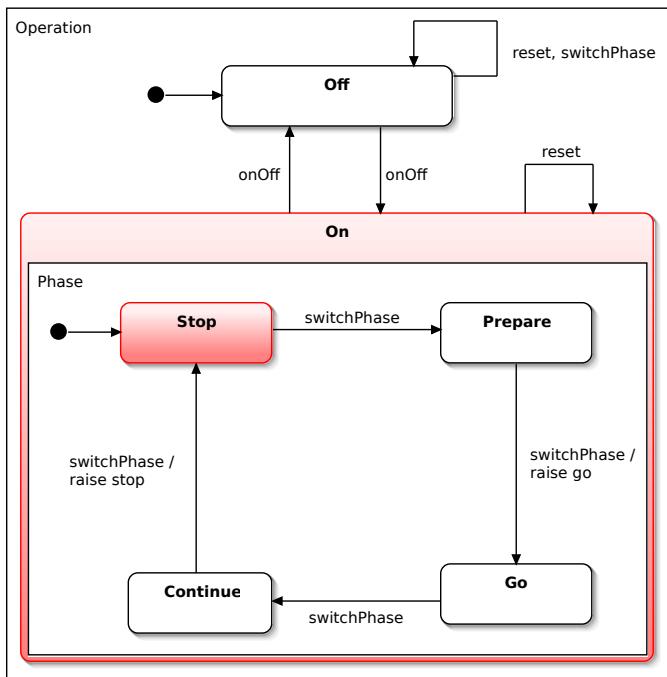
Egy összetett állapot szintaktikailag megfelel egy egyszerű állapotnak, azzal a kivétellel, hogy saját régióval rendelkezik, mely további állapotokat (beleértve a kezdőállapotokat) és köztük tranzíciókat tartalmazhat. Régiók közül kiemelt jelentőséggel bír a legfelső szintű régió, mely magát az állapotgépet tartalmazza.

A 4.4. ábra szemlélteti az On összetett állapot bevezetésével kapott modellt.

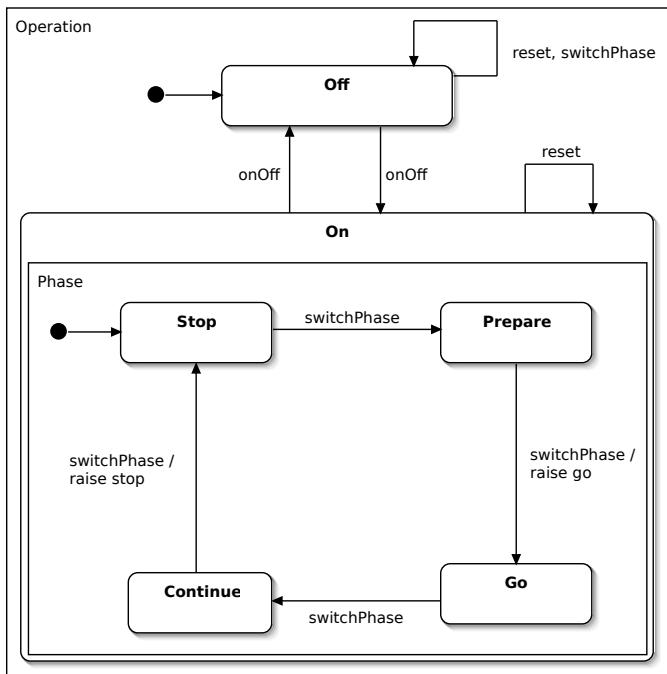
A teljes állapotgépet az Operation, míg az összetett állapot belséjét a Phase régió tartalmazza. A Phase régió kezdőállapota a Stop állapot, így a régióba való belépéskor ez az állapot lesz a rendszer pillanatnyi állapota. A reset esemény az On állapotból On állapotba vezet, így hatására valóban a Stop állapot lesz aktív.

A fenti példát szimulálva az onOff esemény Off állapotban történő fogadását követően az elvárásoknak megfelelően a Stop állapot a rendszer pillanatnyi állapota lesz. Ugyanakkor a Stop állapotot tartalmazó On állapot is pillanatnyi állapot lesz:

Általánosságban ha egy tartalmazott (egyszerű vagy összetett) állapot aktív, akkor az őt tartalmazó összetett állapot is aktív. Ezt fejezi ki az **állapotkonfiguráció** fogalma, ami állapotok egy olyan maximális (azaz nem bővíthető) halmaza, melyek egyszerre lehetnek aktívak a rendszerben.



4.4. ábra. Hierarchikus állapot pillanatnyi állapotként



4.5. ábra. Jelzőlámpa állapotgépe összetett állapottal

Példa. A jelzőlámpa állapotkonfigurációi:

- { Off }
- { On, Stop }
- { On, Prepare }
- { On, Go }
- { On, Continue }

Tartalmazó és tartalmazott állapot között tehát nem érvényesül a kizárolagosság. Ennek megfelelően a hierarchikus állapot bevezetésével többféle érvényes állapottér adódik.

Példa. A jelzőlámpa érvényes állapotterei:

- { Off, On }
- { Off, Stop, Prepare, Go, Continue }

Ezen felül az állapotkonfigurációk halmaza is tekinthető állapottérnek:

- { { Off }, { On, Stop }, { On, Prepare }, { On, Go }, { On, Continue } }

Ugyanakkor az { Off, On, Stop, Prepare, Go, Continue } nem jó állapottér, hiszen ebben az esetben például az { On, Prepare } részhalmaz sérti a kizárolagosságot. Általános szabály, hogy egy állapottér vagy az összetett állapotot, vagy annak összes részállapotát tartalmazza, de nem minden változatot.

A { Stop, Prepare, Go, Continue } részállapottér az On állapotot *finomítja*, míg az On állapot a { Stop, Prepare, Go, Continue } állapotokat *absztrahálja*. Jó modellezési gyakorlat a modell állapotainak fokozatos finomítása.

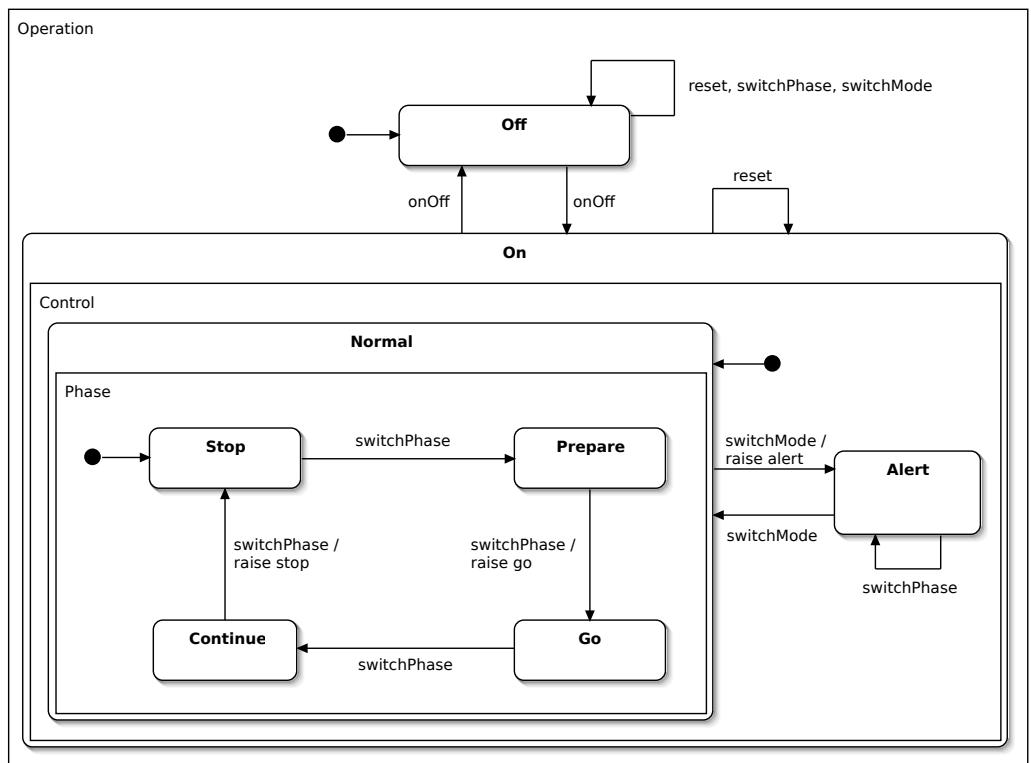
4.2.2. Többszintű állapothierarchia

Amint a következő példából kiderül, az állapotok közötti hierarchia nem feltétlenül egyszintű.

Példa. A 4.6. ábrán szemléltetett modell az On állapotot tovább bővíti egy Alert állapottal, mely a jelző sárgán villogó, figyelemzettelő állapotát modellezte. A rendes üzem jelzéseit a Normal összetett állapot tartalmazza. A Normal és Alert állapotok közötti váltás a switchMode bemeneti esemény hatására történik, a *Normal → Alert* állapotváltás Alert kimeneti eseményt vált ki.

4.3. Változók és őrfeltételek

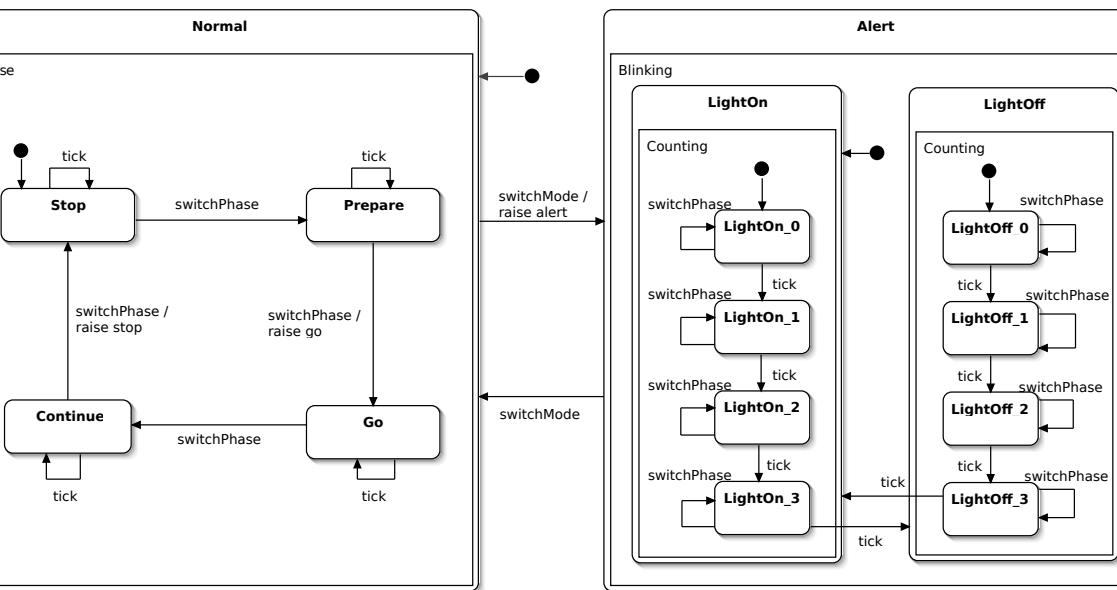
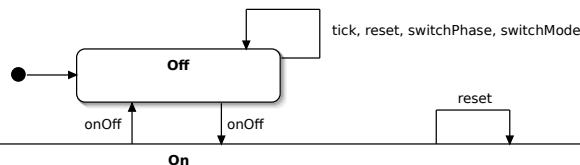
Finomítsuk az Alert állapotot LightOn és LightOff alállapotokkal, melyek fél másodpercenként váltakozva a sárga fény villogását modellezik! A villogás modellezéséhez



4.6. ábra. Jelzőlámpa állapotgépe többszintű összetett állapottal

feltételezzünk egy tick bejövő eseményt, amely a specifikáció szerint egy 8 Hz frekvenciájú órajel, tehát egyenletes ütemben, másodpercenként nyolcszor jelez.

Ahhoz, hogy a 2 Hz-es váltakozást modellezzük, a tick esemény minden negyedik bekövetkeztekor kell váltani LightOn és LightOff között. Ezért az állapotokat tovább kell finomítanunk, hogy a bekövetkező órajeleket számolni tudjuk.



4.7. ábra. Villogó jelzés külső órával

Egy lehetséges megvalósítást szemléltet a 4.7. ábra.

A fenti állapotgép valóban a kívánt viselkedést modellezi: az állapotgép Alert állapotban pontosan minden negyedik tick eseményre vált LightOn és LightOff állapotok között.

Abban az esetben azonban, ha csak minden századik eseményre kellene váltani LightOn és LightOff között, minden két összetett állapot száz alállapotot tartalmazna. Látható tehát, hogy ez nem lesz jó modellezési gyakorlat, hiszen az így bemutatott

modell elkészítése nagy erőfeszítést igényel, sok hibalehetőséget rejt és nehezen átlátható; valamint a számszerű paraméterek megváltozása esetén jelentős módosítást igényel.

4.3.1. Belső változók

A probléma megoldható *változók* alkalmazásával. A változó típussal rendelkezik, ez Yakinduban lehet **boolean**, **integer**, **real** vagy **string**, ezek a programozási nyelveknél megszokott logikai, egész, lebegőpontos, illetve karakterlánc típusoknak felelnek meg. Változók jelenlétében a rendszer állapotát már nemcsak a vezérlés állapota (állapot csomópontok), hanem az éppen érvényes *változóértékelés* is meghatározza.

Példa. A bekövetkezett tick események számlálására a rendszer interfészdefinícióiát kiegészítük a counter egész típusú változóval:

```
var counter : integer
```

A változó értékét *utasítással* lehet módosítani, amely – az eseménykiváltáshoz hasonlóan – tranzícióhoz kapcsolható *akció*. Akció ezen felül kapcsolható állapothoz is az entry és exit triggerek segítségével, melyek az állapotba való belépéskor, illetve az állapotból történő kilépéskor aktiválódnak.

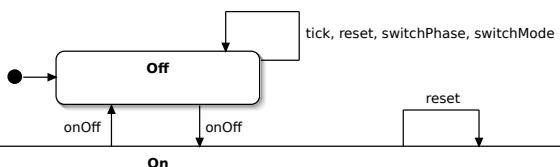
Azt, hogy a változó aktuális értéke befolyással lehessen a vezérlésre, a tranzícióra felírt örfeltételekkel lehet megvalósítani. Az örfeltétel biztosítja, hogy a tranzíció csak akkor tüzelhessen, ha az örfeltételbe felírt logikai kifejezést az aktuális állapot változóértékelése kielégíti.

Példa. A tick események számlálása változóval megvalósítható a 4.8. ábra állapotgépe szerint. Figyeljük meg, hogy a változó értékét szögletes zárójelekbe tett örfeltételek használják.

Példa. A fenti rendszer egy végrehajtásiszekvencia-részlete:

$$\begin{aligned} \langle LightOn, \{counter \mapsto 0\} \rangle &\xrightarrow{\text{tick}} \langle LightOn, \{counter \mapsto 1\} \rangle \xrightarrow{\text{tick}} \langle LightOn, \{counter \mapsto 2\} \rangle \\ \langle LightOn, \{counter \mapsto 3\} \rangle &\xrightarrow{\text{tick}} \langle LightOff, \{counter \mapsto 0\} \rangle \end{aligned}$$

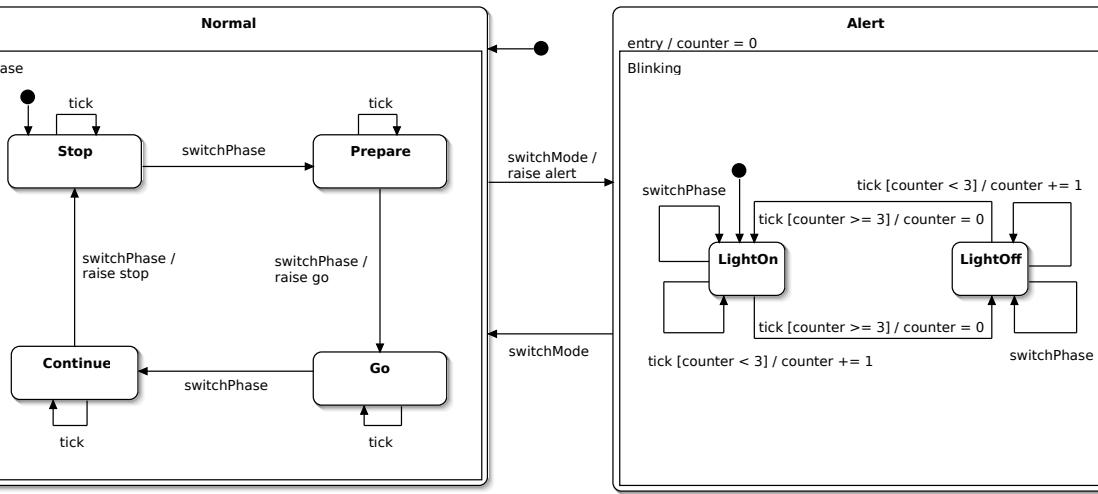
Feladat. Hogyan módosul örfeltételek jelenlétében a *determinizmus* definíciója?



ion

trol

Phase

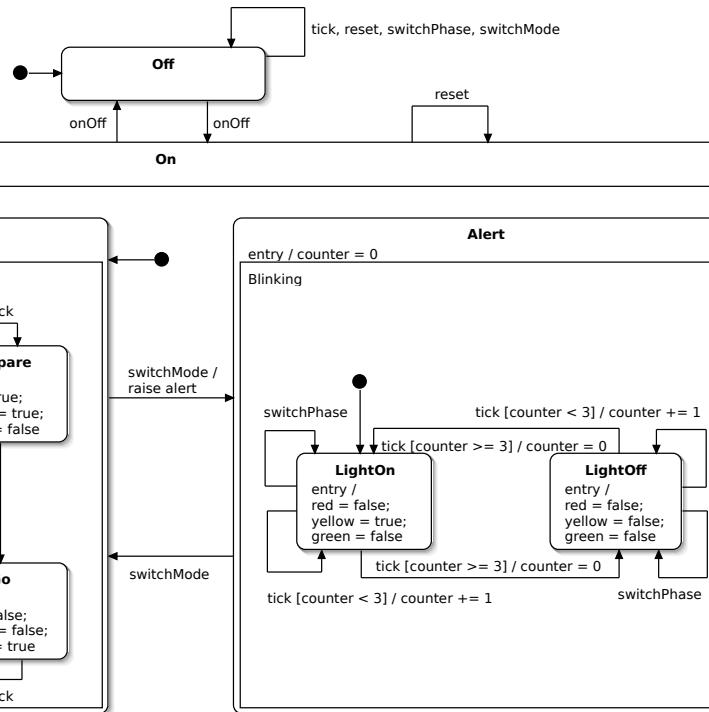


4.8. ábra. Villogó jelzés számlálóval

4.3.2. Interfészváltozók

A vörös, sárga ill. zöld színű fények állapotának nyilvántartására felvesszük red, yellow és green logikai változókat. A korábban bevezetett counter változóval ellen-tétben ezek a változók nem csak az állapotgéppel leírt vezérlő belső működésében játszanak szerepet; úgy tekintjük, hogy a különböző színű fények villányégoit közvetlenül ezek az *interfészváltozók* kapcsolják. Általánosságban a be- és kimeneti események mellett interfészváltozókon keresztül kommunikálhat az állapotgép a külvilággal.

Mivel a vezérlési állapotokat úgy vettük fel, hogy egyértelműen meghatározzák a fényeket leíró változók értékét, ezen változóknak értéket adó utasításokat entry triggerhez rendeljük – ez egy tömör jelölése annak, hogy az adott állapotba lépő összes állapotátmenet végrehajtson egy adott akciót.



4.9. ábra. Fények állapotának modellezése változókkal

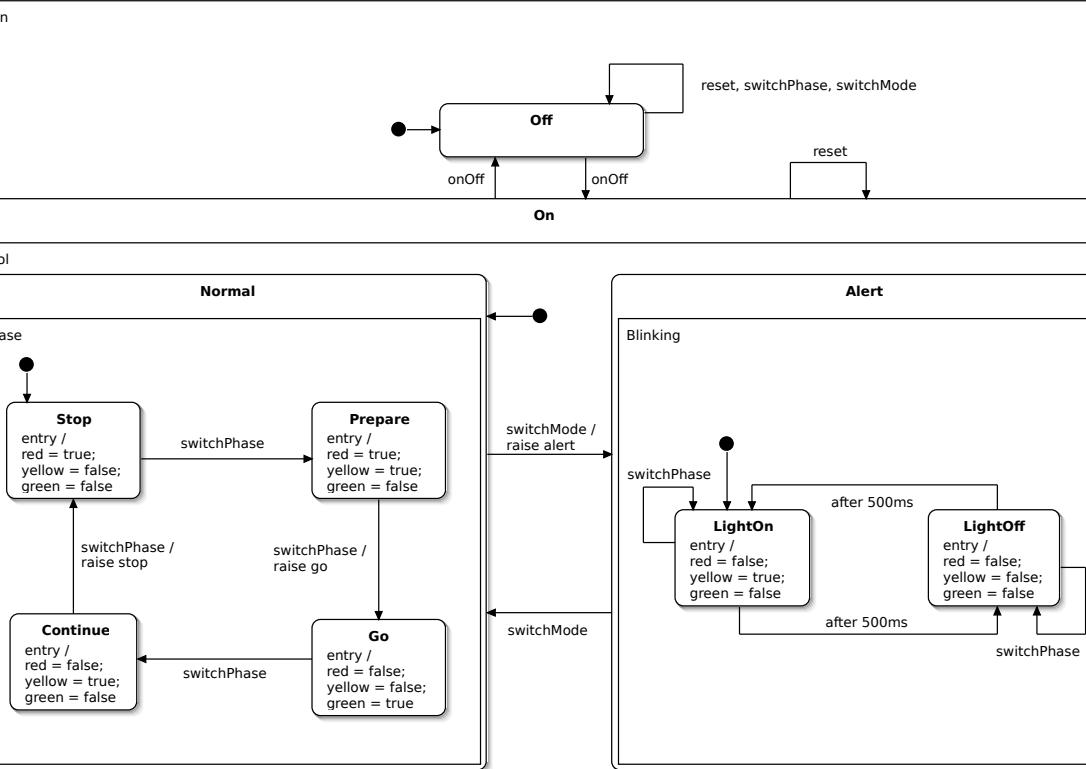
A bővített állapotgépet a 4.9. ábra szemlélteti.

4.4. Időzítés

A modell időbeli viselkedését eddig egy külső óra által szolgáltatott, megszabott frekvenciájú órajelet feltételezve modelleztük. A Yakindu azonban lehetőséget biztosít időzített események explicit modellezésére az after kulcsszóval. Ezen felül az after és every trigger használatával időzített esemény állapothoz is rendelhető.

Az időzítés explicit modellezésével a modell tömörebb, kifejezőbb lehet, és a későbbi tényleges technikai megvalósítás apró részleteitől (órajel frekvenciája) függetlenül érvényes.

Az időzítést használó modellt a 4.10. ábra szemlélteti.



4.10. ábra. Villogó jelzés időzítéssel

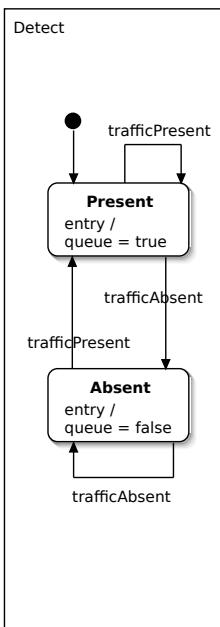
4.5. Ortogonális dekompozíció

Bővítsük a modellt járműérzékelő² funkcióval!

²https://en.wikipedia.org/wiki/Induction_loop#Vehicle_detection

Tipp. A jelzőlámpa (bekapcsolt állapotban) periodikus trafficPresent és trafficAbsent bemeneti események fogadásával értesül arról, hogy tartózkodik-e jármű a lámpa előtti útszakaszon. A jelző ezt az információt a queue logikai változóban tárolja (ennek értéke kezdetben true). A foglaltság nyilvántartásának értelme, hogy ilyenkor csak akkor kell Stop állapotból switchPhase esemény hatására jelzést váltani, ha van a lámpa előtt várakozó jármű, vagyis queue = true.

A járműérzékelő jeleinek fogadását a 4.11. ábrán szemléltetett állapotgép valósítja meg.



4.11. ábra. Járműérzékelő funkció állapotgépe

4.5.1. Állapotgépek szorzata

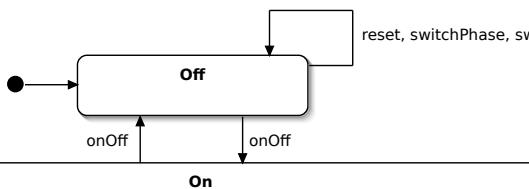
Gondoljuk végig, hogyan lehet a fenti Detect régió viselkedését kombinálni a már meglévő modellel! Mivel a foglalátságérzékelés On állapotba lépve kezd működni, így elég a meglévő modell Control régióra koncentrálni. Ekkor az eredeti modell pillanatnyi állapota Stop. Ahhoz, hogy a bővített modellben ilyenkor mind a switchPhase, mind a trafficAbsent és trafficPresent eseményeket megfelelően kezelni tudjuk, szükséges egy Stop_Present kezdőállapot és egy Stop_Absent állapot felvétele. Ugyanebből az okból kifolyólag szükséges a Prepare, Go, Continue, LightOn és LightOff állapotok megkettőzése, valamint a származtatott állapotok között a két működésnek megfelelő tranzíciók felvétele.

A fent vázolt művelet az állapotgépeken értelmezett *aszinkron szorzás*, melynek eredményét *szorzatautomatának* is nevezik. Jól látható, hogy a szorzatautomata vonatkozó régiójában az állapotok száma a két összeszorozott régió (egyszerű) állapotai számának szorzata (innen a név). Könnyen végiggondolható, hogy ha öt állapotgép együttes működését vizsgálnánk, amelyeknek külön-külön négy állapota van, akkor $4^5 = 1024$ állapota lenne a szorzatautomatának. Ebből kifolyólag ez a megközelítés egymástól nagyban független viselkedések modellezésére nem szerencsés, hiszen kezelhetetlenül nagy méretű modellekhez vezet (ún. *állapottér-robbanás* jelensége).

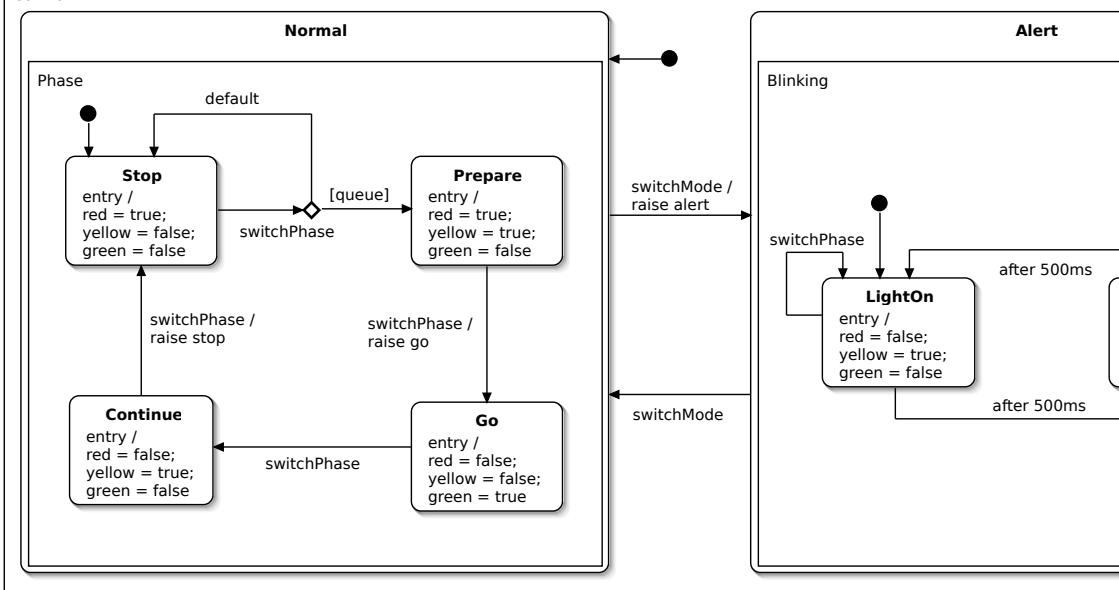
4.5.2. Ortogonális állapot

Ilyen esetekben alkalmazható eszköz az *ortogonális állapot*. Az ortogonális állapot egy olyan összetett állapot, mely több régióval rendelkezik. Az ortogonális állapot régiói *ortogonális régiók*, melyek – az egyrégiós összetett állapottal megegyező módon – akkor aktívak, ha a tartalmazó állapot aktív.

Az így elkészített állapotgépet a 4.12. ábra szemlélteti.



Control

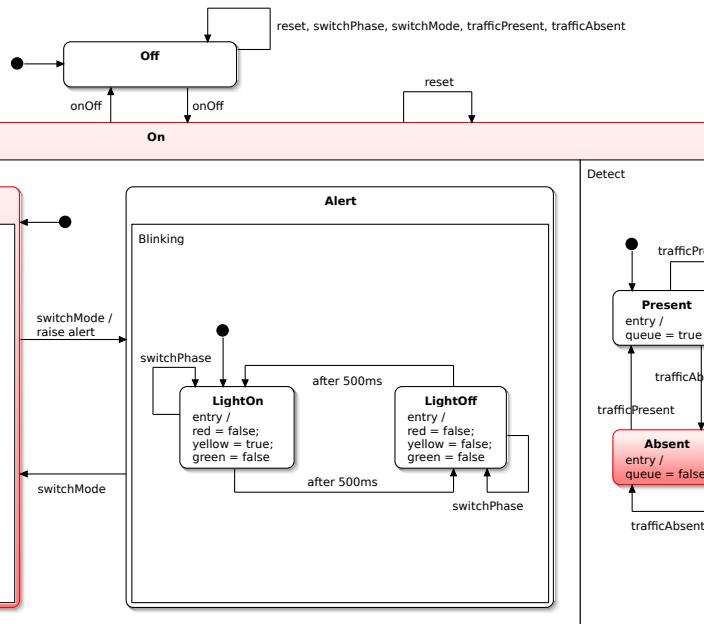


4.12. ábra. Járműérzékelős jelzőlámpa állapotgépe ortogonális

Szemantikailag az ortogonális régiók *aszinkron* módon működnek, a tranzíciók külön-külön tüzelnek, szemben a *szinkron* működéssel, amikor egy adott esemény bekövetkeztekor az ortogonális régiók tranzíciói egyszerre tüzelnek. Fontos, hogy az ortogonális régiók különböző eseményeket dolgozzanak fel, különben *versenyhelyzet* alakulhat ki. A működés összhangolása történhet például megosztott változókon keresztül; egyéb módszerek is léteznek (pl. belső események mentén történő szinkronizálás), amelyekkel itt nem foglalkozunk.

Példa. A fenti rendszert szimulálva, majd az (onOff, switchPhase, trafficAbsent) eseményeket sorrendben kiváltva a rendszer a 4.13. ábra látható állapotkonfigurációba kerül. Vegyük észre, hogy az aktív ortogonális állapot (On) minden ortogonális régiójában (Control, Detect) pontosan egy közvetlenül tartalmazott állapot aktív.

on



4.13. ábra. Ortogonális állapot aktuális állapotként

Ortogonális állapotok bevezetésével az állapotkonfiguráció fogalma is összetettebbé válik. Ilyen esetben ugyanis minden időpillanatban, amikor egy ortogonális állapot aktív, minden régiójának pontosan egy állapota aktív (az egy ortogonális állapothoz tartozó régiók között tehát nem érvényesül a kizárolgosság).

Példa. A rendszer állapotkonfigurációi:

- {Off}
- {On, Normal, Stop, Present}
- {On, Normal, Prepare, Present}
- {On, Normal, Go, Present}
- {On, Normal, Continue, Present}
- {On, Normal, Stop, Absent}
- {On, Normal, Prepare, Absent}
- {On, Normal, Go, Absent}
- {On, Normal, Continue, Absent}
- {On, Alert, LightOn, Present}
- {On, Alert, LightOff, Present}
- {On, Alert, LightOn, Absent}
- {On, Alert, LightOff, Absent}

Példa. A fentieknek megfelelően az állapotkonfigurációk halmaza tekinthető a változókat elabsztraháló állapottérnek. Ha azonban a változókon kívül a járműérzékelő működését is elabsztraháljuk, akkor arra jutunk, hogy a már ismerős

{Off, Stop, Prepare, Go, Continue, LightOn, LightOff}

halmaz továbbra is egy érvényes állapottere a rendszernek. Természetesen az is egy érvényes absztrakció, ha a Control régió állapotait absztraháljuk el; ilyenkor az

{Off, Present, Absent}

halmaz adódik állapottérnek.

Feladat. Gondoljuk végig, hogy a szorzatautomata, ill. szorzat-állapottér fogalmaknak mi köze van a matematikából ismert Descartes-szorzat művelethez!

Feladat. Miért nem a Stop állapot alállapotaiként vettük fel a járműérzékelő funkciionalitást?

4.6. A végső modell

Utolsó lépésként a különböző eseményeket és változókat *interfészekbe* (*interface*) rendezzük. Ennek a csoportosításnak az az elsődleges szerepe, hogy elkülönítsük

egymástól az eltérő külső rendszerekhez kapcsolódó elemeket; így pl. a forgalomdetektor megvalósításakor kizárálag a `Detector` interfész kell figyelembe venni, a többi interfész részleteivel nem kell megismerkedni, azok esetleges áttervezését nem kell nyomon követni. Speciális esetként megjelenik a kizárálag belső használatú `queue` változó, amely egyik külső interfésznek sem része, így külső rendszerekkel nem lesz közvetlen érintkezésben. (Yakinduban az ilyen változókat és eseményeket a speciális `internal` interfész tartalmazza.)

Az interfészdefiníciók Yakinduban a következőképpen alakulnak:

```
interface Controller:  
    in event onOff  
    in event reset  
    in event switchPhase  
    in event switchMode  
  
interface Detector:  
    in event trafficPresent  
    in event trafficAbsent  
  
interface Recorder:  
    out event stop  
    out event go  
    out event alert  
  
interface Light:  
    var red : boolean  
    var yellow : boolean  
    var green : boolean  
  
internal:  
    var queue : boolean
```

Mivel a különböző interféseknek lehetne azonos nevű eseménye ill. állapotváltozója, ezért Yakinduban minden interfész megadásával kell hivatkozni a változóra és eseményekre. A módosított modellt szemlélteti a 4.14. ábra.

4.7. Kooperáló állapotgépek szinkronizációja*

Definíció. Állapotgépek *szinkronizációján* (más néven randevú, esetenként handshake, vagy programnyelvek esetén barrier) azt értjük, hogy két kooperáló állapotgépen bizonyos állapotátmenetek csak egyszerre történhetnek meg. A szinkronizálálandó tranzíciókat szinkronizációs címkével jelöljük meg. Jelölése az állapotátmeneten: `trigger <szinkronizáció> [őrfeltétel] / akció`.

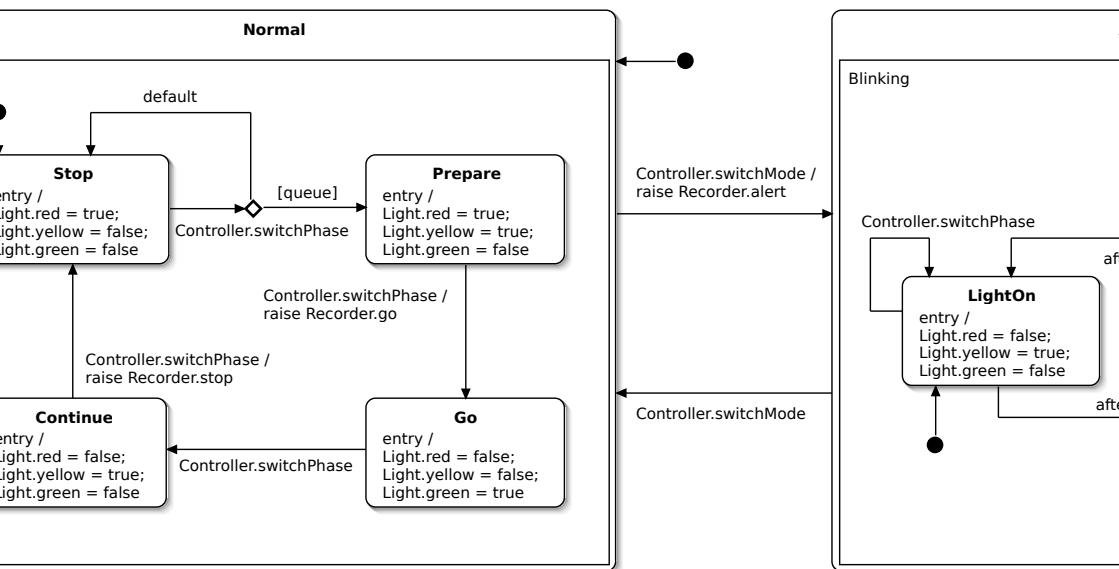
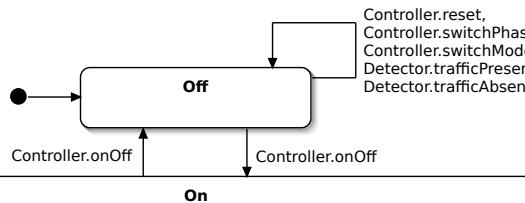
A szinkronizációval leírható, hogy a két állapotgépben megjelölt állapotátmenetek valójában a teljes rendszer egyetlen (összetett) állapotátmenetének vetületei, külön-külön nem, csak egyszerre végrehajthatók. A címke az összetett állapotátmenetre utal, így több helyen is előfordulhat ugyanaz a szinkronizációs címke, valamint többféle címke is használható.

Megjegyzés. A szinkronizáció két állapotgép között működik, vagyis szintaktikailag is csak akkor helyes, ha legalább két állapotgépet tekintünk. Ha a kooperáló állapotgépek közül csak egyet vizsgálunk, a szinkronizált átmenetek spontán átmenetté válnak. Mivel ilyenkor elveszítjük azt az információt, hogy az adott átmenet mikor hajtható végre, absztrakció történik. Ugyanakkor itt is megfigyelhető, hogy finomítással (ebben az esetben az állapotgép-hálózat többi tagjának ábrázolásával) feloldható a nemdeterminizmus, jelen esetben a spontán átmenet(ek) megszüntetésével.

Példa. A kooperáló állapotgépek közötti szinkronizáció (és egyéb interakciók) illusztrációjáért lásd a Folyamatmodellezés gyakorlati feladatsor 1/e) feladatainak megoldását.

Felhasznált irodalom

- Az állapottérkép modellezési módszer kidolgozása [?, ?]
- Az állapottérkép modell UML-ben [?]
- Állapottérkép modellek értelmezése (modellszemantika) [?, ?, ?]
- Állapottérkép alapú forráskód generálás [?]
- Pintér Gergely, *Model Based Program Synthesis and Runtime Error Detection for Dependable Embedded Systems* [?]
- UML állapottérképek használata biztonságkritikus rendszerekben [?, ?]
- Pap Zsigmond, *Biztonságossági kritériumok ellenőrzése UML környezetben* [?]



4.14. ábra. A jelzőlámpa modellje interfészszekkel

5. fejezet

Folyamatmodellezés

Az informatikában és azon kívül is számos esetben találkozunk olyan viselkedéssel, amikor megadott tevékenységek megadott sorrendben zajlanak. Például egy autómegosztó szolgáltatásban egy fuvar rendeléséhez, banki ügyintézés esetén a hitelbírálathoz, egyetemi környezetben egy tárgy felvételéhez egy jól meghatározott folyamat tartozik. Amennyiben ezeket digitálisan szeretnénk végezni, fontos, hogy a folyamatokat precízen definiáljuk.

A folyamatmodelleket széleskörűen használják, többek között az informatikában rendszerek működtetésére, protokollok specifikációjára, adatelemzési folyamatok specifikálására. Láttni fogjuk, hogy a szoftverek programkódjának elemzése is folyamatmodellre vezet.

5.1. Folyamatok

A *viselkedési modellek* a rendszer viselkedését többféle aspektusból jellemezhetik:

- Az *állapot alapú modellek* esetén a rendszereket az *állapotukkal* jellemezzük. Az állapotgép alapú viselkedésmodell arra válaszol, hogy „miként változhat” a rendszer. Másként fogalmazva: a modell elsődlegesen arra összpontosít, hogy milyen állapotokban lehetséges fel a rendszer (és nevekkel látja el az állapotokat), ill. milyen hatásokra mely állapotból mely állapotokba léphet át. Az másodlagosnak tekinthető, hogy ez a változás részletesebben megvizsgálva hogyan zajlik le, ezért a modell azt az egyszerűsítést alkalmazza, hogy az állapotátmenetek pillanatszerű események. Ilyen állapot alapú modellekkel bővebben a 4. fejezetben foglalkoztunk.
- Ezzel szemben a *folyamatmodellek* fókusza az, hogy „mit csinál” egy rendszer. A tevékenységeknek időbeli kiterjedést tulajdonítunk (ahelyett, hogy

pillanatszerűvé egyszerűsítenénk őket), és azt vizsgáljuk, hogy mely tevékenységek végezhetőek el más tevékenységek előtt vagy után, esetleg velük átla-polódva. Ugyan a rendszer állapotainak jellemzése (adott időpontban mely tevékenységek vannak folyamatban, fejeződtek már be stb.) implicit módon kikövetkeztethető a folyamatmodellből, de ez mintegy másodlagos; a modell a folyamatot alkotó tevékenységeknek ad nevet, és ezek viszonyának megadását várjuk el a modellezőtől.

A folyamatmodellezés célja tehát, hogy a rendszer folyamatát leírja.

Definíció. A *folyamat* tevékenységek összessége, melyek adott rendben történő végrehajtása valamelyen célra vezet.

5.2. A folyamatmodellek építőkövei

Az alábbiakban bemutatjuk a folyamatmodellek építőköveinek nevét, grafikus jellelését és szemantikáját.

5.2.1. Elemi tevékenység

Mielőtt valódi folyamatmodelleket vizsgálnánk, először meg kell ismerkednünk azzal az esettel, amikor valamelyen viselkedés részleteit *nem* modellezük folyamatként.

Példa. Szoftverünk C nyelvű forráskódjából futtatható programot szeretnénk előállítani; ennek egyik lépéseként egy konkrét forrásállományt le kell fordítanunk (*compile*) a fordítóprogram segítségével. Mivel a fordítóprogramot nem mi készítjük, ezért nem szükséges részleteiben vizsgálnunk, hogy milyen lépésekből áll a futása. Így tehát azt mondhatjuk, hogy a fordítóprogram futása egy *elemi tevékenység*; valamikor el fog kezdődni, utána némi idővel be fog fejeződni, és nem részletezzük, hogy közben mi történik. Ebben a jegyzetben az 5.1. ábrán láthatóhoz hasonló rajzjelekkel fogjuk a hasonló elemi tevékenységeket jelölni.

Definíció. Az *elemi tevékenység* olyan időbeli kiterjedéssel rendelkező tevékenység, amelynek a megkezdésén és befejezésén túl további részleteit nem modellezük.

Compile

5.1. ábra. Elemi tevékenység grafikus szintaxisa

Példa. Mit is értünk azalatt, hogy a fordítóprogram futtatása időbeli kiterjedéssel bír? Ahogy az 5.2. ábrán látható idődiagram is illusztrálja, kezdetben a tevékenység nem fut. Valamikor a működés során eljön a tevékenység kezdete - ezt egy pillanatszerű eseményként modellezzük; utána úgy tekinthető, hogy a fordítás tevékenység *folyamatban van*. Később eljön az az idő, amikor a fordítás befejeződik; ez egy újabb pillanatszerű esemény, amely után az elemi tevékenység már nincs folyamatban, befejezettnék tekinthető.



5.2. ábra. Elemi tevékenység időbeli lefutása idődiagramon

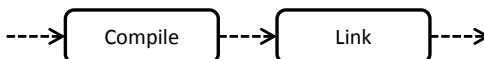
Ahogy a fenti példa is illusztrálja, minden elemi tevékenység önmagában egy hárromelemű állapotteret határoz meg: {még nem kezdődött el, folyamatban van, már befejeződött}; később az összetett folyamatmodellek állapotteréről is lesz szó. Láttható, hogy az elemi tevékenység is leírható a korábban tanult állapotmodellezési eszköztárral; ebben az esetben viszont más a modell fókusza, más elemeket tartunk elnevezésre és vizsgálatra érdemesnek.

Megjegyzés. Bizonyos források *atomic* tevékenység vagy lépés néven hivatkoznak ugyanerre az *elemi tevékenység* fogalomra, de ebben a jegyzetben ezt kerüljük. Ellenkező esetben összekeverhető lenne egy hasonló nevű másik fogalommal: az *atomic művelet* (*atomic operation*) kifejezetten a pillanatszerűnek tekinthető, időbeli kiterjedés nélkül modellezett tevékenységekre utal. Az atomokhoz hasonlóan az atomi művelet nem osztható: vagy el se kezdődött, vagy már befejeződött, de nem találhatjuk magunkat olyan időpillanatban, amikor részben már lezajlott, de még folyamatban van. Ezzel szemben az elemi tevékenység időbeli kiterjedéssel bír, és a modell megenged olyan időpontot, amikor épp folyamatban van; még ha nem is részletezi, a tevékenység mely elemei milyen készültségi fokon vannak. A tevékenységek kezdetét és befejezését viszont már atominak, pillanatszerűnek tekintjük.

5.2.2. Szekvencia

Ha a modelljeink csak egymástól izolált elemi tevékenységeket tartalmaznának, nem sok hasznos tudást fejeznének ki. A folyamatmodellek igazi erőssége, hogy a tevékenységekből *folyamatot* építenek fel, amely azt fejezi ki, hogy az egyes tevékenységek egymáshoz viszonyítva mikor hajthatóak végre. A legegyszerűbb ilyen konstrukció a *szekvencia*, ahol a tevékenységeket úgynevezett *vezérlési éllek* (*vezérlésifolyam-élék*) kötik össze.

Példa. Az ipari gyakorlatban egy C program forráskódja tipikusan nem csak egyetlen fájlból áll. Miután egy C forrásfájlt tárgykóddá fordítunk, utána össze kell *linkelni* más tárgykódokkal (korábban lefordított forrásállományok, függvénykönyvtárak), hogy végül megkapjuk a futtatható állományt. Így tehát a következő folyamatot kell elvégezni: először a fordítás elemi tevékenységet kell végrehajtani, majd annak befejezése után kezdhető meg a linkelés. Az 5.3. ábrán ennek a *szekvenciának* a jelölését látjuk; a szaggatott nyíl rákövetkezést jelöl, tehát hogy a *Compile* tevékenység vége után kezdhető meg a *Link*.



5.3. ábra. Szekvencia grafikus szyntaxa

Definíció. A *szekvencia* tevékenységek szigorú végrehajtási sorrendjét definiálja.

Megjegyzés. Ahogy az ábrán is látható, a vezérlési éleket jelen jegyzetben egységesen szaggatott nyíllal jelöljük, de más forrásokban, különböző modellezési nyelvek szabványaiban gyakran jelölik folytonos nyíllal.

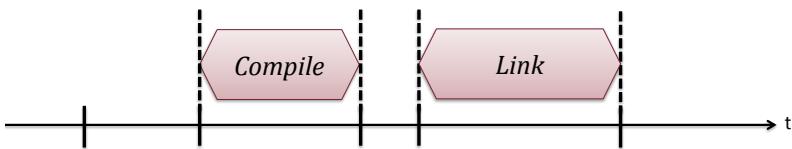
A következőkben több külön módszerrel értelmezzük a *szekvencia* szemantikáját. Bár ez a vizsgálat feleslegesen alaposnak és szájbarágónak tűnhet („ágyúval verébre”), de a később előkerülő összetettebb folyamatmodell-konstrukciók megértését nagymértékben segíti.

Példa. Hogyan szimulálhatjuk a szekvenciánk működését? Ha az 5.3. ábrán látható folyamatdiagramot kinyomtatjuk, és a papírra helyezünk egy régi egy-forintost (vagy csavaranyát, vagy bármely egyéb jelölőt, amelyet a továbbiakban a *token* névvel illetjük), akkor az ábra alapján könnyen követhetjük a folyamat működését.

- Helyezzük kezdetben a tokent az ábra bal szélén belépő szaggatott nyílra! Mivel a token nem tevékenységen áll, ezért ezt úgy értelmezzük, hogy nem fut jelenleg egyik feltüntetett elemi tevékenység se.
- Csúsztassuk ujjunkkal kissé arrébb a tokent. Kövessük nyilat, tehát kerüljön a token a *Compile* tevékenységre! Amíg a tevékenység rajzjelén áll a token, úgy tekintjük, hogy a tevékenység folyamatban van.
- Amikor a nyilak irányában ismét továbbmozgatjuk, a token a két tevékenység közötti szaggatott nyílszakaszra kerül. Ekkor az első tevékenység már nem fut, tehát befejeződött; ugyanakkor a második tevékenység még nem kezdődött el.
- Harmadszor is mozgatva a tokent, elkezdhetjük a *Link* tevékenységet.
- Végül, az ábra jobb szélén látható nyílra helyezve a tokent, kifejezzük a második tevékenység befejeződését is.

Ha belegondolunk, a tokennel valójában a következő állapotteret jártuk be: { még nem kezdődött el egyik tevékenység sem, *Compile* folyamatban van, *Compile* befejeződött és *Link* még nem kezdődött el, *Link* folyamatban van, befejeződött minden tevékenység }. A folyamatmodell határozza meg ezt az öt állapotot, valamint hogy milyen állapotátmenetek megengedettek közöttük (jelen esetben csak a felsorolás sorrendjében lehet állapotot váltani). A folyamat tehát ezt az állapotmodellt indukálja.

Ha a folyamatot (például a fenti példához hasonlóan token mozgatásával) szimuláljuk, egy konkrét lefutását kapjuk. A folyamat konkrét lefutásait a folyamatmodell *folyamatpéldányainak* nevezzük. Ez nyilván akkor lesz izgalmas fogalom, ha egy folyamatot nem csak egyszer hajtunk végre, hanem többször. Ha belegondolunk, a tokennel valójában a végrehajtás pillanatnyi állapotát jelöltük ki, és a következő állapotteret jártuk be vele: { még nem kezdődött el egyik tevékenység sem, *Compile* folyamatban van, *Compile* befejeződött és *Link* még nem kezdődött el, *Link* folyamatban van, befejeződött minden tevékenység }. A folyamatmodell határozza meg ezt az öt állapotot, valamint hogy milyen állapotátmenetek megengedettek közöttük (jelen esetben csak a felsorolás sorrendjében lehet állapotot váltani). A folyamatmodell tehát ezt az állapotmodellt indukálja, a folyamat szimulációját pedig visszavezetjük a fent konstruált állapotmodell szimulációra. A szimuláció eredményeképpen az 5.4. ábrán látható idődiagramhoz hasonló eredményt kapunk.



5.4. ábra. Szekvencia időbeli lefutása

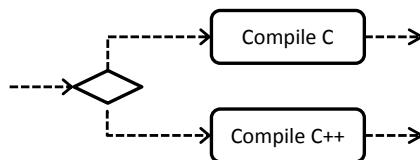
5.2.3. Elágazás, örfeltétel

Azt is ki lehet fejezni megfelelő folyamatmodellel, ha nem minden lefutás ugyanazokat a tevékenységeket hajtja végre. Ilyen elágazások modellezésére szolgál a *döntési csomópont*, amely az első általunk megismert *vezérlési elem* (*vezérlési csomópont / vezérlésifolyam-csomópont*) a folyamatmodellben.

Definíció. A *vezérlési elem* (*vezérlési csomópont / vezérlésifolyam-csomópont*) olyan csomópont a folyamatban, mely a folyamatmodell tevékenységei közül választ ki egyet vagy többet végrehajtásra.

Példa. Bizonyos fájlokra a C fordítót, másokra a C++ fordítót kell meghívni. Ezt az 5.5. ábrán látható folyamatmodell fejezi ki, ahol a rombusz jelképezi a *döntési csomópontot* (*elágazást*). Ennek a folyamatmodellnek az is érvényes lefutása, ha az egyik fordítót hívjuk meg, és az is, ha a másikat.

A folyamat szimulációja során először a bal oldalról belépő vezérlési ére helyezzük a tokent; ezek után szabadon választhatunk, hogy a döntési csomópontból kilépő élek közül melyikre rakjuk át. A folytatás már a jól ismert módon történik: a tokent először a választott elemi tevékenységre, majd a továbblépő vezérlési ére helyezzük.



5.5. ábra. Elágazás grafikus szintaxisa

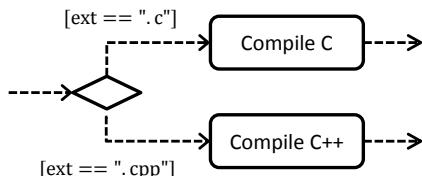
Természetesen 2 helyett 3, 4 stb. ágú döntési csomópont is elképzelhető.

Az elágazási pontnál bármelyik ágat is választjuk, a folyamatmodell egy érvényes lefutását kapjuk. Másképpen szólva: a modell nem fejezi ki azt az információt, hogy mi alapján dönthető el, melyik esetben melyik lehetőség fog megtörténni. A korábban tanult szóhasználattal élve *nem determinizmust* mutat a modell. Gyakran hasznos így modellezni, pl. ha emberi döntéstől függ a választás; vagy bármilyen

egyéb esetben, ha vagy nincs rálátásunk a döntést meghatározó tényezőkre, vagy a modellezés jelenlegi absztrakciós szintjén a szükséges részleteket el akarjuk hanyagolni.

Gyakran esetekben azonban a rendszermodellünkben elérhető olyan információ, amely meghatározza, hogy a folyamat végrehajtása melyik ágon történhet, determinisztikussá téve a választást. Más modellekknél erről nincs szó, de a rendelkezésre álló információ alapján legalább időnként csökkenthető a választási lehetőségek száma. Mindkét esetben a szoros értelemben vett folyamatmodellen kívüli tudás alapján kizártuk a döntés után előálló ágak nemelyikét; erre pedig (az állapotgépekhez hasonló módon) az ún. *őrfeltétel* szolgál.

Példa. Elérhető az `ext` karakterlánc, amely a fordítandó forrásfájl kiterjesztését adja meg. Ez alapján minden esetben előírható, hogy melyik nyelv fordítóprogramját kell végrehajtani. Az 5.6. ábrán látható folyamatmodell őrfeltételekkel fejezi ezt ki.



5.6. ábra. Elágazás grafikus szintaxisa őrfeltételekkel

A korábban bevezetett fogalmaknak megfelelően akkor determinisztikus a folyamatmodell, ha minden egyes elágazás minden végrehajtásánál kizárták egymást az őrfeltételek; és akkor teljesen specifikált, ha minden egyes elágazás minden végrehajtásánál az őrfeltételek legalább egyike mindig teljesül. Ha nincsenek őrfeltételek megadva, az úgy tekintendő, mintha az állandó „[igaz]” őrfeltételt alkalmaztuk volna.

Összefoglalásként tehát ezt a definíciót alkothatjuk:

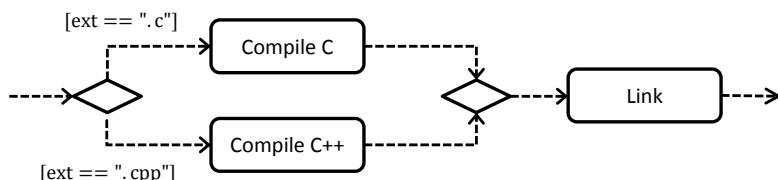
Definíció. A *döntési csomópont* olyan csomópont a folyamatban, amely a belé érkező egyetlen vezérlési él hatására a belőle kiinduló ágak (vezérlési élek) közül pontosan egyet választ ki végrehajtásra, az esetleges őrfeltételekkel összhangban.

5.2.4. Merge (besorolódás)

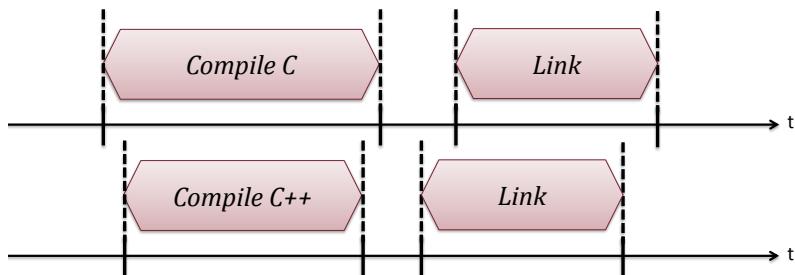
Gyakran egy elágazás különböző ágai egy adott ponton túl egyformán folytatódnak. Ennek kifejezésére szolgál egy újabb vezérlési elem, a *merge (besorolódási)*

csomópont. Sajnos nem terjedt el rá frappáns magyar név, így leggyakrabban az angol merge szót használjuk.

Példa. Akár C, akár C++ nyelvű forrásfájt fordítottunk le, utána mindenkor a linkelés következik. Az 5.7. ábrán feltüntetett merge csomópont (rajzjele egyezik a döntési csomópontéval) éppen ezt fejezi ki. A modell szimulációja kor, miután az egyik fordítási tevékenységet végrehajtottuk, a token a merge csomópontba mutató nyilak egyikén áll; ezután egyszerűen átmozgathatjuk a merge csomópont túloldalára, hogy utána már a *Link* tevékenység végrehajtása következhessen. Akármelyik ágról is érkezik a token, a merge csomópont után ugyanazon kimenő vezérlési élre kerül; ez ahhoz hasonlatos, mint amikor a közúti forgalomban egy közlekedési sáv megszűnésekor két járműoszlop ból is ugyanabba a sávba sorolódnak be a járművek. Az 5.8. ábra idődiagramon mutat be két eltérő lefutást.



5.7. ábra. Elágazás és merge (besorolódás) grafikus szintaxisa



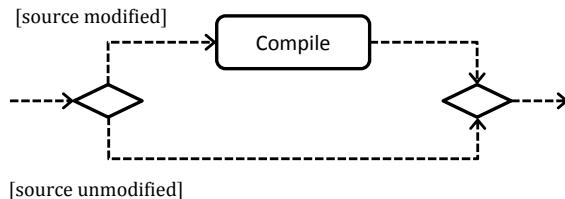
5.8. ábra. Elágazás és merge (besorolódás) kétféle lefutása idődiagramon

Definíció. A *merge (besorolódási) csomópont* olyan csomópont a folyamatban, amely a belé érkező ágak közül akármelyik végrehajtásakor kiválasztja a belőle kiinduló egyetlen vezérlési élet további végrehajtásra.

Természetesen a többágú döntési csomópont mintájára többágú merge is elképzelhető. További különleges eset, ha a döntési és merge csomópontok között valamelyik ágon csak egy üres vezérlési él vezet, semmilyen tevékenység nem hajtódi vég;

ilyet akkor használunk, ha egy tevékenység végrehajtása opcionális, vagy egy döntés valamelyen kimenetele esetén nincs teendő.

Példa. Valójában csak olyankor kell lefordítani egy forrásállományt, ha a fordítóprogram utolsó végrehajtása óta módosult; egyéb esetben a régi tárgykódot meghagyva a fordítási lépés kihagyható. Ilyen elven épített folyamatot mutat be az 5.9. ábra.

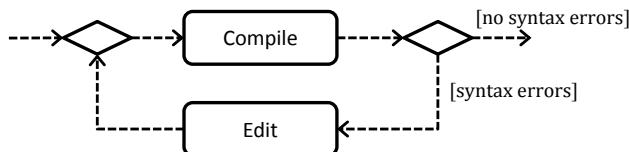


5.9. ábra. Elágazás üres ággal

5.2.5. Ciklus

Ha már megismertük a döntés és merge vezérlési elemeket, akkor építhetünk velük ciklust, amely a folyamat egy részletét többször is képes ismételni.

Példa. Az 5.10. ábra olyan folyamatot mutat be, ahol a fordítás után megvizsgáljuk, találtunk-e fordítási hibát; amennyiben van hiba, akkor azt megpróbáljuk kijavítani, és újrafordítjuk az állományt. Előfordul, hogy továbbra is vannak hibák, ekkor ismét a kód szerkesztése és újrafordítás következik. Ezek a tevékenységek mindaddig ismétlődnek, amíg végül el nem tűnnek a hibák.



5.10. ábra. Ciklus grafikus szintaxisa

Definíció. A *ciklus* olyan folyamatmodell (részlet), amelyben egy elágazás valamelyik ágán az elágazást *megelőző* merge csomópontba jutunk vissza.

Nem nehéz végiggondolni, hogy ciklikus folyamatok futásakor ezek a vezérlési csomópontok többször is érinthetőek. Próbáljuk ezt a fenti példán a tokenes kézi szimuláció módszerével kipróbálni!

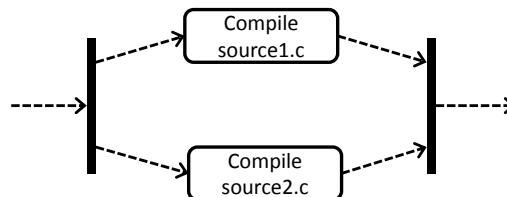
5.2.6. Konkurens viselkedés

Előfordulhat olyan folyamat, amelyben nincs előírva két tevékenység (vagy részfolyamat) egymáshoz képesti sorrendje, csak hogy mindenki meg kell történnie. A szóban forgó két tevékenység közül történhet az egyik a másik előtt, vagy fordítva; sőt, futhatnak (részben) egyszerre is. Ilyen viselkedést fejez ki a *fork csomópont* és a *join (találkozási / szinkronizáló) csomópont* párosa (itt ismét nincsenek általánosan elfogadott magyar fordítások).

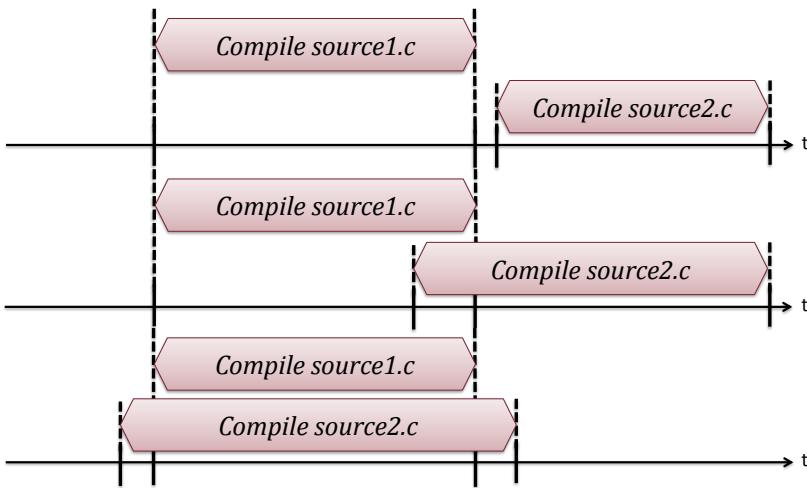
Definíció. Két bekövetkező tevékenység vagy esemény *konkurens*, ha a bekövetkezési sorrendjükre nézve nincs megkötés.

Példa. Az 5.11. ábra olyan folyamatot mutat be, ahol két forrásfájt is lefordítunk, azonban a sorrendjük nincs meghatározva. Fordítható az 1-es számú állomány a 2-es előtt, vagy fordított sorrendben. Ha többmagos processzorunk van, érdemes lehet a két fájl fordításával egyszerre is megpróbálkozni.

Ezt a folyamatmodellt a következőképpen szimulálhatjuk: először természetesen a bal szélről belépő vezérlési élen van a token. A következő lépében a fork csomópont hatására *megkettőzzük* a tokent: az egyik token a felső, a másik az alsó kimenő vezérlési ére kerül. A továbbiakban szabadon választhatóan akár melyik token léptethető. Például elkezdheti először a felső token a tevékenységet, majd befejezheti, mielőtt az alsó elkezdené és befejezné a saját tevékenységét. Egy másik lehetőség, hogy a felső token elkezdi a felső tevékenységet, majd az alsó token az alsó tevékenységet, így átlapolva a két tevékenység végrehajtását; ezek után akár milyen sorrendben befejezhetik a saját tevékenységeiket. Számos lehetőség van; ám végül így vagy úgy, de eljutunk abba a helyzetbe, amikor minden token a join csomópont egy-egy bemenő vezérlési élén van. Ekkor egyetlen lépésként a két tokent újra összeolvastjuk, és egyetlen token helyezünk a join kimenő élére. Néhány lehetséges lefutás látható az 5.12. ábrán.



5.11. ábra. Fork és join grafikus szintaxisa



5.12. ábra. Fork és join néhány lehetséges lefutása idődiagramon

Nagyon fontos megérteni, hogy a két *párhuzamos folyam* megvárja egymást (szinkronizál) a join csomópontnál; egyik se haladhat tovább, amíg a join csomópont összes bemenő vezérlési élre nem érkezik token.

Természetesen a többágú döntési és merge csomópont mintájára többágú fork ill. join is használható. Ilyenkor a fork hatására megkettőzés helyett többszöröződik a token, ill. több token várja össze egymást a join csomópontnál.

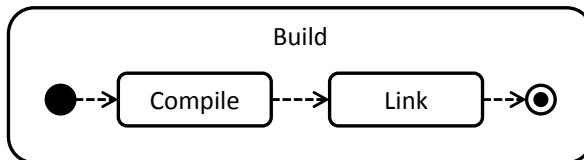
Definíció. A *fork csomópont* olyan csomópont a folyamatban, amely a belé érkező egyetlen vezérlési él hatására a belőle kiinduló összes *párhuzamnos folyamot* (vezérlési élet) kiválasztja végrehajtásra.

Definíció. A *join (találkozási / szinkronizáló) csomópont* olyan csomópont a folyamatban, amely a belé érkező összes *párhuzamos folyam* végrehajtása után kiválasztja a belőle kiinduló egyetlen vezérlési élet további végrehajtásra.

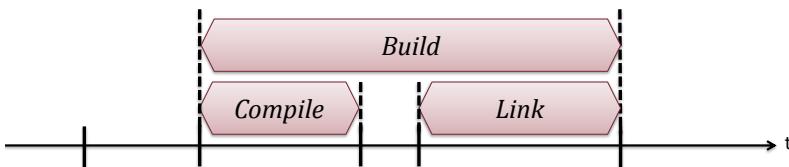
5.2.7. Teljes folyamatok

Eddig csak a folyamatok építőelemeivel foglalkoztunk; most megnézzük, hogy lehet a segítséggel egy egész folyamatot leírni az elejtől a végéig. Ehhez csupán két új vezérlési csomópontra lesz szükségünk; ezek az új elemek a folyamat kezdetét jelentő, egyszerű körrel/koronggal jelölt *start (flow begin) csomópont*, valamint a folyamat befejeztét jelentő és a dupla falú körrel jelölt *cél (flow end) csomópont*.

Példa. Vegyük az az 5.13. ábrán látható, *Build* névvel illetett egyszerű folyamatot! Jól látható, hogy a teljes folyamat két elemi tevékenység szekvenciájából áll. A folyamat szimulációját úgy kezdjük meg, hogy a kezdőcsomópontban létrehozunk és az onnan kilépő vezérlési ére mozgatunk egy token; ez jelképezi a *Build* folyamat elindulását. Ezek után a szekvencia a már megismert módon szimulálható, amíg végül a *Link* tevékenységből kilépő vezérlési ére nem kerül a token. Ekkor a folyamat befejeződhet, amelyet úgy jelzünk, ha a célcsomópontba vezető élről a célcsomópontba mozgatjuk és egyúttal felszedjük a token. A lefutás idődiagramját az 5.14. ábra mutatja; jól látható, hogy az egyes tevékenységek végrehajtása teljes egészében a folyamat futása alatt zajlik.



5.13. ábra. Flow begin és flow end grafikus szintaxisa



5.14. ábra. Teljes folyamat idődiagramja

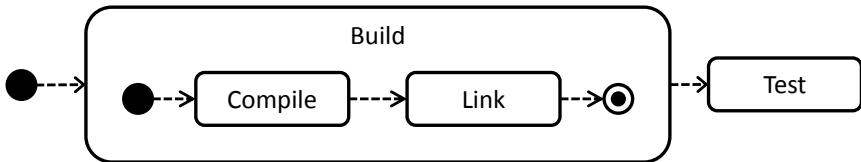
Definíció. minden folyamat egy *start (flow begin) csomópont* vezérlési elemmel indul, és egy *cél (flow end) csomópont* elemmel fejeződik be. Az *start (flow begin) csomópont* a folyamat elindulását jelentő elem, melynek pontosan egy kiemenete van. A *cél (flow end) csomópont* a folyamat befejezését jelentő elem, melynek pontosan egy bemenete van.

5.2.8. Hierarchikus folyamatmodellek

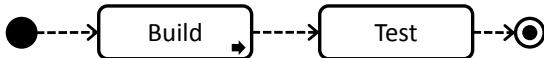
Egészen idáig elemi tevékenységeket használtunk a folyamatainkban. Azonban lehetőségünk van összetett tevékenységek modellezésére is, ahol a tevékenység belső lépései egy külön folyamatmodell írja le. Egyfelől a hierarchikus modellezés elvét követve egy alfolyamat beágyazható tevékenysékként egy főfolyamatba; másrészt elkülönítetten definiált folyamatokra is hivatkozhat egy tevékenység.

Példa.

Az 5.15. ábra az alfolyamattá részletezett tevékenység használatát mutatja be, míg az 5.16. ábra az előzővel azonos jelentésű folyamatot épít fel úgy, hogy a folyamat első tevékenysége hivatkozza (hívja) az 5.13. ábrán látható, korábban definiált folyamatot.



5.15. ábra. Hierarchia grafikus szintaxisa



5.16. ábra. Hívás grafikus szintaxisa

5.2.9. Folyamatpéldányok

Példa. Vegyük példának az 5.13. ábrán látható folyamatmodellt, és szimuláljuk. Ahogy az 5.14. ábra idődiagramja is mutatja, a szimuláció alatt sorban a következő események következtek be:

1. A *Build* folyamat elkezdődik.
2. A *Compile* tevékenység elkezdődik.
3. A *Compile* tevékenység befejeződik.
4. A *Link* tevékenység elkezdődik.
5. A *Link* tevékenység befejeződik.
6. A *Build* folyamat befejeződik.

Ha egy folyamatmodellt szimulálunk (például a korábban bemutatott manuális módszerrel, token mozgatásával), akkor a folyamat egy konkrét lefutását kapjuk. A folyamat konkrét lefutásait a folyamatmodell *folyamatpéldányainak* nevezzük. A folyamatpéldány olyan események sorozata, amelyek a folyamatot alkotó elemi tevékenységek kezdetét és befejezését jelzik, illetve az egész folyamat kezdetét és befejezését. A folyamatmodell szemantikája voltaképp az, hogy ezen eseményeket azonosítja, és lehetséges sorrendjükre tesz megkötéseket.

Definíció. Egy folyamatmodellhez tartozó *folyamatpéldány* olyan diszkrét eseménysor, amelyet a következő jellegű események alkotják, a folyamatmodell által megszabott időrendben:

- a folyamat kezdete,
- a folyamatot alkotó egyik tevékenység kezdete,
- a folyamatot alkotó egyik tevékenység vége,
- a folyamat vége.

Megjegyzés. Egy folyamatnak több folyamatpéldánya lehet; sőt, több olyan példánya is, amely ugyanolyan eseményeket tartalmaz ugyanolyan sorrendben.

A folyamatmodellek és folyamatpéldányok közti viszony nyilván akkor lesz izgalmas, ha egy folyamatot nem csak egyszer hajtunk végre, hanem többször, egymás után vagy akár részben átlapolódva. Az egyszerre végrehajtott folyamatpéldányokat úgy lehet szimulálni, hogy minden folyamatpéldányhoz egy-egy tokenet rendelünk, amelyik a példány pillanatnyi állapotát jellemzi; ezután az összes tokenet felrakjuk a folyamat diagramjára, és külön-külön léptetgetjük őket.

Megjegyzés. A több folyamatpéldány reprezentálására szolgáló tokensokaság nem keverendő össze azzal az esettel, amikor egy *fork csomópont* hatására többszörözzük egyetlen folyamatpéldány tokenjét. A megkülönböztetést az indokolja, hogy a join csomópontnál természetesen továbbra is csak egyazon folyamatpéldányhoz tartozó szétvált tokenek egyesülnek. Így biztosítható, hogy minden egyes folyamatpéldány önmagában értelmes, a folyamatmodellel konform eseménysor. Szimuláció közben célszerű úgy felfogni, hogy minden folyamatpéldánynak különböző színű tokenje van, és a fork, ill. join csomópontok csak egyszínű tokeneket vágnak szét, ill. egyesítének.

A 6. fejezetben kifejezetten azzal az esettel foglalkozunk majd, amikor ugyanaz a folyamat egyszerre nagyon sok példányban fut. Ilyen esetben a szimuláció során nem is érdemes a tokenekkel egyenként vesződni; csak azt tartjuk számon, hogy hány token tartózkodik éppen a diagram egy adott pontján.

5.3. Folyamatmodellek felhasználása

5.3.1. Programok vezérlési folyama

Alapismeretek

Bizonyára észrevetettük, hogy a legtöbb programozási nyelven (pl. C, C++ stb.) készült eljárások, függvények, metódusok, szkriptek sokban hasonlítanak a folyamatmodellekhez: egymás után elvégzendő lépések szekvenciáját határozzák meg, bizonyos esetekben elágazásokkal, más kódrészletek hívásával. Sőt, egyes programozási nyelvek nyelvi elemként tartalmaznak fork, ill. join jellegű primitíveket is.

Definíció. Egy program által meghatározott folyamatmodellt, amely az elvégzendő lépéseket, ill. a végrehajtásukra előírt sorrendet írja le, a program vezérlési folyamának (*control flow*) nevezik.

Definíció. Azon programokat, amelyek vezérlési folyamot határoznak meg, *imperatív* programnak nevezzük.

Megjegyzés. Bár a kezdő programozók általában az imperatív programokkal kezdik tanulmányiakat, léteznek ettől eltérő programozási paradigmák is, ahol a program a végrehajtási lépéssort nem határozza meg követlenül. Egyetemünkön a *Deklaratív programozás* című tárgy foglalkozik a funkcionális, ill. a logikai programozás világába való bevezetéssel.

Persze ne higgyük azt, hogy a program csak a vezérlési folyamból áll; számos egyéb részletet is meg kell határoznia (pl. adatszerkezetek, adatáramlás), amelyeket a vezérlési folyam kiemelésekor elabsztrahálunk.

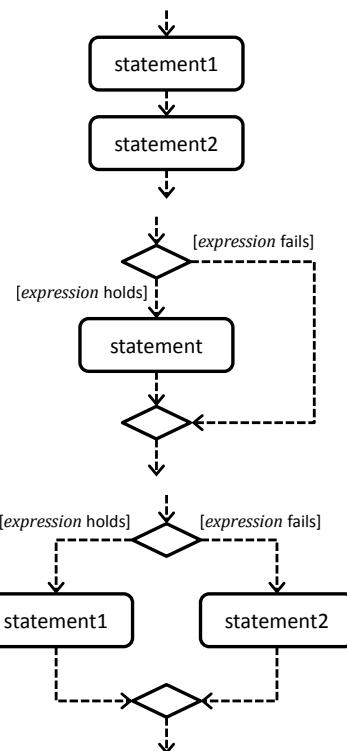
Leképzés

Az alábbiakban egy C-szerű programozási nyelvet alapul véve, számos vezérlési struktúrára megmutatjuk, hogy milyen vezérlési folyamot határoz meg.

```
<statement1>  
<statement2>
```

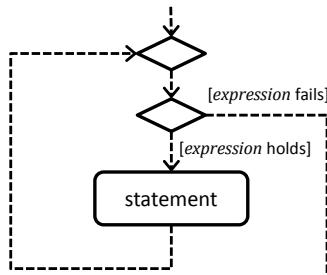
```
if (<expression>)  
  <statement>
```

```
if (<expression>)  
  <statement1>  
else  
  <statement2>
```

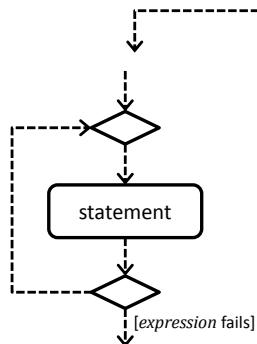


Összetett példa ábrázolása

```
while (<expression>)
    <statement>
```

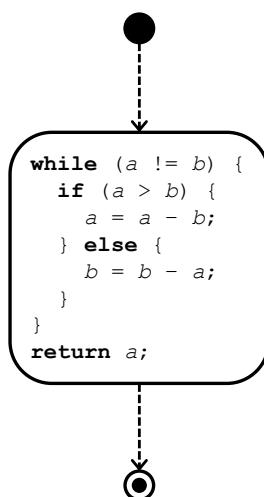


```
do
    <statement>
while (<expression>)
```

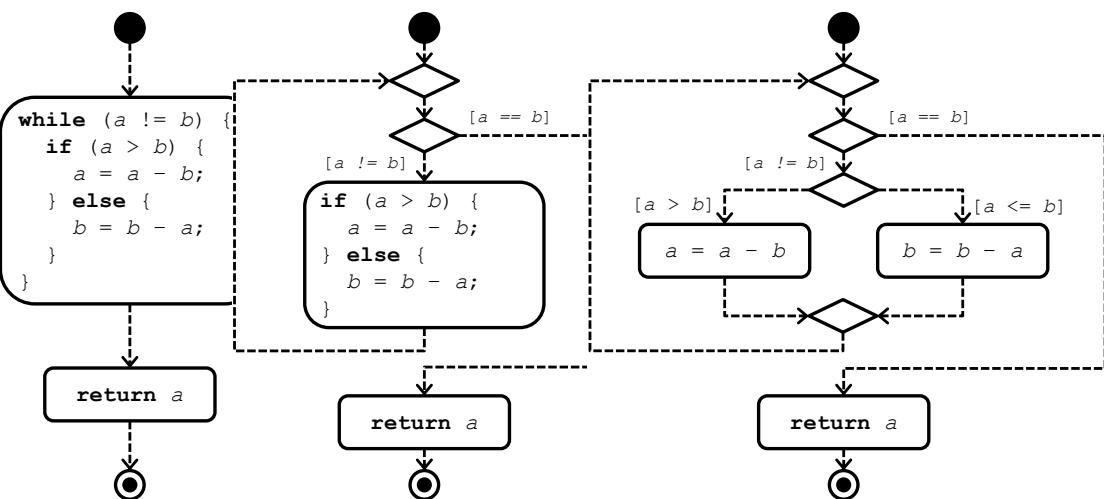


Nézzünk meg egy összetettebb példát!

```
while (a != b) {
    if (a > b) {
        a = a - b;
    } else {
        b = b - a;
    }
}
return a;
```



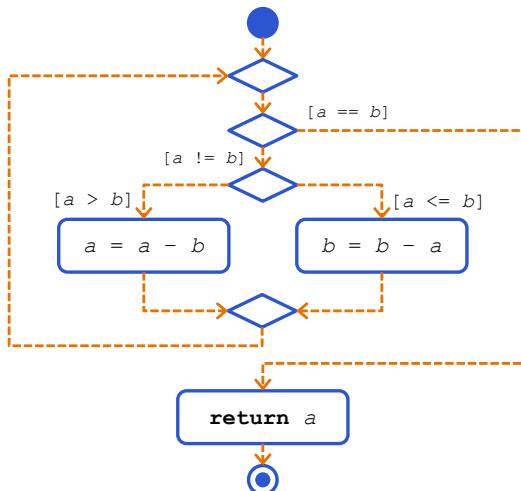
Lépésenként átalakítva:



Vezérlési folyam ciklomatikus komplexitása

A vezérlési folyamok ismeretének egyik létjogosultsága, hogy segítségükkel a programok könnyebben elemezhetőek. Bár a programok elemzése túlmutat jelen tárgy keretein, maradjon itt illusztrációnak egy egyszerű elemzési mód, amely arra próbál választ adni, hogy egy programkód mennyire szövevényes, nehezen átlátható.

Definíció. A vezérlési folyamhoz tartozó *ciklomatikus komplexitás*: $M = E - N + 2$, ahol E a vezérlési élek, N a vezérlési csomópontok és tevékenységek száma.



5.17. ábra. A ciklomatikus komplexitás fogalmai. E : élek (narancssárga), N csomópontok (kék)

Megjegyzés. A ciklomatikus komplexitással bővebben a *Szoftvertechnológia* tárgy foglalkozik. A kiszámítására használt formula pedig ismerős lehet a *Bevezetés a számításelméletbe 2.* tárgyból (v.ö. összefüggő, síkbarajzolható gráfokra vonatkozó Euler-formula).

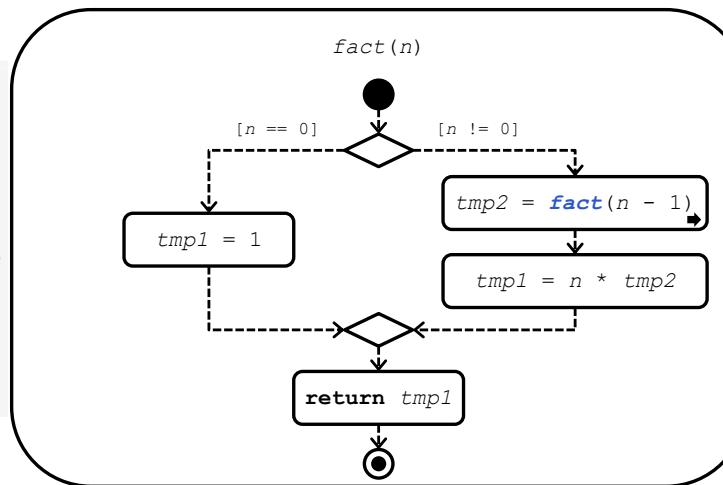
Példa: $n!$ meghatározása

Vizsgáljuk meg az alábbi programkódot, ami egy szám faktoriálisát határozza meg!

```
int fact(int n) {
    return (n == 0) ? 1 : n * fact(n - 1);
}
```

A `? :` operátor tömör kódöt eredményez, de esetünkben fontosabb szempont, hogy a kódban bejárható útvonalakat lássuk. Mentsük el továbbá a visszatérési értéket egy átmeneti változóba. Így az alábbi kódot kapjuk:

```
int fact(int n) {
    int tmp1;
    if (n == 0) {
        tmp1 = 1;
    } else {
        int tmp2 = fact(n - 1);
        tmp1 = n * tmp2;
    }
    return tmp1;
}
```



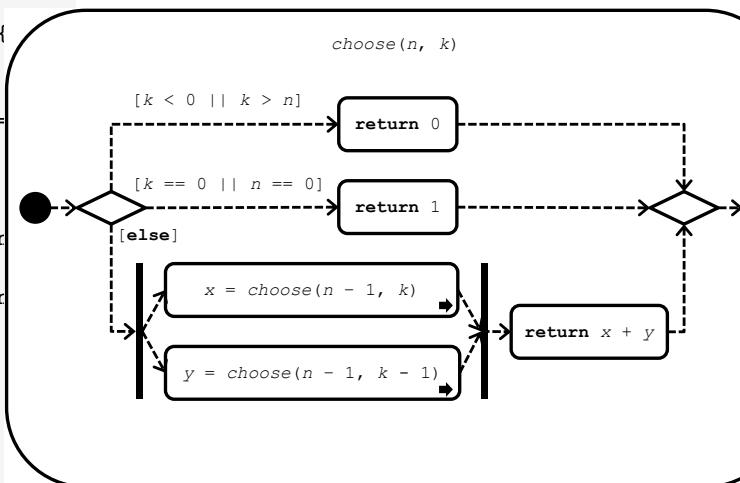
Példa: $\binom{n}{k}$ meghatározása

Az alábbi rekurzív függvény meghatározza $\binom{n}{k}$ értékét. A számításhoz felhasználjuk, hogy $\binom{0}{0} = 1$ és $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.

```

int choose(int n, int k) {
    if (k < 0 || k > n) {
        return 0;
    } else if (k == 0 && n == 0) {
        return 1;
    } else {
        int x = spawn choose(r
            k);
        int y = spawn choose(r
            k - 1);
        sync;
        return x + y;
    }
}

```



5.3.2. Jólstrukturált folyamatok

Eddig semmilyen megszorítást nem tettünk arra, hogy a vezérlési élek mely csomópontokat melyekkel köthetik össze; így pedig sok értelmetlen vagy helytelenül működő folyamatmodell építhető. Ráadásul az egyébként értelmes folyamatmodellek is gyakran átláthatatlanok, nehezen érthetőek lehetnek. Az átláthatóság egyik fő akadálya, ha egy bonyolult részfolyamatba több ponton is be lehet lépni, és több ponton is ki lehet lépni belőle.

Ezért szokás a folyamatmodelleknek az alábbiakban definiált „biztonságos” részhalmazát elkülöníteni, amely megengedett blokkokból építkezik csak.

Definíció. A következő (egy belépési és egy kilépési pontú) részfolyamatokat tekintjük jólstrukturált blokknak (más néven jólstrukturált részfolyamatnak):

- egyetlen elemi tevékenység önmagában;
- egyetlen folyamathivatkozás/hívás (máshol definiált folyamatmodell újrafelhasználása);
- üres vezérlési élszakasz;
- „soros kapcsolás”: a P_1, P_2, \dots, P_n jólstrukturált blokkok szekvenciája (egyszerű vezérlési éssel egymás után kötve őket);
- „fork-join kapcsolás”: a P_1, P_2, \dots, P_n jólstrukturált blokkok egy n ágú fork és egy n ágú join közé zárva;
- „decision-merge kapcsolás”: a P_1, P_2, \dots, P_n jólstrukturált blokkok egy n ágú decision és egy n ágú merge közé zárva;
- „Ciklus”: egy kétágú merge csomóponttal kezdődik, amely után egy jólstrukturált P_1 blokk következik, majd egy kétágú decision, melynek egyik ága a részfolyamat vége, a másik a P_2 jólstrukturált blokkokon keresztül az előbbi merge-be tér vissza.

Egy teljes folyamatmodell jólstrukturált, ha egyetlen belépési pontja (*Flow begin*) és kilépési pontja (*Flow end*) egy jólstrukturált blokkot zár közre.

Amint az a definícióból látszik, egy teljes folyamat lehet úgy jólstrukturált, hogy pl. egy egyszerű elemi tevékenység, egy fork-join blokk és egy ciklus szekvenciájából áll, de csak ha a fork-join blokk és a ciklus maga is kisebb jólstrukturált blokkokból van felépítve.

A jólstrukturáltság célja, hogy áttekinthetőbbé tegye a folyamatot, és hogy bizonyos hibalehetőségeket (pl. holtpont) eleve kizárjon. A folyamatmodellekre jellemző hibamintákról később lesz szó; ezek egy része jólstrukturált modellnél elő sem fordulhat. Ha a folyamat nem jólstrukturált, akkor külön ellenőrzési eljárásokkal kell kizárnai a hibalehetőségeket. Mindemellett megjegyzendő, hogy nem csak a jólstrukturált folyamatmodellek lehetnek értelmesek vagy hasznosak.

Léteznek olyan folyamatmodellezési nyelvek is, amelyek nem engedik tetszőleges gráfként megalkotni a vezérlési folyamot, hanem kizárálag jólstrukturált blokkokat lehet létrehozni és más jólstrukturált blokkokból felépíteni. Ilyen nyelv pl. a BPEL (alap jelkészlete), vagy a programozásoktatásból ismerős Nassi-Shneiderman-féle *strukrogram*. Ezeken a nyelveken a megkötések miatt bizonyos folyamatokat csak körülmenyesebben lehet megfogalmazni; cserébe az adott nyelven készített összes folyamatról külön ellenőrzés nélkül tudható, hogy rendelkezik a jólstrukturáltság fent tárgyalt összes előnyével.

A tanultak szerint az imperatív programnyelvek vezérlésifolyam-gráfja is folyamatmodell; mit jelent a programokra nézve a jólstrukturáltság? A program vezérlési

folyamja akkor lesz jólstrukturált, ha egy belépési és egy kilépési pontja van, és egyszerű utasításokból szekvenciális egymás után fűzéssel áll össze, ill. elágazásokat vagy ciklusokat tartalmaz (ill. megfelelő programozási nyelv/platform esetén akár párhuzamosan végrehajtott blokkokat). A `goto`, `break`, idő előtti `return` és hasonló jellegű ugrások azonban túlmutatnak a jólstrukturáltságon, más szóval kivezetnek a jólstrukturált modellek közül. Ennek megfelelően könnyen átláthatatlanná tehetik a forráskódot, így mértékletes alkalmazásukat szokták javasolni, lehetőség szerint kerülendők. Megjegyzendő, hogy ritkán, de olyan eset is van, ahol épp pl. a `return` használata tesz egy mély vezérlési struktúrát egyszerűbbé.

5.4. Kitekintés*

5.4.1. Technológiák

Adatelemzés

Az Airbnb cég Airflow eszköze¹ adatelemzési folyamatok definiálására és végrehajtására alkalmas.

Több olyan rendszer is létezik, amelyek tudományos folyamatok futtatását teszik lehetővé (*scientific workflow engine*), beleértve az adatok összegyűjtését, elemzését és vizualizálását. Ilyen rendszerek például a Kepler² és a Taverna³.

Üzleti folyamatmodellek

Az üzleti folyamatok modellezésére használt szoftverek manapság tipikusan a BPMN 2.0 szabványt valósítják meg.⁴ Ilyen eszközök például a jBPM⁵, a Bonita BPM⁶, Camunda⁷ és az Eclipse Stardust⁸.

¹<https://github.com/airbnb/airflow>

²<https://kepler-project.org/>

³<http://taverna.incubator.apache.org/>

⁴https://en.wikipedia.org/wiki/List_of_BPMN_2.0_engines

⁵<http://www.jbpm.org/>

⁶<http://www.bonitasoft.com/>

⁷<https://camunda.org/>

⁸<https://www.eclipse.org/stardust/>

6. fejezet

Teljesítménymodellezés

6.1. Alapfogalmak

Teljesítménymodellezéskor egy *rendszert* vizsgálunk, amely *felhasználói kérések* ki-szolgálásához illetve feldolgozásához különböző (véges) *erőforrásokat* használ. Vizsgálatunk fókusza elsősorban az egyes tranzakciók feldolgozási ideje (*válaszidő*), az egységnyi idő alatt feldolgozott tranzakciók száma (*átbocsátás*), illetve az erőforrások *kihasználtásága*, mindez a rendszer *egyensúlyi állapotában*, tehát *átlagos értékek* mérve.

Egy rendszert sokszor alrendszerek együtteseként modellezünk (ilyen alrendszernek tekinthetők az erőforrások is), ilyenkor az egyes fogalmak több szinten is megjelenhetnek. A továbbiakban a teljes rendszer felé érkező felhasználói kéréseket *tranzakcióknak* fogjuk hívni (darabszámának mértékegysége *tr*), az ezek feldolgozása során a rendszer által az alrendszeröknek továbbított feladatrészeket pedig *kéréseknek* (darabszámának mértékegysége *k*). Fontos megjegyezni, hogy valójában ugyanarról a fogalomról van szó, de más *rendszerekre* nézve.¹

Általában is fontos, hogy minden pontosan definiáljuk az éppen vizsgált rendszert. A továbbiakban bemutatásra kerülő képletek szempontjából is fontos a *rendszer határa*. Egy rendszeren belül lehet várakozási sor, illetve feldolgozó egység, utóbbi állhat több alrendszerből is.² Ha egy rendszerben nincs átlapolódás, akkor minden pillanatban (tehát átlagosan is) legfeljebb egy tranzakció lehet a rendszerben. Ha van várakozási sor, vagy több feldolgozó egység is van, akkor definíció szerint van átlapolódás is.

¹Ezért sem tesz különbséget a fogalmak között a tárgyhoz kiadott diasor.

²A rendszer részének tekinthetjük még a hálózati kapcsolatot, és bármi mást, ami késleltetést okozhat, de ettől ebben a segédletben most eltekintünk.

6.2. Rendszerszintű tulajdonságok és a Little-törvény

Definíció. Fogalmak (lásd 6.1. ábra):

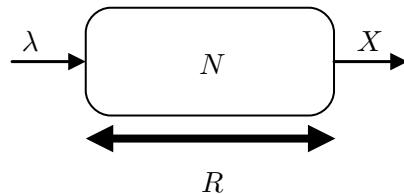
- **Érkezési ráta** (jele: λ): A vizsgált rendszer határához egységnyi idő alatt érkező felhasználói kérések átlagos száma.^a Mértékegysége: db/s.
- **Átbocsátás** (jele: X , mint „throughput”): A vizsgált rendszert egységnyi idő alatt elhagyó feldolgozott felhasználói kérések átlagos száma. Mértékegysége: db/s.
- **Válaszidő** (jele: R , mint „round-trip time”): A felhasználói kérések által a rendszer határain belül töltött átlagos idő. Mértékegysége: s.
- **Rendszerben lévő kérések átlagos száma** (jele: N): Nevezhetnénk az átlapolódás mértékének is. Mértékegység: db.

^aA felhasználói kérés itt lehet tranzakció vagy kérés is, attól függ, honnan nézzük. Ennek megfelelően a darab, mint mértékegység is specializálandó az adott esetnek megfelelően.

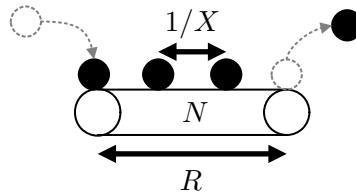
Azt mondjuk, hogy egy rendszer *egyensúlyi állapotban* van, ha $\lambda = X$, vagyis egységnyi idő alatt ugyanannyi új felhasználói kérés érkezik a rendszerbe, mint amennyit ezalatt az idő alatt feldolgozott. Egyensúlyi állapotban igaz a *Little-törvény*:

$$N = X \cdot R$$

Szavakkal, a rendszerben tartózkodó kérések átlagos száma megegyezik az átbocsátás és az átlagos rendszerben töltött idő szorzatával. A rendszert például egy futószalagként (6.2. ábra) elköpzelve ez azt jelenti, hogy ha a szalagon R ideig tart végighaladni, de $1/X = 1/\lambda$ időnként rátessünk egy-egy újabb elemet, akkor R idő múlva az első elem levételenek pillanatában $R/(1/X) = R \cdot X$ elem lesz a szalagon, vagyis a rendszer határain belül.



6.1. ábra. Rendszerszintű tulajdonságok.



6.2. ábra. A Little-törvény szemléltetése.

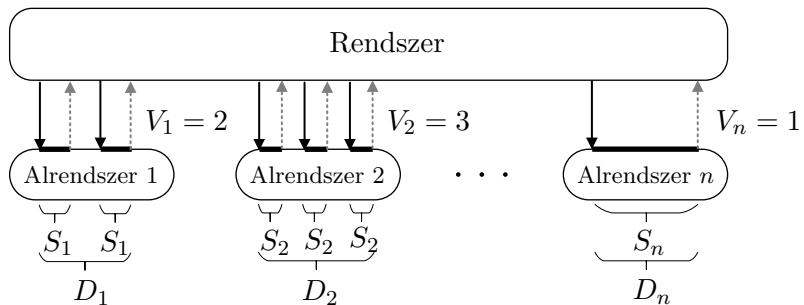
6.3. Erőforrások tulajdonságai

A rendszer tulajdonságai elsősorban a belső szerkezetétől, az alrendszerektől és főképp az erőforrásoktól függ. A rendszerszintű teljesítményjellemzőket ezek tulajdonságaiból kell levezetni. A továbbiakban nullás index jelöli a rendszerszintű tulajdonságokat, míg az i . alrendszer (erőforrás) tulajdonságait i -vel indexeljük.

6.3.1. Rendszerek és alrendszereik kapcsolata

Az egyes alrendszerek és erőforrások teljesítményjellemzőit a következő mértéket felhasználva tudjuk átváltani a rendszer jellemzőire, illetve fordítva:

Definíció. Látogatások átlagos száma (jele: V_i , mint "visits"): Megadja, hogy egy tranzakció átlagosan hány kérést generál az i . alrendszer (erőforrás) felé. Mértékegysége: k/tr (kérés/tranzakció).



6.3. ábra. Rendszer és alrendszereinek kapcsolata.

6.3.2. Felhasználói kérések szolgáltatásigénye

Egy tranzakció „terhelését” a rendszerre nézve a *szolgáltatásigény* fogalmával ragadjuk meg. A szolgáltatásigény az az átlagos időtartam, amíg a tranzakció fel-dolgozása közben a rendszer egy adott erőforrást használ (az egyes kérések során összesen), tehát minden erőforráshoz külön érték tartozik. Az alábbiakban feltételezzük, hogy az alrendszerek erőforrások.³

Definíció. *Szolgáltatásigény* (jele: D_i , mint „service demand”): Megadja, hogy egy tranzakció átlagosan mennyi ideig használja az adott erőforrást (alrendszer). Mértékegysége: $\frac{s}{tr}$.

Definíció. *Erőforrásigény* (jele: S_i , mint „resource demand”): Megadja, hogy egy kérés átlagosan mennyi ideig használja az adott erőforrást (alrendszer). Mértékegysége: $\frac{s}{k}$.

Látható, hogy a két fogalom gyakorlatilag ugyanazt takarja, de az egyik a rendszer szintjén, a másik pedig az erőforrás (alrendszer) szintjén.⁴ A két mennyisége közötti kapcsolatot a látogatások átlagos számának (V_i) segítségével a következő képlet adja meg:

$$D_i = V_i \cdot S_i$$

Látható, hogy itt sem történik semmi meglepő – ha egy kérés átlagosan S_i ideig foglalja az erőforrást, egy tranzakció pedig átlagosan V_i kérést generál, akkor a tranzakció átlagosan D_i ideig fogja használni az erőforrást, tehát az erőforrásra vonatkozó szolgáltatásigénye D_i . Ez az összefüggés látható a 6.3. ábrán is.

6.3.3. Erőforrások kihasználtsága

Véges készletű erőforrások esetén a teljesítmény szempontjából fontos tulajdonság az erőforrások átlagos *kihasználtsága* (jele: U , mint **utilization**), ugyanis ez mutatja meg, hogy a globális teljesítménykorlátoktól nagyjából milyen távol működik a rendszer.

Vegyük észre, hogy az erőforrás, mint alrendszer önmagában is egy rendszert alkot, ezért a 6.2. szakaszban leírtak itt is alkalmazhatók. A felhasználói kérés ekkor a tranzakció által generált kérés, a kérés által a rendszerben töltött átlagos idő

³Ez bizonyos szempontból csak formaság, annyi a jelentősége, hogy erőforrás szint alatt nem foglalkozunk a további alrendszerekkel, itt húzzuk meg vizsgálódásaink határát.

⁴Az elnevezés is csak azért különbözik, hogy külön tudunk hivatkozni a két értékre, az "Erőforrásigény" név pedig kihasználja, hogy most csak az erőforrásokat tekintjük alrendszernek.

(R) pedig az erőforrásigénynek (S_i) felel meg. Az erőforrás, mint alrendszer átbo-csátását az ún. *Forced Flow törvény* segítségével számíthatjuk ki a teljes rendszer átbocsátásából:⁵

$$X_i = V_i \cdot X_0$$

Ezzel felírva a Little-törvényt, az alábbi képletet kapjuk:

$$N_i = S_i \cdot X_i$$

Az N érték tehát szokás szerint megadja, hogy átlagosan hány kérés tartózkodik a rendszerben, ez esetben az erőforráson belül. Az egyes erőforrásokból több példány is rendelkezésre állhat, ezeken belül feltételezzük, hogy nincs átlapolódás. Az erőforrás tehát egyszerre legfeljebb n_i kérést képes kiszolgálni, ahol n_i az i . erőforrásból elérhető példányok száma. Ha az i . erőforrásban, mint rendszerben átlagosan N_i kérés tartózkodik, és ez kevesebb, mint a maximális n_i , akkor a rendszer nem használja ki az erőforrást. Gyakorlatilag ezzel definiáltuk is a kihasználtság fogalmát, amire a *kihasználtság törvénye* ad képletet:

$$U_i = \frac{N_i}{n_i} = \frac{S_i \cdot X_i}{n_i}$$

Az is látható, hogy két darabszámot osztunk el egymással, tehát az eredmény dimenzió nélküli, százalékban kifejezhető arányszám. Az $n = 1$ speciális esetben N értéke közvetlenül megadja a kihasználtság értékét:

$$U_i = S_i \cdot X_i$$

Ilyenkor a kihasználtság úgy is értelmezhető, mint az egységnyi időnek azon hányada, amelyben átlagosan az erőforrás munkát végez. Ez az értelmezés bizonyos szempontból analóg a fizikából ismert hatásfok fogalmával, az erőforrás a vizsgált 1s időben X_i alkalommal S_i ideig hasznos munkát végzett, ez összesen $S_i \cdot X_i$ idő hasznos munkát jelent, ami tehát $\frac{S_i \cdot X_i}{1} = U$ hatásfokot, itt *kihasználtságot* jelent. Több erőforrás példány esetén is hasonló a helyzet, de ilyenkor az egységnyi időt mindegyik példányhoz fel kell számolni.

⁵Figyelem! A mértékegység az alrendserek átbocsátása esetén k/s, a teljes rendszer esetén pedig tr/s!

6.3.4. Az átbocsátóképesség és a szűk keresztmetszet

Az imént láttuk, hogy az erőforráskészlet felső határt szab az elvégezhető munka mennyiségenek, ezáltal az egységes idő alatt kiszolgálható kéréseknek, vagyis az átbocsátásnak. Ezt a felső határt hívjuk **átbocsátóképességnak**.

Meghatározásához az egyes erőforrásokból kell kiindulnunk. Feltételezzük, hogy az erőforrásokat maximálisan kihasználjuk, vagyis $U_i^{max} = 1$. A kihasználtság képleteből az átlagos erőforrásigény (S_i) ismeretében kiszámolhatjuk az adott erőforrás átbocsátóképességét:

$$X_i^{max} = n_i \cdot \frac{U_i^{max}}{S_i} = n_i \cdot \frac{1}{S_i}$$

Az $n_i = 1$ esetben ismét egyszerűbben:

$$X_i^{max} = \frac{U_i^{max}}{S_i} = \frac{1}{S_i}$$

Következő lépésként ki kellene számolnunk a teljes rendszer átbocsátását, de még a teljes rendszerből az erőforrásra következtetni a többi erőforrástól függetlenül is tudtunk a 6.3.3 szakaszban, visszafelé ez most nem lesz igaz. Az egyes erőforrásokat a teljes rendszer ugyanis különböző mértékben használja, így csak az egyikük (esetleg néhányuk, nagyon ritkán mindegyik) korlátozza ténylegesen a rendszer tényleges átbocsátóképességét. Ezt az erőforrást nevezzük **szűk keresztmetszetnek**.

Ki kell tehát számolnunk az egyes erőforrások átbocsátásából a rendszer átbocsátásának elméleti maximumát, majd az így kapott értékek legkisebbikét kell kiválasztanunk, ez lesz a rendszer átbocsátóképessége, az értékhez tartozó erőforrás(ok) pedig a rendszer szűk keresztmetszete(i):

$$X_0^{max} = \min_i \left(\frac{X_i^{max}}{V_i} \right)$$

Az átbocsátóképességgel felírva a Little-törvényt megkaphatjuk N_{max} -ot, vagyis az átlapolódás maximális mértékét a rendszerben. Abban a speciális esetben, amikor tudjuk, hogy a rendszer (erőforrás) minden rendelkezésre áll, és nincs benne átlapolódás, vagyis $N_{max} = 1$ tr, az átbocsátóképesség (de nem az átbocsátás!) és a rendszer átlagos válaszideje között fordított arányosság áll fenn:⁶

$$X_0^{max} = \frac{1 \text{ tr}}{R}$$

⁶Lényegében ugyanez az összefüggés jelenik meg az erőforrás átbocsátóképességének meghatározásakor $R = S_i \cdot 1k$ választással.

6.3.5. A szolgáltatásigény törvénye

Az eddigiekből levezethető még egy összefüggés, ami sokszor jól használható. A *szolgáltatásigény törvénye* egy adott erőforrásra vonatkozó szolgáltatásigény meghatározását teszi lehetővé az erőforrás kihasználtsága és a rendszer átbocsátóképessége segítségével, egyetlen erőforráspéldány ($n_i = 1$) esetén:

$$D_i = \frac{U_i}{X_0}$$

A levezetés a forced flow törvény és a kihasználtság törvényének $n_i = 1$ feltétel melletti egyszerűbb alakja szerint:

$$\underbrace{D_i = V_i \cdot S_i}_{\text{Szolgáltatásigény def.}} = \underbrace{\frac{X_i}{X_0}}_{\text{Forced flow tv.}} \cdot \underbrace{\frac{U_i}{X_i}}_{\text{Kihasználtság tv.}} = \frac{U_i}{X_0}$$

A szolgáltatásigény törvénye lényegében a kihasználtság törvényének olyan átalakítása, hogy az erőforrás-szintű tulajdonságok helyett rendszerszintű tulajdonságokkal számoljon. Az alábbi levezetés ezt az elvet mutatja be, itt lényegében a V_i váltószám segítségével minden oldalon áttérünk az erőforrás szintjéről a rendszer szintjére:

$$U_i = \frac{S_i \cdot X_i}{n_i}$$

$$S_i = n_i \cdot \frac{U_i}{X_i}$$

$$\underbrace{V_i \cdot S_i}_{D_i} = n_i \cdot \underbrace{V_i \cdot \frac{1}{X_i}}_{\frac{1}{X_0}} \cdot U_i$$

$$D_i = n_i \cdot \frac{U_i}{X_0}$$

Itt is jól látható, hogy a törvény eredeti formája az $n_i = 1$ esetre érvényes, minden más esetben megjelenik egy n_i -s szorzó is. A kihasználtság törvényénél látott szemléltető magyarázat ismét alkalmazható: Ha egy tranzakció D_i ideig használja az adott erőforrást, és egy egységnyi idő alatt X_0 ilyen tranzakció történik, akkor az erőforrás az egységnyi idő $\frac{D_i \cdot X_0}{1}$ részéig volt foglalt.

6.4. Átlagos mértékek számítása mérési eredményekből

A fenti értékeket jellemzően mérések vagy szimulációk eredményeképp kapjuk. Ilyen mérések során a rendszert állandósult egyensúlyi állapotban vizsgáljuk, amikor te-hát $\lambda = X_0$. Ilyenkor az alábbi fogalmak jelennek meg, ezek segítségével tudjuk kiszámolni a rendszer és az erőforrások tulajdonságait:

Definíció.

- Mérési idő (jele: T , mint „time”). Mértékegysége: s.
- Tranzakciók száma (jele: C_0 , mint „count”): A mérési idő alatt elvégzett tranzakciók száma. C_i -vel jelölhetjük az egyes alrendszerre vonatkozó értékeket, ha ez szükséges. Mértékegysége: tr (vagy alrendszer esetén k).
- Foglaltsági idő (jele: B_i , mint „busy time”): Az egyes erőforrások foglaltási ideje a mért időtartamon belül. Mértékegysége: s.

Ezekből a fogalmakból könnyedén kiszámítható például egy egypéldányos, átlapolódás nélküli erőforrás ($n_i = 1$) átlagos kihasználtsága a mérés ideje alatt, csupán a foglaltási idő és a mért idő arányát kell kiszámítanunk:

$$U_i = \frac{B_i}{T}$$

A mérés ideje alatt az átlagos átbocsátás (és a rendszer egyensúlya miatt az érkezési ráta is) megkapható a mérési időből és az elvégzett tranzakciók számából:⁷

$$X_0 = \frac{C_0}{T}$$

A tranzakciók átlagos szolgáltatásigénye is megkapható az egyes erőforrásokhoz, ha a foglaltási időket leosztjuk a tranzakciók számával:

$$D_i = \frac{B_i}{C_0}$$

Érdekességképp megjegyezzük, hogy a szolgáltatásigény törvénye ($n_i = 1$ esetre) akár ebből a három összefüggésből is levezethető:

$$U_i = \frac{B_i}{T} = \frac{B_i}{\frac{C_0}{X_0}} = \frac{B_i}{C_0} \cdot X_0 = D_i \cdot X_0$$

⁷Értelemszerűen az egyes alrendszerök vizsgálata esetén az i index használandó a 0 helyett.

7. fejezet

Modellek fejlesztése

8. fejezet

Modellek ellenőrzése

A korábbiakban láthattuk, hogy modelleket számos célból készíthetünk. Függetlenül attól, hogy a modellek felhasználási célja a dokumentáció, a kommunikáció segítése, az analízis vagy a megvalósítás származtatása, a modellek minősége fontos kérdés. Egy hibás modell könnyen vezethet hibás megvalósításhoz, amely pedig a felhasználási területtől függően katasztorfális következményekkel is járhat.

8.1. Követelmények modellekkel szemben

Fontos megjegyezni, hogy a modellek „helyessége” önmagában nem egy értelmes, vizsgálható kérdés. Ennek vizsgálatához fontos tudni, hogy mi a modell célja, kontextusa, mik a követelmények vele szemben.

Példa. Gondoljunk a BKK alábbi egyszerűsített, sematikus metróhálózati térképére!



8.1. ábra. Budapest metróhálózati térképe [?]

Ez tekinthető a metróhálózat egy (gráfával) modelljének. Ha a követelményünk a metróhálózat sematikus reprezentálása, ami alatt például az állomások megfelelő sorrendjét, illetve a helyes átszállási kapcsolatokat értjük, akkor ez a térkép (gráf) helyes. Hibás akkor lenne, ha például a Nyugati pályaudvar az M4-es metró egyik állomásaként lenne feltüntetve. Ha viszont a követelmény az, hogy a térképről leolvashatók legyenek a metróállomások közti távolságok, ez a térkép hibás, hiszen a térkép alapján az Újbuda-központ és Móricz Zsigmond körtér állomások és a Móricz Zsigmond körtér és Szent Gellért tér állomások közti távolság azonos, míg a valóságban az egyik távolság a másiknak közel a duplája.

Látható, hogy önmagában nincs értelme egy modell helyességéről beszélni, csak arról beszélhetünk, hogy bizonyos meghatározott követelményeknek megfelel-e vagy sem. Ebben a fejezetben áttekintjük, milyen követelményeket támaszthatunk a modellekkel szemben, hogyan csoportosíthatjuk és hogyan ellenőrizhetjük ezen követelményeket.

Követelmények funkcionális szerint. Az egyik leggyakoribb felosztás a követelményeket aszerint különbözteti meg, hogy a rendszer elsődleges funkcióját írják le vagy sem.

Definíció. *Funkcionális követelményeknek* nevezzük azokat a követelményeket, amelyek egy rendszer(összetevő) által ellátandó funkciót definiálnak [?].

Definíció. *Nemfunkcionális követelményeknek* (vagy extrafunkcionális követelményeknek) nevezzük az ezeken kívül eső követelményeket, amelyek a rendszer minőségére vonatkoznak, például megbízhatóságra, teljesítményre vonatkozó kritériumok [?].

Megjegyzés. Tehát a *funkcionális követelmények* meghatározzák, *mit* fog a rendszer csinálni. A *nemfunkcionális követelmények* arról szólnak, *hogyan* kell a rendszernek ezeket a funkciókat ellátnia.

Biztonsági és élőségi követelmények. A követelmények egy másik klasszikus kategorizálása alapján biztonsági és élőségi követelményeket különböztetünk meg [?].

Definíció. A *biztonsági követelmények* a megengedett viselkedést definiálják: megadják, hogy milyen viselkedés engedélyezett és mely viselkedések tiltottak. Ezek univerzális követelmények, melyeknek a rendszerre minden időpillanatban teljesülniük kell.

Definíció. Az *élőségi követelmények* az elvárt viselkedést definiálják. Ezek egzisztenciális követelmények, amelyek szerint a rendszer megfelelő körülmények között előbb-utóbb teljesíteni képes bizonyos elvárásokat.

Bizonyos követelmények biztonsági és élőségi követelmények keverékei, így – különösen komplex esetekben – nem feltétlen lehetséges valamelyik kategóriába sorolni a követelményt.

Megjegyzés. Biztonsági követelmény például az, hogy egy jelzőlámpán egyszerre sosem világíthat a piros és a zöld fény. Élőségi követelmény, hogy a lámpa (előbb-utóbb) képes legyen zöldre váltani.

Gyakori általános követelmények. Itt kitérünk néhány olyan gyakori általános követelményre, amelyek általában a modell által leírt folyamatotól, vagy a megvalósított rendszer érdemi funkcionálisától lényegében függetlenek. Az egyik ilyen általános követelmény a holtpontmentesség.

Definíció. Holtpontnak (deadlocknak) nevezzük azt az állapotot egy rendszerben, amelyben a végrehajtás megáll, a rendszer többé nem képes állapotot váltni, és nem mutat semmilyen viselkedést.

A holtpont egy gyakran előforduló oka, ha a rendszerben két vagy több folyamat egymásra várakozik. Ez egy olyan állapot, amelyből külső (nem modellezett) beavatkozás nélkül nem lehet kilépni. Emiatt párhuzamos rendszereknél egy gyakori követelmény a *holtpontmentesség*, a holtpontok lehetőségének hiánya. Sajnos ez egy olyan probléma, amely kifinomult eszközök nélkül igen nehezen vizsgálható, gyakran a holtpont létrejöttéhez a körülmények ritka, különleges együttállása szükséges.

Megjegyzés. Természetesen előfordulhat olyan eset is, hogy a holtpont nem tiltott, hanem megfelelő körülmények között egyenesen elvárt. Emlékezzünk arra, hogy egy folyamatpéldány a dolga végeztével semmilyen további viselkedést nem mutat. Ilyenkor inkább az lehet követelmény, hogy a folyamat csak úgy kerülhessen holtpontba, ha már befejeződött.

Hasonló fogalom a livelock. *Livelock* esetén az érdemi végrehajtás ugyanúgy megáll, mint holtpont esetén. Ennek viszont nem az az oka, hogy a rendszer nem képes állapotot váltni, hanem az, hogy a livelockban résztvevő komponensek egy végtelen ciklusba ragadnak, amelyben nem végeznek hasznos tevékenységet.

Példa. A klasszikus példa livelockra a való életből az, amikor két ember szembetálikozik, és mindenkor udvariasan kitérnének egymás elől, de azt minden megegyező irányba teszik. Ilyenkor az emberek tudnak mozogni („képesek állapotot váltni”), de nem haladnak előre („nem végeznek hasznos tevékenységet”). Holtpontról akkor beszélnénk, ha kölcsönösen nem tudnának megmozdulni a másik személy miatt. Tehát a holtpont esetén a problémát az „örök várakozás”, livelock esetén pedig egy végtelen ciklus okozza.

Állapotgépek esetén gyakori általános követelmények például a *determinizmus* és a *teljesen specifikált* működés. Ezekről bővebben a 4. szakaszban szólunk. A determinizmus és teljesen specifikált működés hasonlóan értelmezhető más viselkedésmodellezési formalizmusok, pl. folyamatmodellek esetén is. Ha a folyamatmodell minden döntési pontjára („decision” csomópont) olyan őrfeltételeket írunk, hogy minden legfeljebb ág legyen engedélyezett, akkor determinisztikus a folyamat; teljesen specifikálnak pedig akkor nevezzük, ha minden legalább egy ág engedélyezett.

Vizsgálatok fajtái. Ha rendelkezésre állnak a követelmények, már lehetséges a modellek helyességének vizsgálata. Azonban a „helyességvizsgálat” nem egy precíz fogalom.

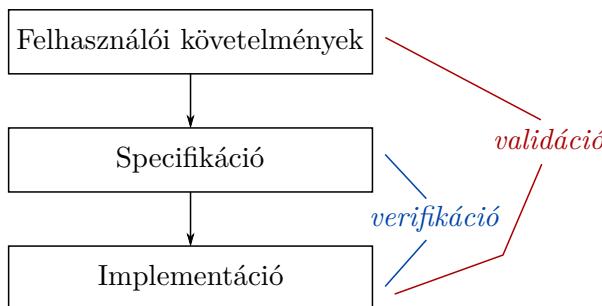
Példa. Képzeljük el, hogy egy kereszteződést szerelünk fel jelzőlámpákkal. A leszállított rendszert ellenőriztük, megfelel a specifikációnak: a fények megfelelő sorrendben, megfelelő időzítéssel követik egymást és minden csak az engedélyezett irányok kapnak egyidejűleg szabad jelzést. Az átadáskor a megrendelő megkérdezi: „– És hol lehet átkapcsolni villogó sárgára?” Mivel ez nem volt a specifikáció része, ilyen funkció nem is került a jelzőlámpák vezérlésébe. Mi meg vagyunk győződve arról, hogy a rendszer helyes, mivel teljesíti a specifikáció minden elemét. Ugyanakkor a megrendelő biztos abban, hogy a rendszer hibás, hiszen nem megfelelő a számára.

Annak érdekében, hogy a helyességvizsgálatról pontosabban beszéljünk, két új fogalmat vezetünk be.

Definíció. *Verifikációt* nevezünk, amikor azt vizsgáljuk, hogy az implementáció (az elkészített modell vagy rendszer) megfelel-e a specifikációnak. Ekkor a kérdés az, hogy helyesen fejlesztjük-e a rendszert, megfelel-e az az előírt kívánalmaknak.

Definíció. *Validációt* nevezünk azt a folyamatot, amelyben a rendszert a felhasználói elvárásokhoz hasonlítjuk, azaz azt vizsgáljuk, hogy a megfelelő rendszert fejlesztjük-e.

Mint ahogyan azt a korábbi példán láthattuk, a sikeres verifikáció nem feltétlen jár együtt sikeres validációval. A verifikáció és a validáció közti különbséget illusztrálja a 8.2. ábra.



8.2. ábra. A verifikáció és a validáció közti különbség illusztrációja

A modellek vagy rendszerek ellenőrzésére többféle módszer is rendelkezésre áll, ezeket mutatjuk be a fejezet hátralévő részében. Először a fontosabb statikus ellenőrzési technikákat ismertetjük (8.2. szakasz), amelyekhez a rendszert nem szükséges futtatni. Utána a tesztelést mutatjuk be (8.3. szakasz), amely egy dinamikus, a rendszert futás közben, de még fejlesztési időben megfigyelő módszer. Ha a rendszert normál üzemű futás közben is meg szeretnénk figyelni, futásidéjű verifikáció-

ról beszélünk, amelyről a 8.4. szakaszban szólunk. A fejezetet a formális ellenőrzési módszerekre történő kitekintéssel zárjuk (8.5. szakasz).

8.2. Statikus ellenőrzés

Definíció. *Statikus ellenőrzés* során a vizsgált rendszert vagy modellt annak véghajtása, szimulációja nélkül elemezzük.

Bizonyos hibák „ránézésre látszanak”, könnyen felismerhetők, ezekre célszerű statikus ellenőrzési módszereket alkalmazni. Ilyenkor a statikus vizsgálat alapvető előnye, hogy gyorsan és könnyen szolgáltat eredményt. Ráadásul a statikus ellenőrzési módszerek gyakran a hiba helyét is pontosan behatárolják, míg például dinamikus módszereknél gyakran a hiba felismerése és annak okának megtalálása két külön feladat. Statikus ellenőrzési technikákat akkor is használunk, amikor szintaktikai hibákat keresünk, illetve a rendszer tipikusan nem is futtatható.

Az alábbiakban részletesen foglalkozunk a statikus ellenőrzés technikáival és használatával. Először a szintaktikai hibák vizsgálatát tekintjük át (8.2.1. szakasz), majd a szemantikai hibák vizsgálatára koncentrálnunk (8.2.2. szakasz).

8.2.1. Szintaktikai hibák vizsgálata

Szintaktikai hibáknak nevezzük azokat a hibákat, amelyek következtében egy modell nem felel meg a metamodelleknek, vagy egy program nem felel meg a használt programozási nyelv formai megkötéseinek. Ilyen lehet például egy állapotokhoz nem kötött állapotátmenet egy állapotgépben vagy egy hiányzó zárójel egy programban.

Grafikus modellek esetén tipikus, hogy a szerkesztő megakadályozza a szintaktikai hibák elkövetését és betartatja a strukturális helyességet, de szöveges leírások esetén ezek elkerülhetetlenek a fejlesztés folyamán. A modern fejlesztőeszközök általában már a beírás során jelzik a szintaktikai hibákat a fejlesztő számára, így azok azonnal javíthatók. Ilyenkor az eszköz beépített szintaktikai statikus ellenőrzőjét láthatjuk működni. Más esetekben (például egyszerű szöveges szerkesztő használata esetén) az esetleges szintaktikai hibákra csak fordítás vagy véghajtás során derül fény.

Általánosan igaz a szintaktikai hibákra, hogy ezek nagy biztonsággal kimutathatók (legkésőbb futtatás vagy véghajtás során) és ritka, hogy egy statikus ellenőrző helyes kódot vagy modellt szintaktikailag hibásnak értékelne¹.

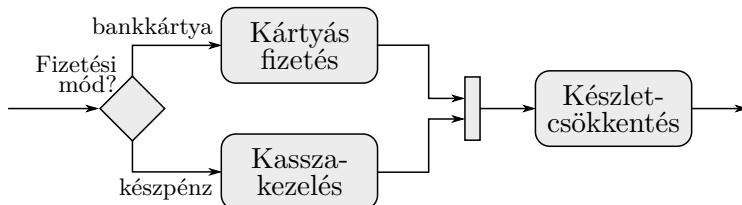
¹ Általánosságban hamis pozitívnak (false positive) találatnak nevezzük azt, ha egy ellenőrző eszköz olyan dolgot jelez hibásnak, amely a valóságban helyes. Ennek fordítottját, azaz amikor az ellenőrző eszköz helyesnek tart valamit, ami valójában hibás, hamis negatív (false negative) találatnak hívjuk. (*)

8.2.2. Szemantikai hibák vizsgálata

Szemantikai hibákról akkor beszélünk, ha a fejlesztés alatt álló rendszer szintaktikailag helyes, ugyanakkor valószínűsíthetően nem értelmes vagy nem az elvárt módon fog viselkedni. Ha egy C programban leírjuk, hogy $x = y / 0;$, akkor az szintaktikailag helyes (feltéve, hogy az x és y változók definiáltak és megfelelő kontextusban szerepel a fenti értékadás), ugyanakkor triviálisan nullával való osztáshoz vezet, amely a legritkábban kívánatos egy programban.

Gyakran a szemantikai problémák nem olyan egyértelműek, mint a fenti nullával osztás. Például az `if (x = 1) ...` C kódban gyanús az értékadás, feltehetően a fejlesztő szándéka az x változó értékétől függő feltételes elágazás implementálása volt, ugyanakkor ezt biztosan nem tudhatjuk. Az ilyen gyanús kódrészleteket angolul *code smell*nek hívjuk, és a statikus ellenőrző eszközök tipikusan ezekre is felhívják a figyelmet.

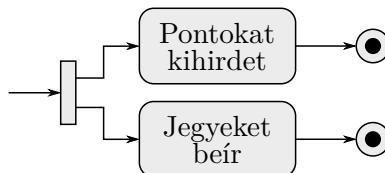
Hasonló probléma folyamatmodellek esetén az alábbi ábrán látható (8.3. ábra):



8.3. ábra. Folyamatmodell decision és join elemmel

A fenti folyamatmodell szintaktikailag helyes, azonban ha közelebbről megvizsgáljuk látszik, hogy valószínűleg szemantikailag helytelen. A decízió elem miatt vagy a Kártyás fizetés, vagy a Kassza-kezelés tevékenység lesz aktív. Mivel a join minden bemeneten tokent fog várni, sosem léphet tovább (tehát holpontra jut), és így a Készletcsökkentés tevékenység sosem fut le.

Hasonló a helyzet az alábbi folyamatmodellnél (8.4. ábra).



8.4. ábra. Folyamatmodell két termináló csomóponttal

A fenti folyamatmodell szintén helyes szintaktikailag, viszont amint a Pontokat kihirdet vagy a Jegyeket beír tevékenység befejeződik, a termináló csomópont leállítja a teljes folyamatot, így a másik tevékenység nem fog tudni lefutni.

Védekezés szemantikai hibák ellen. Két egyszerűbb módszert ismertetünk itt a fenti hibák kivédése érdekében. Az egyik módszerrel a hibák felismerhetők, a másik módszerrel pedig megelőzhetők.

- Ha azonosítottuk a fenti szituációk közül a leggyakrabban előfordulókat, *hibamintákat* fogalmazhatunk meg rájuk. Ez után a statikus ellenőrző eszköz ezeket a hibamintákat keresi a modellben vagy a forráskódban. Például ha egy *decision* elem egy *join* elemmel áll párban egy folyamatmodell vagy a kódban egy *if (< változó > = < érték >)* minta található, erre felhívhatja a felhasználó figyelmét, aki ezután javíthatja a modellt vagy figyelmen kívül hagyhatja a jelzést.
- Lehetőségünk van ezeket a hibákat megelőzni, ha a modellezési vagy programozási nyelv szintaktikájánál önként erősebb megkötéseket alkalmazunk. Ilyen megkötések a kódolási szabályok² (például a mutatók használatának tiltása C-ben) vagy a jólstrukturált folyamatmodellek. *Jólstrukturált folyamatmodellek* esetén kikötjük, hogy a folyamatmodell kizárolag adott minták-ból állítható elő: üres folyamat, elemi tevékenység, szekvencia, ciklus, döntés, párhuzamosság. Ezzel a szintaxist úgy kötjük meg, hogy a tipikus szemantikai hibák ne állhassanak elő. A jólstrukturált folyamatmodellek ről bővebben az 5.3.2. szakaszban szólunk.

Az, hogy mit tartunk szemantikai hibának függ a használt modellezési vagy programozási nyelvtől, de függhet az alkalmazási területtől vagy a konkrét felhasználástól is. Bizonyos alkalmazási területeken további *tervezési szabályokat* definiálunk, amelyek tovább korlátozhatják a modellezés vagy programozás szabadságát. Például biztonságkritikus programok esetén gyakran tiltott a dinamikus memóriafoglalás. Ilyen esetekben kiegészíthetjük a hibaminták készletét a `malloc` és hasonló konstrukciókkal.

Szimbolikus végrehajtás. Bizonyos esetekben a szemantikai hibák kiszűrése bonyolultabb feladat. Gondoljunk például az `x = y / z`; kódrészletre. Előfordulhat itt nullával osztás? A válasz nem egyértelmű, függ `z` értékétől.

²Az egyik leghíresebb kódolási szabálygyűjtemény a C nyelvhez a *MISRA C*. Eredetileg közúti járművek beágyazott rendszereihez fejlesztették, de valójában bármilyen beágyazott rendszer esetén használható. Ilyen felhasználási területen a hibák komoly következményekkel járhatnak, ráadásul javításuk igen nehéz. Ezért olyan megkötéseket alkalmaznak a szoftverek fejlesztésekor, amelyek a kódot átláthatóbbá, egyértelműbbé teszik, amely írása közben nehezebb hibát véteni. Például a száznál több MISRA C szabály egyike megtiltja az *if (x == 0 && ishigh)* kódrészlet használatát, helyette az *if ((x == 0) && ishigh)* formátum használandó. Bár ez a két kódrészlet nyilvánvalóan ekvivalens, ha a logikai műveletek operandusai csak atomi kifejezések (pl. *ishigh*) vagy zárójelezett részkifejezések lehetnek, kisebb egy hibás precedencia feltételezésének valószínűsége [?].^(*)

Példa. Tekintsük például az alábbi C kódot:

```
int foo(int z) {
    int y;

    y = z + 10;
    if (y != 10) {
        x = y / z;
    } else {
        x = 2;
    }
    return x;
}
```

Lehet z értéke 0? Természetesen, hiszen z egy bemeni változó. Vizsgáljuk a z -vel osztás előtt annak az értékét? Nem, csak y változót vizsgáljuk. Lehetséges a nullával osztás a kódban? Nem. Amikor z értéke nulla lenne, akkor y értéke 10 lesz, és így az osztást tartalmazó utasítás nem fut le.

Amikor végrehajtunk egy programot, minden a változók bizonyos konkrét értékei mellett tesszük ezt (pl. `foo(5)`). *Szimbolikus végrehajtás* esetén konkrét értékek helyett szimbolikus értékekkel „imitáljuk” a végrehajtást, azaz a változókat matematikai változóként fogjuk fel. Emellett a belső elágazások által támasztott feltételeket is összegyűjtjük, majd ezen információk alapján következtetünk az egyes változók értékeire a program adott pontjain, vagy egyes programrészek elérhetőségére.

Példa. Ha z -t egy z matematikai változónak tekintjük, akkor tudjuk, hogy a feltételes elágazás *igaz* ágában az $y = z + 10$; utasítás miatt $y = z + 10$ igaz, valamint az elágazási feltételben szereplő $y \neq 10$ miatt $y \neq 10$ igaz. A kettőből együttesen $z + 10 \neq 10$, azaz $z \neq 0$. Így bizonyíthatjuk, hogy sosem osztunk nullával a fenti példakódban.

Példa. Milyen esetekben ad a fenti példakód 2-t eredményül? Erre szintén választ kaphatunk szimbolikus végrehajtás segítségével, míg konkrét végrehajtással minden egyes lehetséges z -re le kellene futtatnunk a függvényt. Ha az elágazás feltétele teljesült, tudjuk, hogy a program végére $x = \frac{z+10}{z} \wedge z \neq 0$, azaz x akkor lesz 2, ha z értéke 10. Ha az elágazás feltétele nem teljesült, akkor tudjuk, hogy a program végére $x = 2 \wedge z = 0$, azaz ha z értéke 0, a kimenet 2 lesz. Tehát $z=0$ és $z=10$ esetén kaphatunk eredményként 2-t.

8.3. Tesztelés

Definíció. *Tesztelés* alatt olyan tevékenységet értünk, amely során a rendszert (vagy egy futtatható modelljét) bizonyos meghatározott körülmények között futtatjuk (vagy szimuláljuk), majd az eredményeket összehasonlítjuk az elvárásainkkal^a [?]. A tesztelés célja a vizsgált rendszer minőségének felmérése és/vagy javítása azáltal, hogy hibákat azonosítunk.

^aEnnél a nagyon általános és gyakran használt fogalomnál kivételesen érdemes az eredeti, angol nyelvű, az IEEE által adott definíciót is elolvasni: „[Testing is an] activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component”. [?]

Láthatjuk az első szembetűnő különbséget a tesztelés és a statikus ellenőrzés között: utóbbi esetben a vizsgált rendszert nem futtatjuk, nem hajtjuk végre. Ugyanakkor attól, hogy a rendszert annak vizsgálata céljából végrehajtjuk, még nem beszélünk tesztelésről. Ahogyan a definíció is mutatja, meghatározott körülmények között futtatjuk a rendszert, azaz nem „próbálgatásról” van szó, hanem a fejlesztés egy alaposan megtervezett részfolyamatáról.

Azt is érdemes megjegyezni, hogy a tesztelés és a „debugolás” is két külön fogalom. Tesztelés esetén hibajelenségek meglétét vagy hiányát vizsgáljuk. Debugolás esetén ismert egy bizonyos hibajelenség (pl. egy teszt kimutatta vagy egy felhasználó jelezte), a célunk pedig ennek a helyének, a kiváltó okának megkeresése, lokalizálása.

Megjegyzés. Érdemes megjegyezni, hogy a tesztelésre számos különböző definíció létezik. Az International Software Testing Qualifications Board (ISTQB) szervezet által adott definíció például beleérte a tesztelésbe az olyan statikus technikákat is, mint például a követelmények vagy a forráskód átolasása, és a forráskód statikus ellenőrzése [?]. Jelen tárgy keretei közt mi a fenti definíciót használjuk és a tesztelést dinamikus technikának tekintjük.

8.3.1. A tesztelés alapfogalmai

Már a definíció alapján is látszik, hogy a teszteléshez nem elég önmagában a rendszer. Tesztek végrehajtásához legalább a következő három komponensre szükséges: a tesztelendő rendszer, a tesztbemenetek és a tesztorákulum.

Definíció. *Tesztelendő rendszer* (*system under test, SUT*): az a rendszer amelyet a teszt során futtatni fogunk vizsgálat céljából.

Definíció. *Tesztbemenetek*: a tesztelendő rendszer számára biztosítandó bemeneti adatok.

Definíció. *Tesztorákulum*: olyan algoritmus és/vagy adat, amely alapján a végrehajtott tesztről eldönthető annak eredménye.

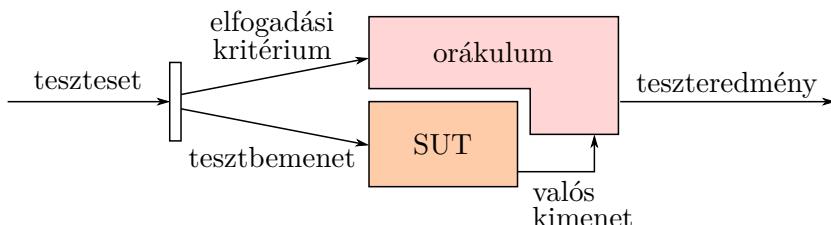
Definíció. Tesztesetnek hívjuk összefoglaló néven azon adatok összességét, amelyek egy adott teszt futtatásához és annak értékeléséhez szükségesek. Tehát a teszteset „bemeneti értékek, végrehajtási előfeltételek, elvárt eredmények (elfogadási kritérium) és végrehajtási utófeltételek halmaza, amelyeket egy konkrét célért vagy a tesztért fejlesztettek” [?, ?].

Definíció. Tesztkészletnek hívjuk a tesztesetek egy adott halmazát.

Definíció. Tesztfuttatás: egy vagy több teszteset végrehajtása [?].

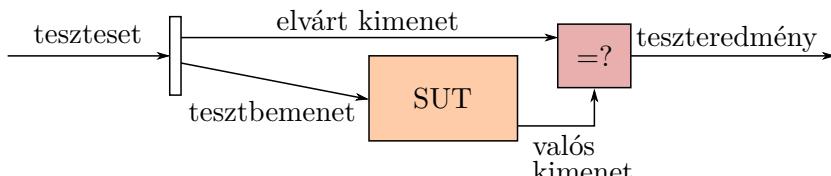
A tesztfuttatás után [?] – az orákulum segítségével – megtudjuk a teszt eredményét, amely lehet *sikeres* (pass), *sikertelen* (fail) vagy *hibás* (error). Utóbbi esetben a tesztről nem tudjuk eldönteni, hogy sikeres-e vagy sem. Ilyen lehet például, ha a tesztrendszerben történt hiba, emiatt a SUT helyességéről nem tudunk nyilatkozni. Általában a teszt eredménye a kapott és a tesztesetben megfogalmazott elvárt kimenetek összehasonlításával kapható meg, de származhat egy referenciaimplementációval összehasonlításból, vagy ellenőrizhetünk implicit elvárásokat, például azt, hogy a kód nem dob kivételt.

A tesztelés általános elrendezése látható a 8.5. ábrán.



8.5. ábra. A tesztelés általános sémája

Legegyeszerűbb esetben a teszteset közvetlenül tartalmazza az adott tesztbemenetekre elvárt kimeneteket (referenciakimeneteket), ahogyan az a 8.6. ábrán látható. Így az orákulum feladata mindenkorra a SUT kimenetének összevetése a tesztesetben leírt elvárt kimenetekkel.



8.6. ábra. A tesztelés menete ismert referenciakimenet esetén

A 8.6. ábrán vázolt elrendezésről van szó például akkor, ha állapotgépeket tesztünk, és a teszeset tartalmazza a bemeneti eseménysort (tesztbemenetként) és az elvárt akciókat, eseményeket (elvárt kimenetként).

Nincs feltétlen lehetőség azonban referenciakimenet megadására olyan teszeseteknél, amelyek deklaratív követelményeket ellenőriznek, vagy többféle kimenetet is megengednek. Ilyenkor speciális orákulum szükséges. Például egy prímtesztelő eljárással szembeni követelmény lehet, hogy amennyiben a bemeneten kapott szám összetett, akkor bizonyítékul a kimeneten be kell mutatni a bemenetként kapott szám egyik valódi osztóját. Ilyenkor a tesztorákulumnak többféle kimenetet is el kell fogadnia. Ehhez például megvizsgálhatja a bemenet oszthatóságát a SUT által adott kimenettel.

Példa. Tekintsük például az alábbi szignatúrájú függvényt:

```
void find_nontrivial_divisor(int n, bool& is_prime, int& divisor).
```

A függvény a megadott **n** egész számra visszadja, hogy prím-e (**is_prime**). Amennyiben a szám nem prím, ennek igazolására megadja egy valódi (1-től és **n**-től eltérő) osztóját (**divisor**).

Egy összetett számnak számos valódi osztója lehet, emiatt a függvény megvalósításának tesztelésénél nem adhatunk meg konkrét elvárt értékeket. Ehelyett azt vizsgálhatjuk, hogy a visszaadott szám (**divisor**) tényleg osztója-e **n**-nek és tényleg 1-től és **n**-től eltérő. Így például az alábbi teszeseteket tervezhetjük meg a **find_nontrivial_divisor** függvény vizsgálatára:

Teszeset	Bemenet (n)	Elfogadási kritérium
1	997	<code>is_prime==true</code>
2	998	<code>is_prime==false && divisor>1 && divisor<998 && 998%divisor==0</code>
3	999	<code>is_prime==false && divisor>1 && divisor<999 && 999%divisor==0</code>
4	1	<i>kivétel dobása</i>

Ezen a példán az is megfigyelhető, hogy a teszeseteket általában nem véletlenszerűen, hanem gondos tervezés alapján határozzuk meg. Például fontos, hogy a különleges, **n==1** esetet is megvizsgáljuk. A tesztervezéssel bővebben a *Szoftver- és rendszerellenőrzés* (BMEVIMIMA01) MSc tárgyban^a foglalkozunk.

^a<https://inf.mit.bme.hu/edu/courses/szore>

8.3.2. A tesztek kategorizálása*

Tesztelést a szoftverfejlesztési életciklus számos fázisában használhatunk. Attól függen, hogy a rendszer mekkora részét vizsgáljuk, különböző tesztelési módszereket különböztethetünk meg.

- *Modultesztnek* (másként *komponensteszt* vagy *egységeszt*) nevezzük azt a tesztet, amely csak egyes izolált komponenseket tesztelnek [?].
- *Integrációs tesztnek* nevezzük azt a tesztet, „amelynek célja az integrált egységek közötti interfésekben, illetve kölcsönhatásokban lévő hibák megtalálása” [?].
- *Rendszertesztnek* hívjuk azt a tesztet, amelyben a teljes, integrált rendszert vizsgáljuk annak érdekében, hogy ellenőrizzük a követelményeknek való megfelelőséget [?].

Ezek a különféle tesztelési módszerek általában egymást követik a fejlesztési ciklusban: először az egyes modulok tesztelése történik meg, később a modulok integrációja után az integrációs tesztek, majd végül a rendszerteszt kerül elvégzésre.

Amennyiben módosítást végezünk a rendszerünkön, a korábbi tesztek eredményeit már nem fogadhatjuk el, hiszen a rendszer megváltozott. Ha ismerjük, hogy az egyes tesztesetek a rendszer mely részeit vizsgálják, elegendő azokat újrafuttatnunk, amelyek a megváltoztatott részt (is) vizsgálják. Az ilyen, változtatások utáni (szelektív) újratesztelést hívjuk *regressziós tesztnek*. Fontos megjegyezni, hogy ez a korábbiakhoz képest egy ortogonális kategória, és egységeket vagy teljes rendszert is vizsgálhatunk regressziós teszteléssel.

8.3.3. A tesztelés metrikái

Ahogyan a fejezet elején írtuk, a tesztelés általános célja a vizsgált rendszer minőségének javítása hibák megtalálásán és javításán keresztül. Nyilvánvaló, hogy ez a végtelenségig nem folytatható, egy idő után az összes hibát megtaláljuk és kijavítjuk. Ennél az utópisztikus néztnél kissé pragmatikusabban azt is mondhatjuk, hogy egy idő után a tesztelés folytatása nem célszerű (nem gazdaságos), mert a vizsgált rendszer minősége már „elég jó”. De honnan tudhatjuk, hogy elértek ezt a szintet?

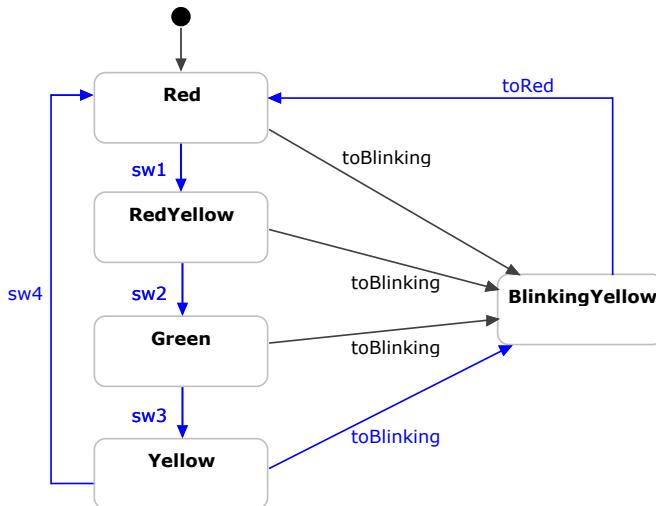
Az egyik gyakran használt módszer a tesztkészlet fedésének mérése. A tesztfedettség alapötlete az, hogy a tesztkészlet egyik tesztesete sem látogat meg egy adott állapotot egy állapotgépen, akkor az az állapot biztosan nem lesz vizsgálva, így annak minőségéről következtetést nem vonhatunk le. Ugyanez elmondható például egy metódus hívásával kapcsolatban is.

Ha viszont a tesztkészletünk meglátogat minden állapotot vagy meghív minden metódust, elmondhatjuk, hogy minden megvizsgáltunk? Sajnos korántsem. Például

abból, hogy minden állapotot bejár egy tesztkészlet nem következik, hogy minden állapotátmenetet is érint. Attól, hogy egy tesztkészlet minden metódust meghív, nem feltétlenül érint minden utasítást. Látható, hogy számos fedettségi metrikát lehet bevezetni. Mi itt mindössze az alábbi három alapvető fedettségi metrikára szorítkozunk.

- Egy állapotgépben az *állapotfedettség* (vagy állapotlefedettség) egy adott tesztkészlet által érintett (bejárt) állapotok és az összes állapotok arányát adja meg.
- Egy állapotgépben az *átmenetfedettség* (vagy átmenetlefedettség) egy adott tesztkészlet által érintett (bejárt) állapotátmenetek és az összes átmenetek arányát adja meg.
- Egy vezérlési folyamban (programban) az *utasításfedettség* (vagy utasításlefedettség) egy adott tesztkészlet által érintett (bejárt) utasítások és az összes utasítások arányát adja meg.

Példa. Tekintsük az alábbi egyszerű állapotgépet, amely egy közlekedési lámpát modellez.



Vegyük az alábbi, két teszesetből álló tesztkészletet (az elfogadási kritérium nem lényeges a fedettség szempontjából, így azt elhagytuk):

Teszteset	Bemenet (n)
1	sw1, sw2, sw3, sw4
2	sw1, sw2, sw3, toBlinking, toRed

Ez a két teszt az összes állapotot be fogja járni. Így az állapotgépben az **állapotfedettség** $\frac{5}{5} = 1 = 100\%$.

Ugyanakkor ez a tesztkészlet mégsem érzékeny minden hibára. Nem tudnánk kimutatni például, ha az implementációban kifejejtenénk a RedYellow-ból BlinkingYellow-ra vezető tranzíciót. Ez azért van, mert bár az állapotfedettség teljes, az átmenetfedettség nem. Összesen 9 tranzíció található az állapotgépen, ebből 6-ot fed le a két teszeset együtt (kékkel jelölt átmenetek), így az **átmenetfedettség** $\frac{6}{9} = 0,666 = 66,6\%$.

Fontos megjegyezni, hogy ha elérnénk néhány új teszeset definíálásával a teljes átmenetfedettséget, az sem feltétlen lenne képes kimutatni minden lehetséges hibát. Ha például az implementációban – hibásan – szerepelne egy Green állapotból Red állapotba vezető átmenet, azt nem feltétlen tudnánk kimutatni egy (a specifikáción) teljes fedettséget elérő tesztkészlettel sem.

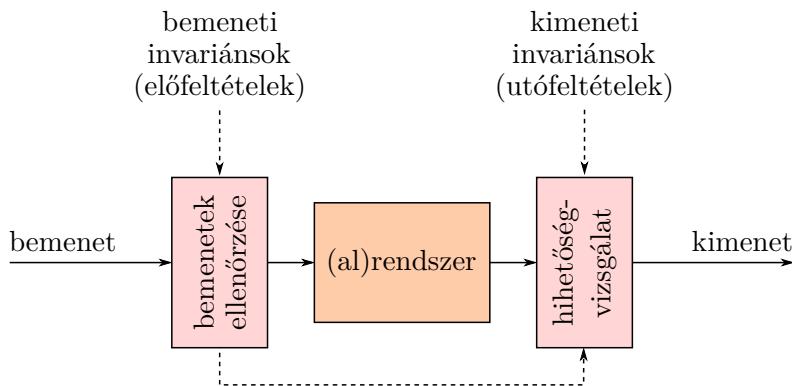
A fenti áttekintésből az is látszik, hogy egy magas fedettségi arány csak szükséges, de nem elég séges feltétele a jó minőségű rendszer fejlesztésének. Gyakran ez a szám

félrevezető is lehet, illetve rossz irányba viheti a teszttervezést.

8.4. Tesztelés futásidőben (futásidejű verifikáció)

Ebben a fejezetben a futásidejű „öntesztelés” vagy monitorozás alapötletét mutatjuk be. Bizonyos esetekben kiemelkedően magas minőségi elvárásaink vannak a rendszerünkkel szemben (pl. biztonságkritikus alkalmazási területek). Más esetekben olyan külső komponenseket használunk, amelyek minőségéről nem tudunk alaposan meggyőződni (pl. egy lefordított, más által fejlesztett alkalmazást csak korlátozottan tudunk tesztelni). Ilyenkor az elvárásaink egy részét elhelyezzük magában a megvalósított rendszerben és folyamatosan vizsgáljuk. Azokat a követelményeket, amelyek teljesülését folyamatosan, minden állapotban elvárjuk, *invariánsoknak* nevezzük.

A monitorozás általános elrendezését szemlélteti a 8.7. ábra.



8.7. ábra. A monitorozás általános elrendezése

A monitorozás két fő lépésből áll:

- *bemenetek ellenőrzéséből*, amely során a bemeneti adatok megfelelőségét vizsgáljuk a definiált bemeneti invariánsok (előfeltételek) alapján, és/vagy
- *hihetősgégvizsgálatból*, amely során a kimeneti adatok megfelelőségét vizsgáljuk a bemeneti adatok és a definiált kimeneti invariánsok (utófeltételek) alapján.

Egyes esetekben az invariánsok igen egyszerűek (például egy valós számok négyzetre emelést megvalósító függvény végén vizsgálhatjuk, hogy a kapott eredmény negatív-e; a negatív eredményt hibásnak minősítjük). Ilyenkor tipikusan az implementáció is három részre tagolódik, követve a 8.7. ábrán látható elrendezést:

- Először az *előfeltételt* vizsgáljuk. Ha ez nem teljesül, kivételről beszélünk. Ez egy normálistól eltérő, váratlan helyzet, aminek a kezelését máshol valósítjuk meg (ilyen körülmények között az implementációt helyességét nem követeljük)

meg). Ha az előfeltétel nem teljesül, annak oka a rendszer hibás használata (nem megfelelő bemeneti adatokat kapott).

- Amennyiben az előfeltétel teljesült, megtörténik az érdemi logika *végrehajtása*.
- A végrehajtás után az utófeltétel vizsgálatára kerül sor. Amennyiben az utófeltétel nem teljesül, olyan hibás állapotba került a rendszer, amely kezelésére nincs felkészítve. Ennek oka lehet a hibás implementáció vagy futásidéjű hiba.

Példa. Az alábbi példakód egy másodfokú egyenlet gyökeit számolja ki:

```
void Roots(float a, b, c, float &x1, &x2) {  
    float d = sqrt(b*b-4*a*c);  
  
    x1 = (-b+d)/(2*a);  
    x2 = (-b-d)/(2*a);  
}
```

Tudhatjuk, hogy ez a kód nem működik helyesen minden esetben. Feltételezzük, hogy a diszkrimináns ($D = b^2 - 4 \cdot a \cdot c$) nemnegatív, különben a gyökvonást negatív számon végezzük el. Tudjuk azt is, hogy a kiszámított x_1 és x_2 értékeknek zérushelyeknek kell lennie, azaz elvárt, hogy $ax_1^2 + bx_1 + c = 0$ és $ax_2^2 + bx_2 + c = 0$. Ezekkel az elő- és utófeltételekkel kiegészíthetjük az implementációt is az alábbiak szerint:

```
void RootsMonitor(float a, b, c, float &x1, &x2) {  
    // előfeltétel  
    float D = b*b-4*a*c;  
    if (D < 0)  
        throw "Invalid input!";  
  
    // végrehajtás  
    Roots(a, b, c, x1, x2);  
  
    // utófeltétel  
    assert(a*x1*x1+b*x1+c == 0 && a*x2*x2+b*x2+c == 0);  
}
```

Monitorozást nem csak ilyen egyszerű esetekben lehet használni, összetett monitorkód is elképzelhetők. Például állapotgépek esetén készíthetünk egy monitor régiót, ami a rendszer megvalósításával párhuzamosan fut és detektálja a hibás vagy tiltott állapotokat, akciókat.

Megjegyzés. Az elő- és utófeltételek adják az ún. *design by contract* elv alapötletét. Ennek célja a rendszer minőségének javítása (a hibák elkerülése) azáltal, hogy a rendszer minden komponensre elő- és utófeltételeket határozunk meg. Egy adott x komponenst felhasználó y komponensnek

garantálnia kell, hogy x előfeltételei teljesülnek a felhasználáskor, cserébe feltételezheti, hogy a válasz teljesíteni fogja x utófeltételeit. A másik oldalról x feltételezi, hogy az előfeltételei teljesülni fognak, és az ő felelőssége az utófeltételek teljesítése. Egy komponensre az elő- és utófeltételek összességét *szerződésnek* hívjuk, amely lényegében az adott komponens egy specifikációja. A legtöbb programozási nyelv beépítve vagy kiegészítésekben keresztül támogatást nyújt a szerződések precíz leírásához és esetleg azok automatikus ellenőrzéséhez is.

8.5. Formális verifikáció*

Formális verifikáció alatt olyan módszereket értünk, amelyek segítségével adott modellek vagy programok helyességét matematikailag precíz eszközökkel vizsgálhatjuk. Hárrom fontosabb formális verifikációs módszert (családot) említünk meg:

- Modelellenőrzés;
- Automatikus helyességbizonyítás, amely során axiómarendszerek alapján tételbizonyítás segítségével próbáljuk a helyességet belátni;
- Konformanciavizsgálat, amely során adott modellek között bizonyos konformaciarelációk teljesülését vizsgáljuk, így beláthatjuk, hogy különböző modellek viselkedése megegyező vagy eltérő az adott relációk szerint.

Jelen jegyzetben kitekintésként a modelellenőrzést mutatjuk be röviden. Bővebben a formális verifikációról a *Formális módszerek* (BMEVIMIM100) MSc tárgy³ keretei közt szólunk.

Modelellenőrzés. A *modelellenőrzés* egy olyan módszer, amelynek során egy adott modellen vagy implementáción egy követelmény teljesülését vizsgáljuk. A modelellenőrzés egyik előnye, hogy amennyiben a követelmény nem teljesül, lehetséges egy ellenpéldát adni. Az *ellenpélda* egy olyan futási szekvencia, amely megmutatja, hogyan lehetséges a vizsgált követelményt megsérteni. Ez nagyban segíthet a hibás működés okának meghatározásában.

A modelellenőrzés – a teszteléssel szemben – egy *teljes* módszer, azaz az adott modell vizsgálata kimerítő. Ennek következtében lehetőség van a helyes működés bizonyítására is, míg ez teszteléssel nem lehetséges. Ugyanakkor a modelellenőrzés igen nagy számítási igényű, ezért használhatósága korlátozott.

8.6. Gyakorlati jelentőség

Mint azt már a fejezet elején említettük, az informatikának, mint mérnöki diszciplínának kulcsfontosságú része az elkészített munka ellenőrzése, legyen az specifikáció

³<https://inf.mit.bme.hu/edu/courses/form>

vagy implementáció. Minél nagyobb az elkészített rendszer kritikussága, annál nagyobb a fejlesztés folyamán az ellenőrzése szerepe.

Manapság már szinte elképzelhetetlen egy modern fejlesztőkörnyezet beépített statikus ellenőrzés nélkül. Elérhetők további, igen kifinomult eszközök, amelyek statikus ellenőrzés segítségével felhívhatják a figyelmet hibákra vagy veszélyes konstrukciókra.

Az általunk írt implementáció nem tekinthető befejezettnak, amíg nem tervezünk és implementáltunk hozzá egy megfelelő tesztkészletet. Természetesen nem csak a megfelelően releváns tesztkészlet megvalósítása az elvárás: az implementációtól a teszteknek sikeresnek kell lenniük ahhoz, hogy a fejlesztés következő fázisára továbbléphessünk, legyen az akár az integráció, akár a megrendelőnek történő átadás.

Különösen kritikus esetekben, például egy repülő vagy egy atomerőmű vezérlőrendszerénél, netán orvosi eszközök beágyazott szoftvereinél a tesztelés ismertetett hibái túlzott kockázatot hordozhatnak, ezért gyakran kiegészül a tervez és a megvalósítás vizsgálata formális módszerek használatával.

8.6.1. Kapcsolódó eszközök

Ebben a szakaszban néhány olyan (többnyire jól ismert és ingyenes) eszközt sorunk fel, amely a gyakorlatban megvalósítja a bemutatott módszerek némelyikét.

Egyszerűbb statikus analízis eszközöket beépítve találhatunk a fejlettebb fejlesztő-eszközökben (pl. Eclipse⁴) és modellező eszközökben (pl. Yakindu). Ezeken kívül számos, mélyebb analízist lehetővé tevő eszköz létezik. C esetén gyakran használatos a Lint stb. nyelv esetén a Cppcheck⁵ és cpplint⁶ eszközök. Java nyelvű programok statikus ellenőrzésére használható például a FindBugs⁷ vagy a PMD⁸ eszköz. A Coverity cég C, C++ és Java nyelvű programok statikus ellenőrzésére kínál megoldást, amelyek közül például a Coverity Scan⁹ nyílt forráskódú programokra ingyenesen használható.

C és C++ nyelvű programokhoz használható tesztfuttató keretrendszer például a Google Test¹⁰. Java programok modultesztelését segítheti például a JUnit¹¹ keretrendszer.

⁴<http://eclipse.org>

⁵<http://cppcheck.sourceforge.net/>

⁶<https://github.com/google/styleguide/tree/gh-pages/cpplint>

⁷<http://findbugs.sourceforge.net/>

⁸<http://pmd.github.io/>

⁹<http://www.coverity.com/products/coverity-scan/>

¹⁰<https://github.com/google/googletest>

¹¹<http://junit.org/>

C és C++ nyelvű szoftverek modellellenőrzésre jól használható például a CBMC¹² eszköz. „Állapotgép-jellegű” modellek esetén jó választás lehet az UPPAAL¹³ vagy a nuXmv¹⁴ eszközök használata.

¹²<http://www.cprover.org/cbmc/>

¹³<http://www.uppaal.org/>

¹⁴<https://nuxmv.fbk.eu/>