Folyamatmodellezés

Hibatűrő Rendszerek Kutatócsoport

2017

Tartalomjegyzék

Bevezetés

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. Látni fogjuk, hogy a szoftverek programkódjának elemzése is folyamatmodellre vezet.

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 lelhető 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 *Állapot alapú modellezés* 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 átlapolódva. 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 valamilyen célra vezet.

2. A folyamatmodellek építőkövei

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

2.1. Elemi tevékenység

Mielőtt valódi folyamatmodelleket vizsgálnánk, először meg kell ismerkednünk azzal az esettel, amikor valamilyen viselkedés részleteit *nem* modellezzük 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 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 modellezzük.

1. ábra. Elemi tevékenység grafikus szintaxisa

Példa. Mit is értünk azalatt, hogy a fordítóprogram futtatása időbeli kiterjedéssel bír? Ahogy a 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, befejezettnek tekinthető.

2. ábra. Elemi tevékenység időbeli lefutása idődiagramon

Ahogy a fenti példa is illusztrálja, minden elemi tevékenység önmagában egy háromelemű állapotteret határoz meg: {még nem kezdődött el, folyamatban van, már befejeződött}; később az összetett

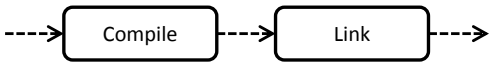
folyamatmodellek állapotteréről is lesz szó. Látható, hogy az elemi tevékenység is leírható a korábban tanult állapotmodellezési eszköztárral; ebben az esetben viszont más a modell fókusza, más elemeket tartunk elnevezésre és vizsgálatra érdemesnek.

Megjegyzés. Bizonyos források *atomi* 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 *atomi 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.

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 élek (vezérlésifolyam-élek)* 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 befejezte után kezdhető meg a linkelés. A 3. ábrán ennek a *szekvenciának* a jelölését látjuk; a szaggatott nyíl rákövetkezést jelöl, tehát hogy a *Compile* tevékenység vége után kezdhető meg a *Link*.



3. ábra. Szekvencia grafikus szintaxisa

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.

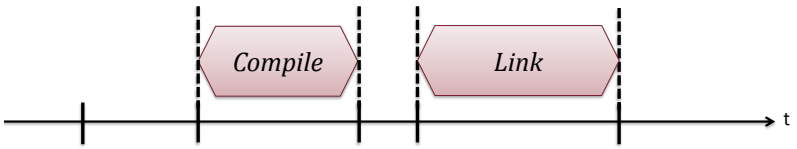
A következőkben több külön módszerrel értelmezzük a *szekvencia* szemantikáját. Bár ez a vizsgálat feleslegesen alaposnak és szájbarágósnak tűnhet („ágyúval verébre”), de a később előkerülő összetettebb folyamatmodell-konstrukciók megértését nagymértékben segíti.

Példa. Hogyan szimulálhatjuk a szekvenciánk működését? Ha a 3. ábrán látható folyamatdiagramot kinyomtatjuk, és a papírra helyezünk egy régi egyforintost (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 mindkét tevékenység }. A folyamatmodell határozza meg ezt az öt állapotot, valamint hogy milyen állapotátmenetek megengedettek köztük (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 mindkét tevékenység }. A folyamatmodell határozza meg ezt az öt állapotot, valamint hogy milyen állapotátmenetek megengedettek köztük (jelen esetben csak a felsorolás sorrendjében lehet állapotot váltani). A folyamatmodell tehát ezt az állapotmodellt indukálja, a folyamat szimulációját pedig visszavezettük a fent konstruált állapotmodell szimulációra. A szimuláció eredményeképpen a 4. ábrán látható idődiagramhoz hasonló eredményt kapunk.



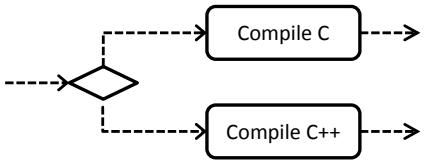
4. ábra. Szekvencia időbeli lefutása

2.3. Elágazás, őrfeltétel

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. 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 élre 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 élre helyezzük.



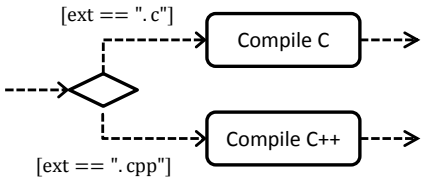
5. ábra. Elágazás grafikus szintaxisa

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 *nemdeterminizmust* 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 modelleknél 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 kizárjuk a döntés után előálló ágak némelyikét; 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 eldönthető, hogy melyik nyelv fordítóprogramját kell végrehajtani. A 6. ábrán látható folyamatmodell őrfeltételekkel fejezi ki.



6. ábra. Elágazás grafikus szintaxisa őrfeltételekkel

A korábban bevezetett fogalmaknak megfelelően akkor determinisztikus a folyamatmodell, ha minden egyes elágazás minden végrehajtásánál kizárják 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.

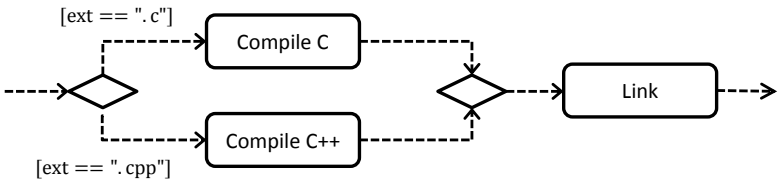
Összefoglalásként tehát ezt a definíciót alkothatjuk:

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.

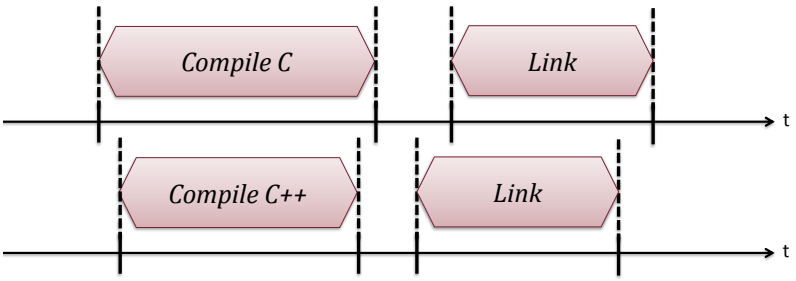
2.4. Merge (besorolódás)

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 mindenképp a linkelés következik. A 7. ábrán feltüntetett merge csomópont (rajzjele egyezik a döntési csomópontéval) éppen ezt fejezi ki. A modell szimulációjakor, 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 járműoszlopból is ugyanabba a sávba sorolódnak be a járművek. A 8. ábra idődiagramon mutat be két eltérő lefutást.



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

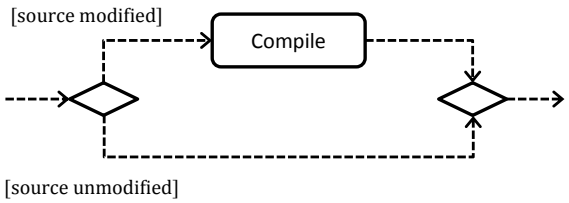


8. ábra. Elágazás és merge (besorolódás) kétféle lefutása idődiagramon

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 hajtódik végre; ilyet akkor használunk, ha egy tevékenység végrehajtása opcionális, vagy egy döntés valamilyen 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 a 9. ábra.

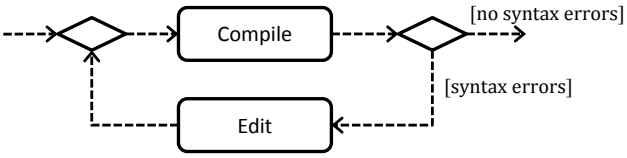


9. ábra. Elágazás üres ággal

2.5. Ciklus

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

Példa. A 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.



10. ábra. Ciklus grafikus szintaxisa

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

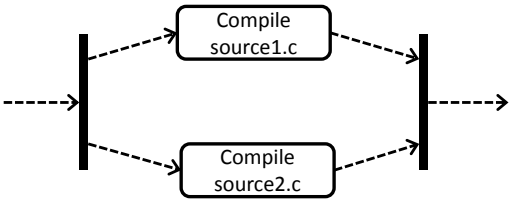
Nem nehéz végiggondolni, hogy ciklikus folyamatok futásakor ezek a vezérlési csomópontok többször is érinthetőek. Próbáljuk ezt a fenti példán a tokenes kézi szimuláció módszerével kipróbálni!

2.6. Konkurens viselkedés

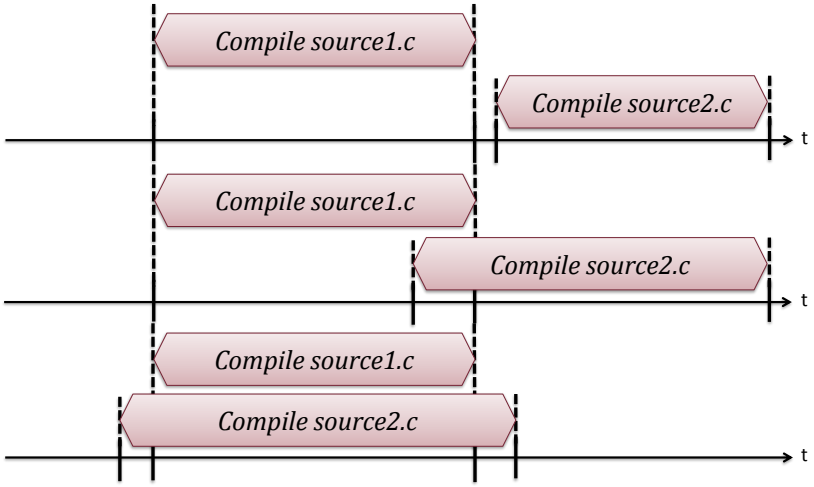
Előfordulhat olyan folyamat, amelyben nincs előírva két tevékenység (vagy részfolyamat) egymáshoz képesti sorrendje, csak hogy mindkettőnek meg kell történnie. A szóban forgó két tevékenység közül történhet az egyik a másik előtt, vagy fordítva; sőt, futhatnak (részben) egyszerre is. Ilyen viselkedést 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. A 11. ábra olyan folyamatot mutat be, ahol két forrásfájlt 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 lépésben a fork csomópont hatására *megkettőzzük* a tokent: az egyik token a felső, a másik az alsó kimenő vezérlési élre kerül. A továbbiakban szabadon választhatóan akármelyik token léptethető. Például elkezdheti először a felső token a tevékenységet, majd befejezheti, mielőtt az alsó elkezdené és befejezni 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 akármilyen sorrendben befejezhetik a saját tevékenységeiket. Számos lehetőség van; ám végül így vagy úgy, de eljutunk abba a helyzetbe, amikor mindkét token a join csomópont egy-egy bemenő vezérlési élén van. Ekkor egyetlen lépésként a két tokent újra *összeolvasztjuk*, és egyetlen tokent helyezünk a join kimenő élére. Néhány lehetséges lefutás látható a 12. ábrán.



11. ábra. Fork és join grafikus szintaxisa



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 élére 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.

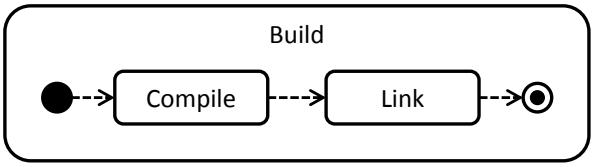
Definíció. A *fork csomópont* olyan csomópont a folyamatban, amely a belé érkező egyetlen vezérlési él hatására a belőle kiinduló összes *párhuzamnos folyamot* (vezérlési élet) kiválasztja végrehajtásra.

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.

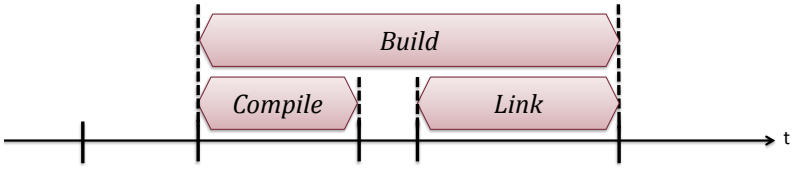
2.7. Teljes folyamatok

Eddig csak a folyamatok építőelemeivel foglalkoztunk; most megnézzük, hogy lehet a segítségükkel egy egész folyamatot leírni az elejétől 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 a 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 élre mozgatunk egy tokent; ez jelképezi a *Build* folyamat elindulását. Ezek után a szekvencia a már megismert módon szimulálható, amíg végül a *Link* tevékenységből kilépő vezérlési élre nem kerül a token. Ekkor a folyamat befejeződhet, amelyet úgy jelzünk, ha a célcsomópontba vezető élről a célcsomópontba mozgatjuk és egyúttal felszedjük a tokent. A lefutás idődiagramját a 14. ábra mutatja; jól látható, hogy az egyes tevékenységek végrehajtása teljes egészében a folyamat futása alatt zajlik.



13. ábra. Flow begin és flow end grafikus szintaxisa



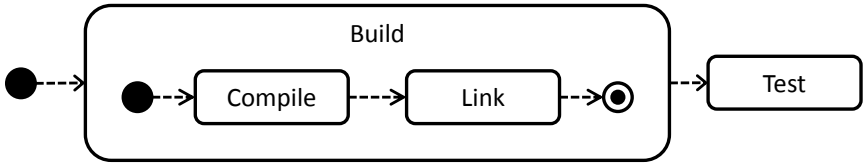
14. ábra. Teljes folyamat idődiagramja

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 kimenete van. A *cél (flow end) csomópont* a folyamat befejezését jelentő elem, melynek pontosan egy bemenete van.

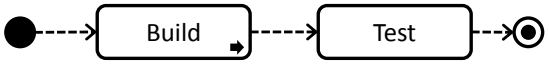
2.8. Hierarchikus folyamatmodellek

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

Példa.
A 15. ábra az alfolyamattá részletezett tevékenység használatát mutatja be, míg a 16. ábra az előzővel azonos jelentésű folyamatot épít fel úgy, hogy a folyamat első tevékenysége hivatkozza (hívja) a 13. ábrán látható, korábban definiált folyamatot.



15. ábra. Hierarchia grafikus szintaxisa



16. ábra. Hívás grafikus szintaxisa

2.9. Folyamatpéldányok

Példa. Vegyük példának a 13. ábrán látható folyamatmodellt, és szimuláljuk. Ahogy a 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 tokent rendelünk, amelyik a példány pillanatnyi állapotát jellemzi; ezután az összes tokent 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. egyesítenek.

A *Teljesítménymodellezés* 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.

3. Folyamatmodellek felhasználása

3.1. Programok vezérlési folyama

3.1.1. 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*) nevezzük.

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

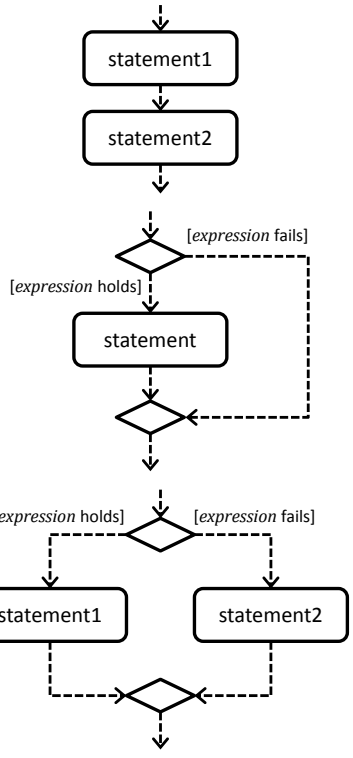
Megjegyzés. Bár a kezdő programozók általában az imperatív programokkal kezdik tanulmányaikat, léteznek ettől eltérő programozási paradigmák is, ahol a program a végrehajtási lépéssort nem határozza meg követlenül. Egyetemünkön a *Deklaratív programozás* című tárgy foglalkozik a funkcionális, ill. a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

3.1.2. Leképzés

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

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



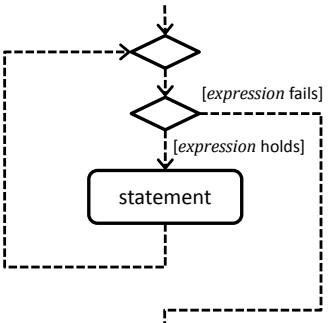

```
if (<expression>)
  <statement>
```



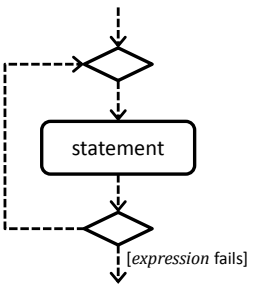
```
if (<expression>)
  <statement1>
else
  <statement2>
```

3.1.3. Összetett példa ábrázolása

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

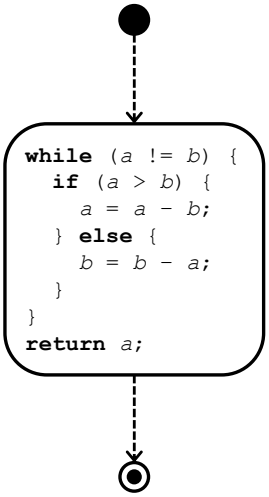



```
do
  <statement>
while (<expression>)
```

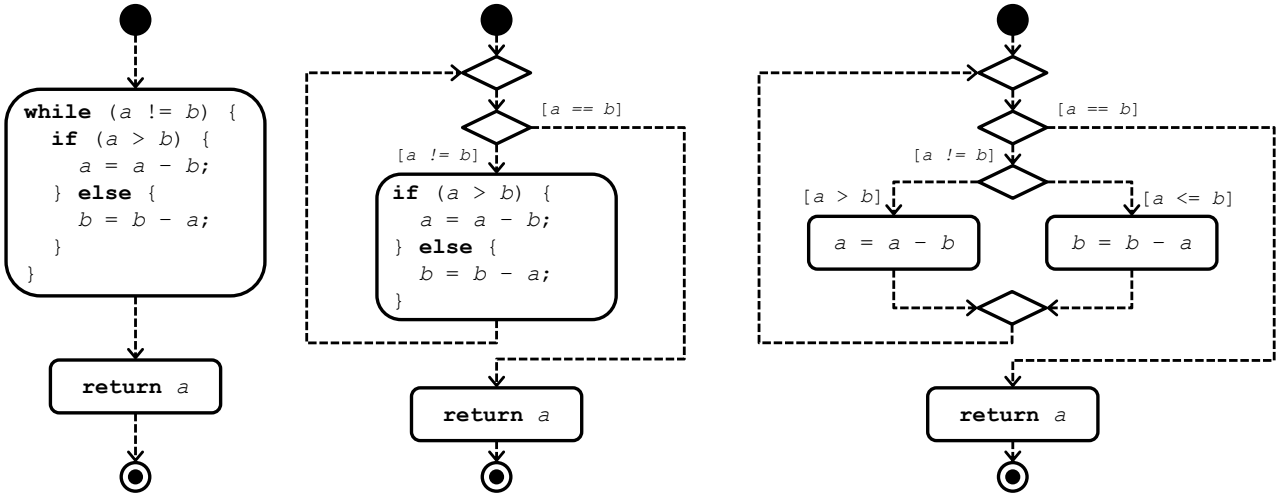


Nézzünk meg egy összetettebb példát!

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



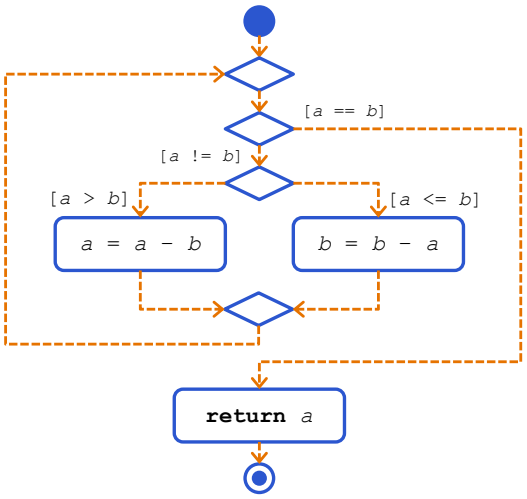
Lépésenként átalakítva:



3.1.4. 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.



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).

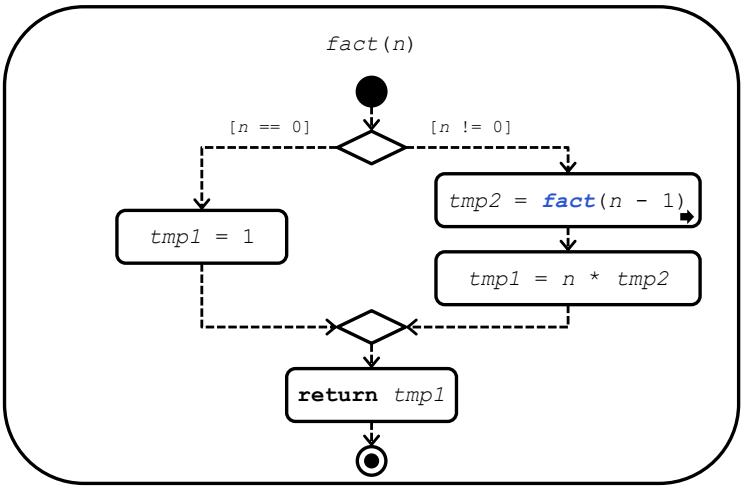
3.1.5. Példa: $n!$ meghatározása

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 kódot 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:

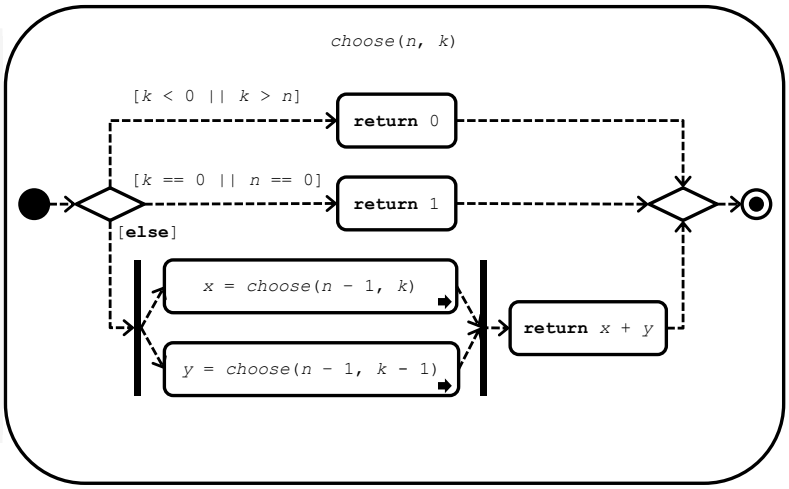
```
int fact(int n) {
  int tmp1;
  if (n == 0) {
    tmp1 = 1;
  } else {
    int tmp2 = fact(n - 1);
    tmp1 = n * tmp2;
  }
  return tmp1;
}
```



3.1.6. 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(n - 1, k);
    int y = spawn choose(n - 1, k - 1);
    sync;
    return x + y;
  }
}
```



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, ..., P_n jólstrukturált blokkok szekvenciája (egyszerű vezérlési éllel egymás után kötve őket);
- „*fork-join* kapcsolás”: a P_1, P_2, ..., P_n jólstrukturált blokkok egy n ágú *fork* és egy n ágú *join* közé zárva;
- „*decision-merge* kapcsolás”: a P_1, P_2, ..., 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 kizárni 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 kizárólag jólstrukturált blokkokat lehet létrehozni és más jólstrukturált blokkokból felépíteni. Ilyen nyelv pl. a BPEL (alap jelkészlete), vagy a programozásoktatásból ismerős Nassi-Shneiderman-féle *struktogram*. Ezeken a nyelveken a megkötések miatt bizonyos folyamatokat csak körülményesebben 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 kerülendőek. Megjegyzendő, hogy ritkán, de olyan eset is van, ahol épp pl. a `return` használata tesz egy mély vezérlési strukturát egyszerűbbé.

4. Kitekintés*

4.1. Technológiák

4.1.1. Adatelemzés

Az Airbnb cég Airflow eszköze[1] 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 Kepler[2] és a Taverna[3].

4.1.2. Üzleti folyamatmodellek

Az üzleti folyamatok modellezésére használt szoftverek manapság tipikusan a BPMN 2.0 szabványt valósítják meg.[4] Ilyen eszközök pédául a jBPM[5], a Bonita BPM[6], Camunda[7] és az Eclipse Stardust[8].

[1] `https://github.com/airbnb/airflow`
[2] `https://kepler-project.org/`
[3] `http://taverna.incubator.apache.org/`
[4] `https://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/`

Hivatkozások

Tárgymutató