

Állapotalapú modellezés

Hibatűrő Rendszerek Kutatócsoport

2017

Tartalomjegyzék

| | | | |
|--------------------------------------|----------|--|-----------|
| 1. Egyszerű állapotgépek | 1 | 3.2. Interfészváltozók | 9 |
| 1.1. Állapottér | 1 | 4. Időzítés | 10 |
| 1.2. Állapotátmenet, esemény | 2 | 5. Ortogonális dekompozíció | 10 |
| 1.3. Végrehajtási szekvencia | 4 | 5.1. Állapotgépek szorzata | 10 |
| 2. Hierarchia | 4 | 5.2. Ortogonális állapot | 12 |
| 2.1. Összetett állapot, állapotrégió | 5 | 6. A végső modell | 15 |
| 2.2. Több szintű állapothierarchia | 6 | 7. Kooperáló állapotgépek szinkronizációja* | 16 |
| 3. Változók és őrfeltételek | 6 | Irodalomjegyzék | 18 |
| 3.1. Belső változók | 7 | | |

Bevezetés

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.

1. Egyszerű állapotgépek

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

1.1. Állapottér

Ehhez első lépésként meg kell határozni a rendszer *állapottérét*. 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. *Kizárólagosság.* Minden időpontban az állapottér legfeljebb egy eleme jellemzi a rendszert.

¹<http://statecharts.org>

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.

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.

1.2. Állapotátmenet, esemény

A rendszer időbeli viselkedésében kulcsfontosságú, hogy a pillanatnyi állapot hogyan változik az idővel. Bizonyos mérnöki diszciplínákban ez a változás folytonos függvénnyel jellemezhető (ilyen rendszerekkel a Rendszerelmélet című tárgy foglalkozik részletesebben). Például egy repülő állapota lehet a tengerszint feletti magasság, amely egy időben folytonosan változó mennyiség. Azonban az informatikai gyakorlatban a *diszkrét állapottereknek* van kiemelt jelentősége, ahol nem létezik folytonos átmenet az állapotok között, tehát a rendszer pillanatnyi állapota mindaddig 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 váltson 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 spon-tán 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 segítő folyamatosan naplózza egy külső fekete dobozba. Ennek megfelelően legyenek a rendszer output eseményei a következők:

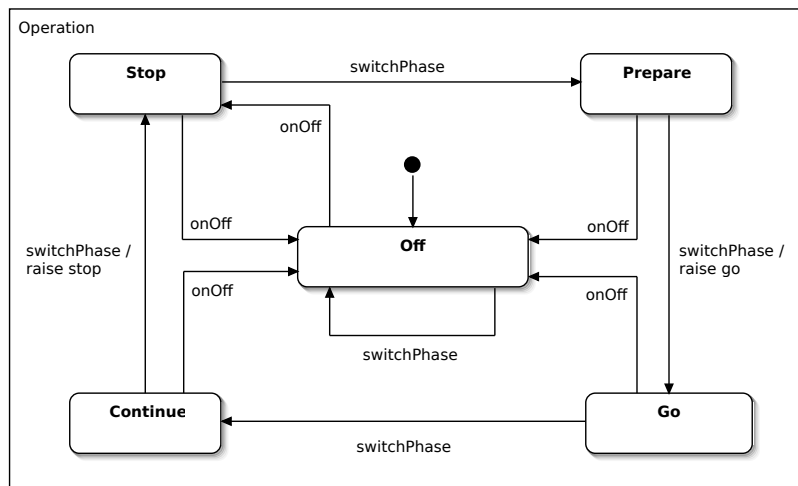
- Stop: a rendszer sárga jelzésből vörös jelzésbe váltott
- Go: a rendszer zöld jelzésbe váltott

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 az 1. ábra állapotgépével.

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 nyilak a megfelelő állapotok közötti tranzíciókat jelképezik. A nyilakra írt címkék a tranzíciót



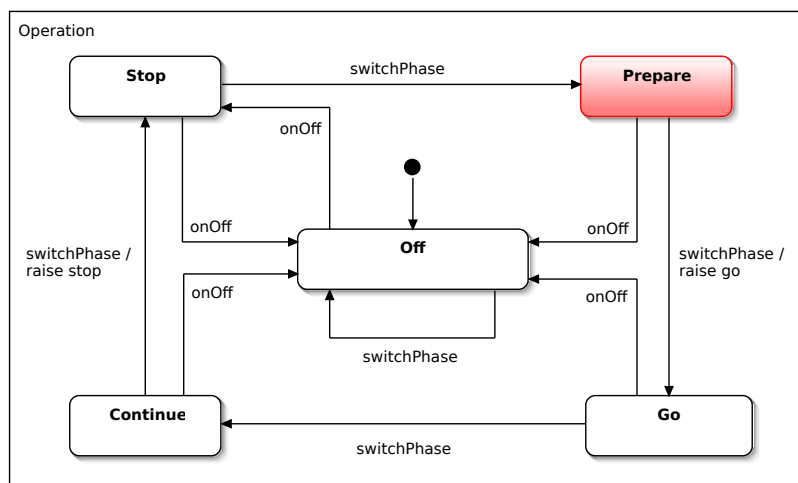
1. ábra. Jelzőlámpa egyszerű állapotgépe

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 téglalap jelöl (*állapot-csomópont*).
- *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.



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. 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 *holtponmentes*, azaz nem tartalmaz olyan állapotot, amelyből nem vezet ki tranzíció.

1.3. Végrehajtási szekvencia

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

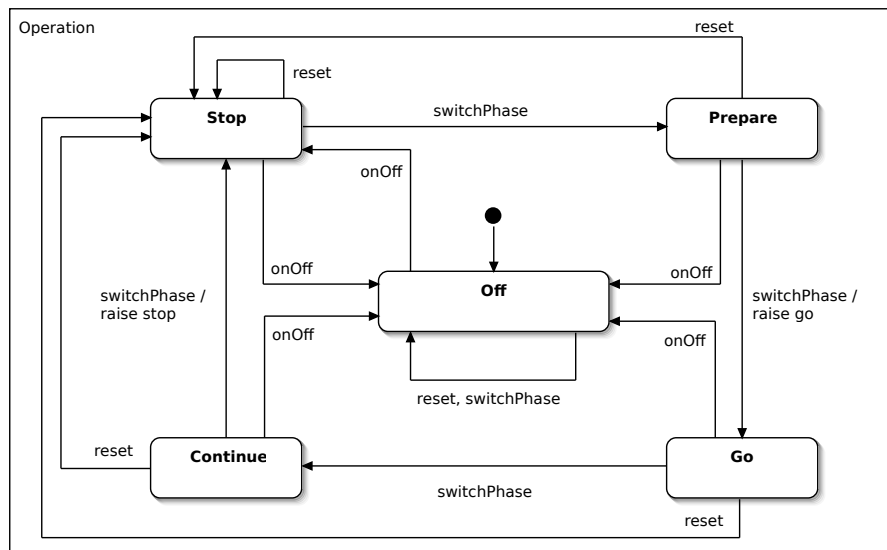
$$s_0 \xrightarrow{i_0/o_0} s_1 \xrightarrow{i_1/o_1} \dots$$

(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 módot ad az állapotgép ellenőrzésére. A Yakindu beépített szimulátora erre lehetőséget biztosít.

2. Hierarchia



3. ábra. Jelzőlámpa állapotgépe reset eseménnyel 🦉

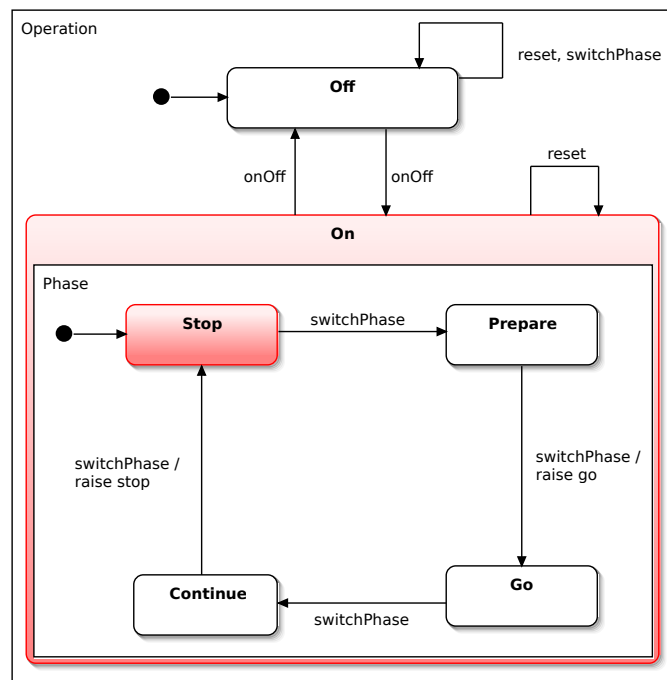
Példa. Egészítsük ki a fenti modellt egy reset bemeneti eseménnyel, melynek hatására bekapcsolt állapotban a jelzőlámpa alaphelyzetbe (Stop állapotba) áll. Az eddig megismert eszközökkel elkészített modellt szemlélteti a 3. ábra.

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 jelezni egy *összetett állapot* bevezetésével, amely a négy állapot közös tulajdonságait és viselkedését általánosítja.

2.1. Összetett állapot, állapot régió

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. ábra szemlélteti az **On** összetett állapot bevezetésével kapott modellt.



4. ábra. Hierarchikus állapot pillanatnyi állapotként

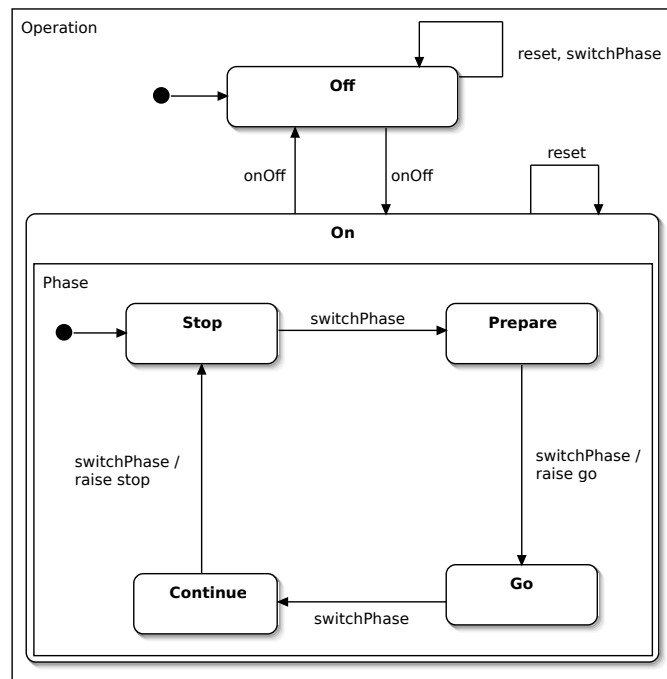
A teljes állapotgépet az **Operation**, míg az összetett állapot belsejét a **Phase** régió tartalmazza. A **Phase** régió kezdőállapota a **Stop** állapot, így a régióba való belépéskor ez az állapot lesz a rendszer pillanatnyi állapota. A **reset** esemény az **On** állapotból **On** állapotba vezet,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

Példa. A jelzőlámpa állapotkonfigurációi:

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



5. ábra. Jelzőlámpa állapotgépe összetett állapottal

Tartalmazó és tartalmazott állapot között tehát nem érvényesül a kizárólagosság. 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 kizárólagosságot. Általános szabály, hogy egy állapottér vagy az összetett állapotot, vagy annak összes részállapotát tartalmazza, de nem mindkét 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.

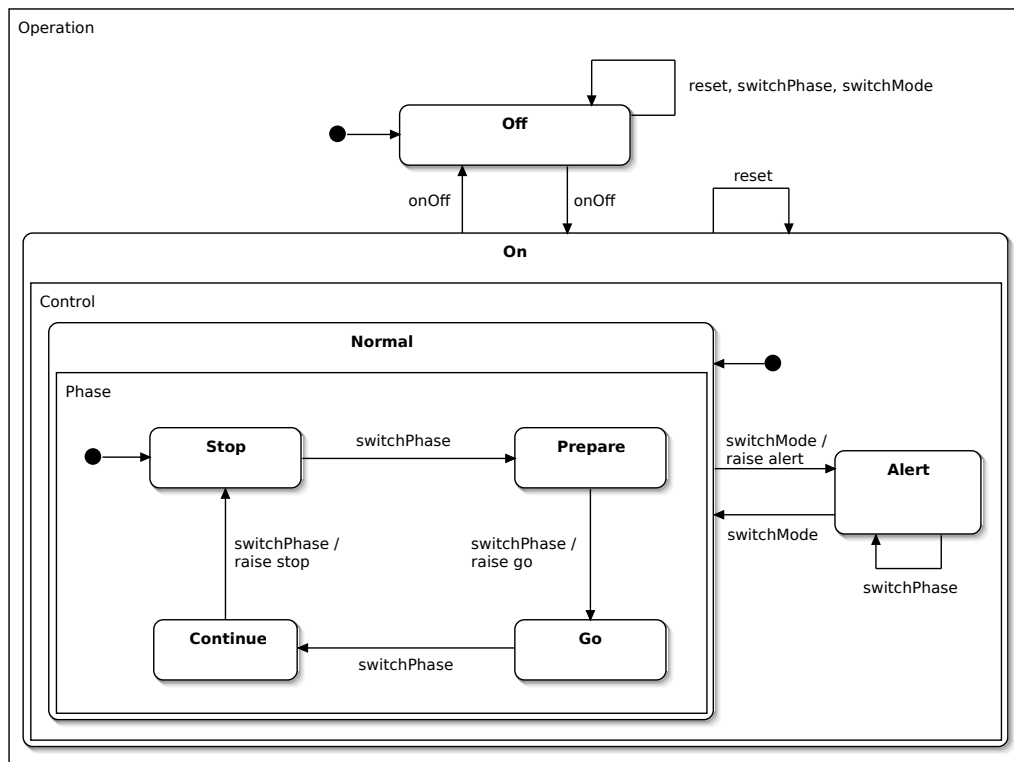
2.2. Többszintű állapothierarchia

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

Példa. A 6. ábrán szemléltetett modell az On állapotot tovább bővíti egy Alert állapottal, mely a jelző sárgán villogó, figyelemztető állapotát modellezi. 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.

3. Változók és őrfeltételek

Finomítsuk az Alert állapotot LightOn és LightOff alállapottakkal, melyek fél másodpercenként váltakozva a sárga fény villogását modellezzik! A villogás modellezéséhez feltételezzünk egy tick bejövő eseményt, amely a specifikáció szerint egy 8 Hz frekvenciájú órajel, tehát egyenletes ütemben, másodpercenként nyolcszor jelez.



6. ábra. Jelzőlámpa állapotgépe többszintű összetett állapottal

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.

Egy lehetséges megvalósítást szemléltet a 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, mindkét összetett állapot száz aláállapotot tartalmazna. Látható tehát, hogy ez nem lesz jó modellezési gyakorlat, hiszen az így bemutatott modell elkészítése nagy erőfeszítést igényel, sok hiba-lehetőséget rejt és nehezen átlátható; valamint a számszerű paraméterek megváltozása esetén jelentős módosítást igényel.

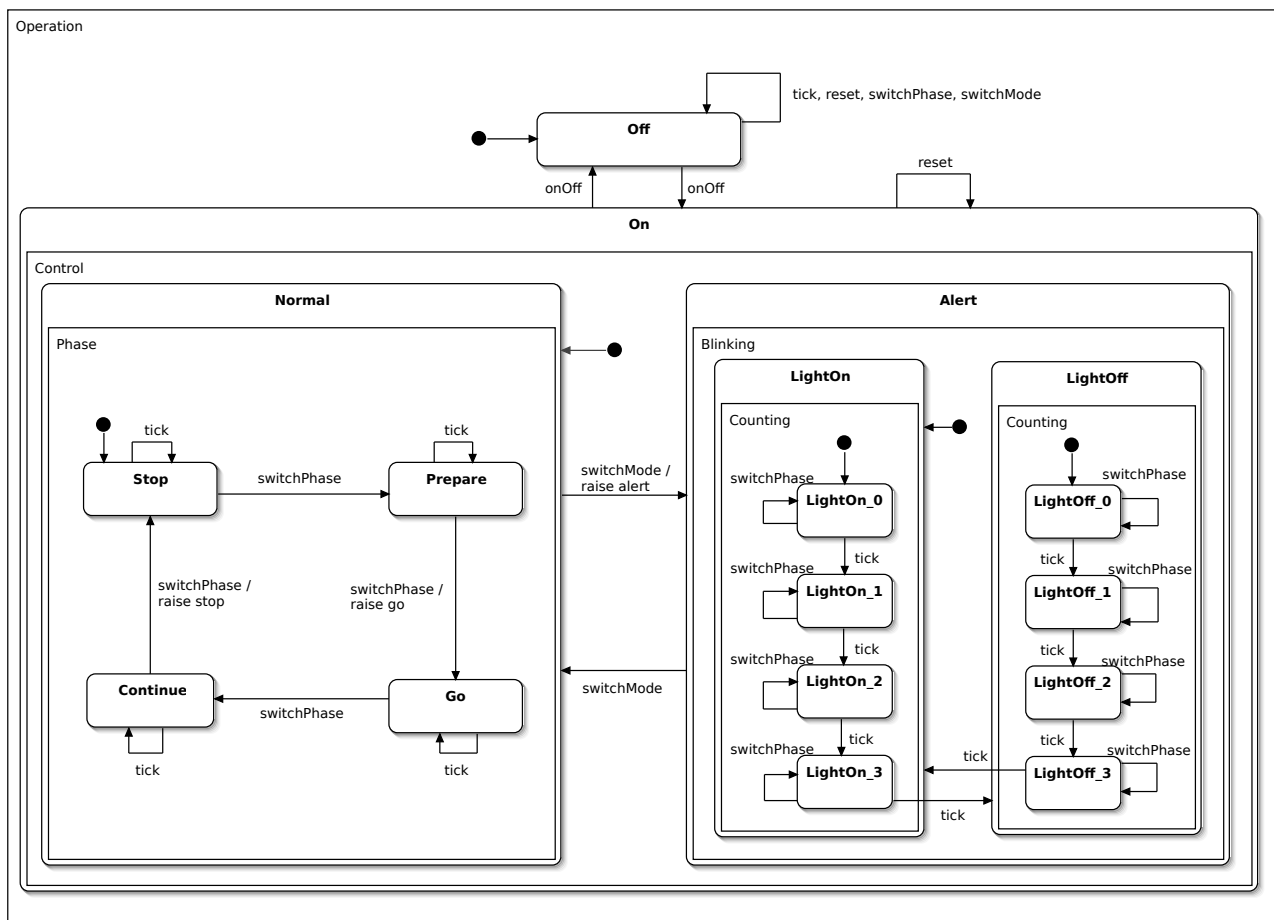
3.1. Belső változók

A probléma megoldható *változók* alkalmazásával. A változó típusal 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 interfészdefinícióját kiegészítjük a **counter** egész típusú változóval:

```
var counter : integer
```

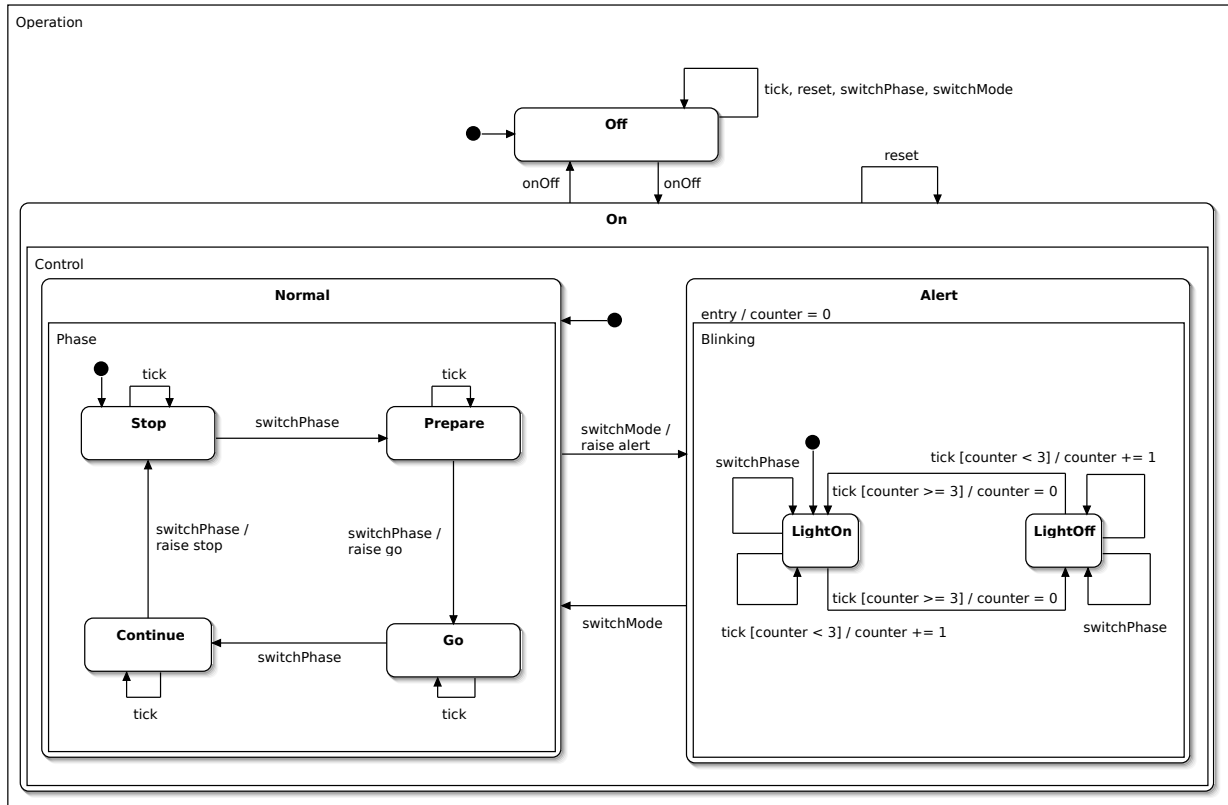
A változó értékét *utasítással* lehet módosítani, amely – az eseménykiváltáshoz hasonlóan – tranzícióhoz kapcsolható *akció*. Akció ezen felül kapcsolható állapothoz is az **entry** és **exit** triggerek segítségével, melyek az állapotba való belépéskor, illetve az állapotból történő kilépéskor aktiválódnak.



7. ábra. Villogó jelzés külső órával ☠

Azt, hogy a változó aktuális értéke befolyással lehessen a vezérlésre, a tranzíciókra felírt őrfeltételekkel lehet megvalósítani. Az őrfeltétel biztosítja, hogy a tranzíció csak akkor tüzelhessen, ha az őrfeltételbe felírt logikai kifejezést az aktuális állapot változóértékelése kielégíti.

Példa. A tick események számlálása változóval megvalósítható a 8. ábra állapotgépe szerint. Figyeljük meg, hogy a változó értékét szögletes zárójelekbe tett őrfeltételek használják.



8. ábra. Villogó jelzés számlálással

Példa. A fenti rendszer egy végrehajtásiz szekvencia-részlete:

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

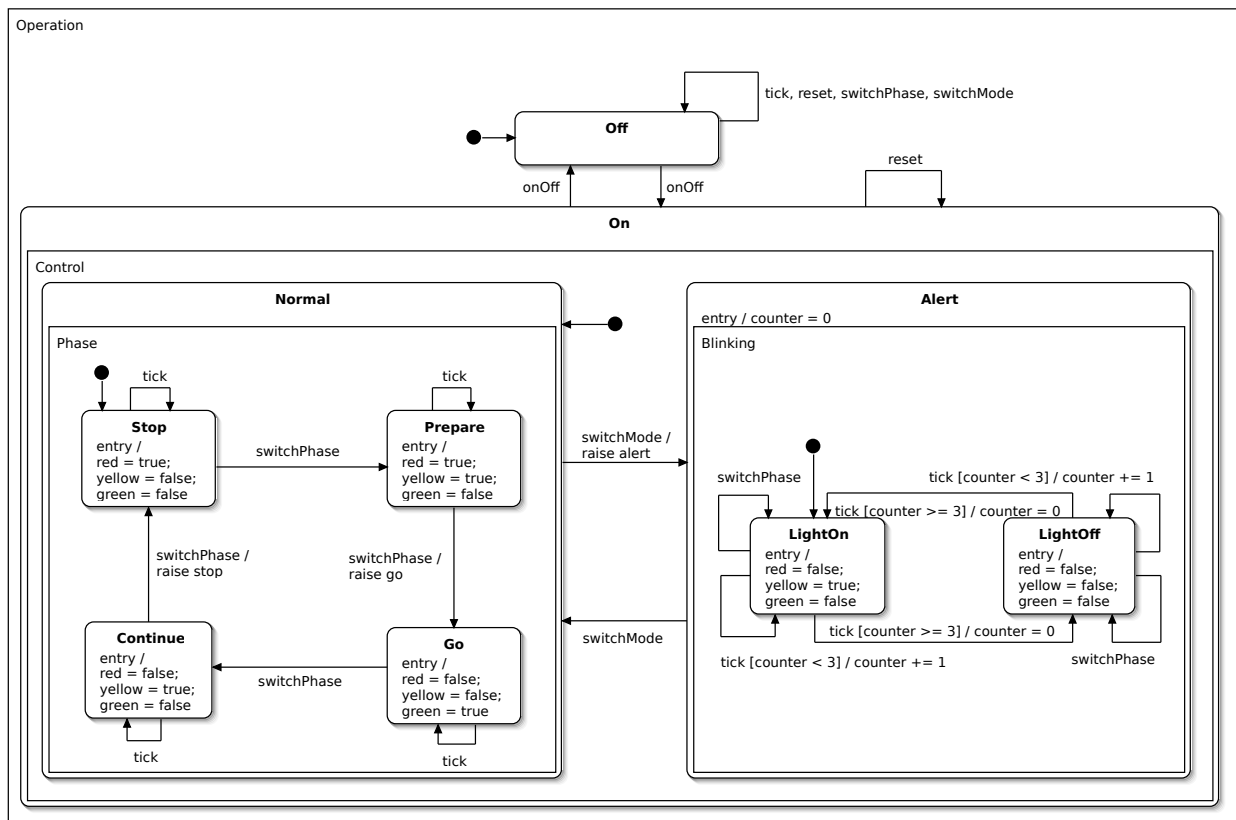
Feladat. Hogyan módosul őrfeltételek jelenlétében a *determinizmus* definíciója?

3.2. Interfészváltozók

A vörös, sárga ill. zöld színű fények állapotának nyilvántartására felvesszük **red**, **yellow** és **green** logikai változókat. A korábban bevezetett **counter** változóval ellentétben ezek a változók nem csak az állapotgéppel leírt vezérlő belső működésében játszanak szerepet; úgy tekintjük, hogy a különböző színű fények villanygőit 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.

A bővített állapotgépet a 9. ábra szemlélteti.



9. ábra. Fények állapotának modellezése változókkal

4. Időzítés

A modell időbeli viselkedését eddig egy külső óra által szolgáltatott, megszabott frekvenciájú óra-jelet 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 10. ábra szemlélteti.

5. Ortogonális dekompozíció

Bővítsük a modellt járműérzékelő² funkcióval!

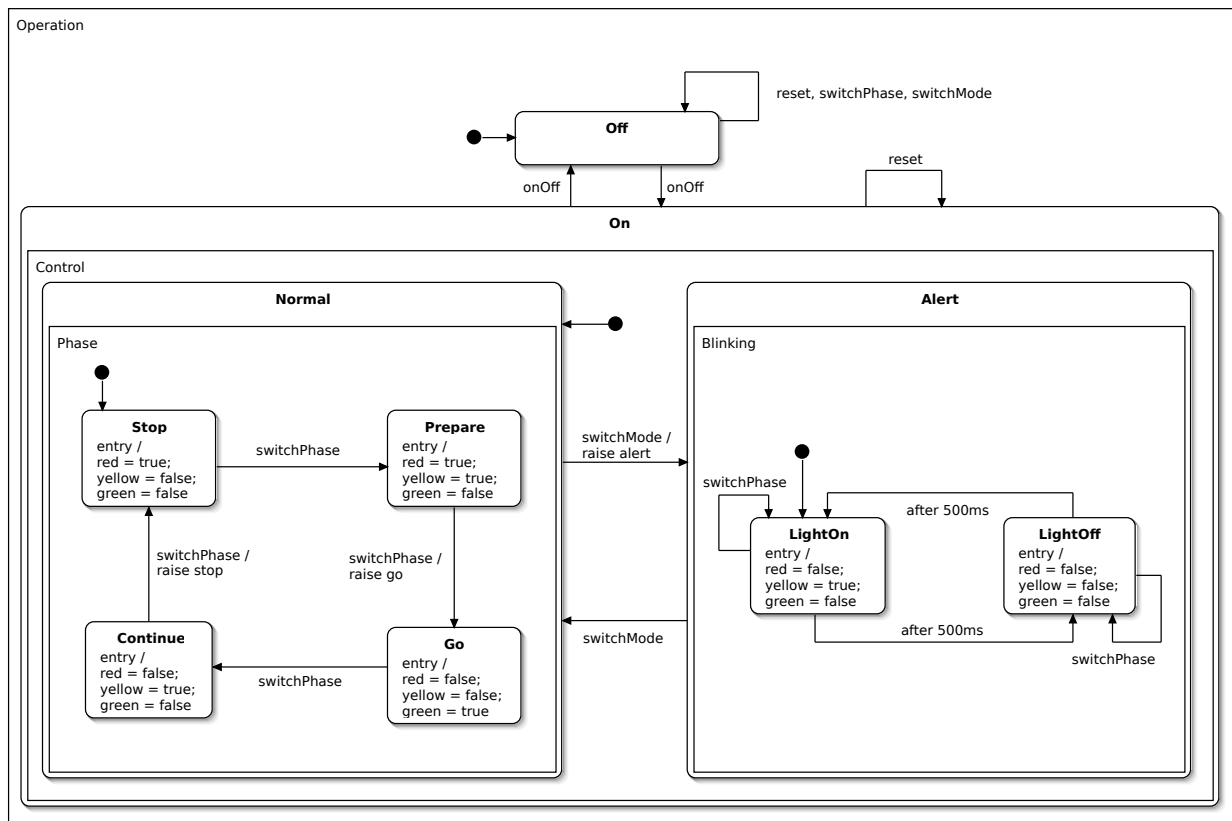
Tipp. A jelzőlámpa (bekapcsolt állapotban) periodikus **trafficPresent** és **trafficAbsent** bemeneti események fogadásával értesül arról, hogy tartózkodik-e jármű a lámpa előtti útszakaszon. A jelző ezt az információt a **queue** logikai változóban tárolja (ennek értéke kezdetben **true**). A foglaltság 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 11. ábrán szemléltetett állapotgép valósítja meg.

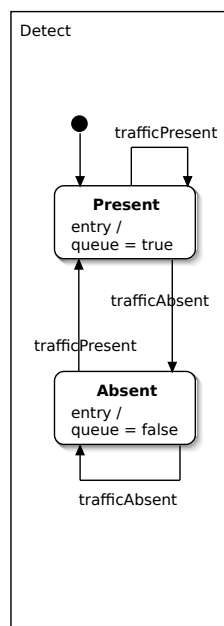
5.1. Állapotgépek szorzata

Gondoljuk végig, hogyan lehet a fenti **Detect** régió viselkedését kombinálni a már meglévő modellel! Mivel a foglaltságérzékelés **On** állapotba lépve kezd működni, így elég a meglévő modell **Control** régióra

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



10. ábra. Villogó jelzés időzítéssel



11. ábra. Járműérzékelő funkció állapotgépe

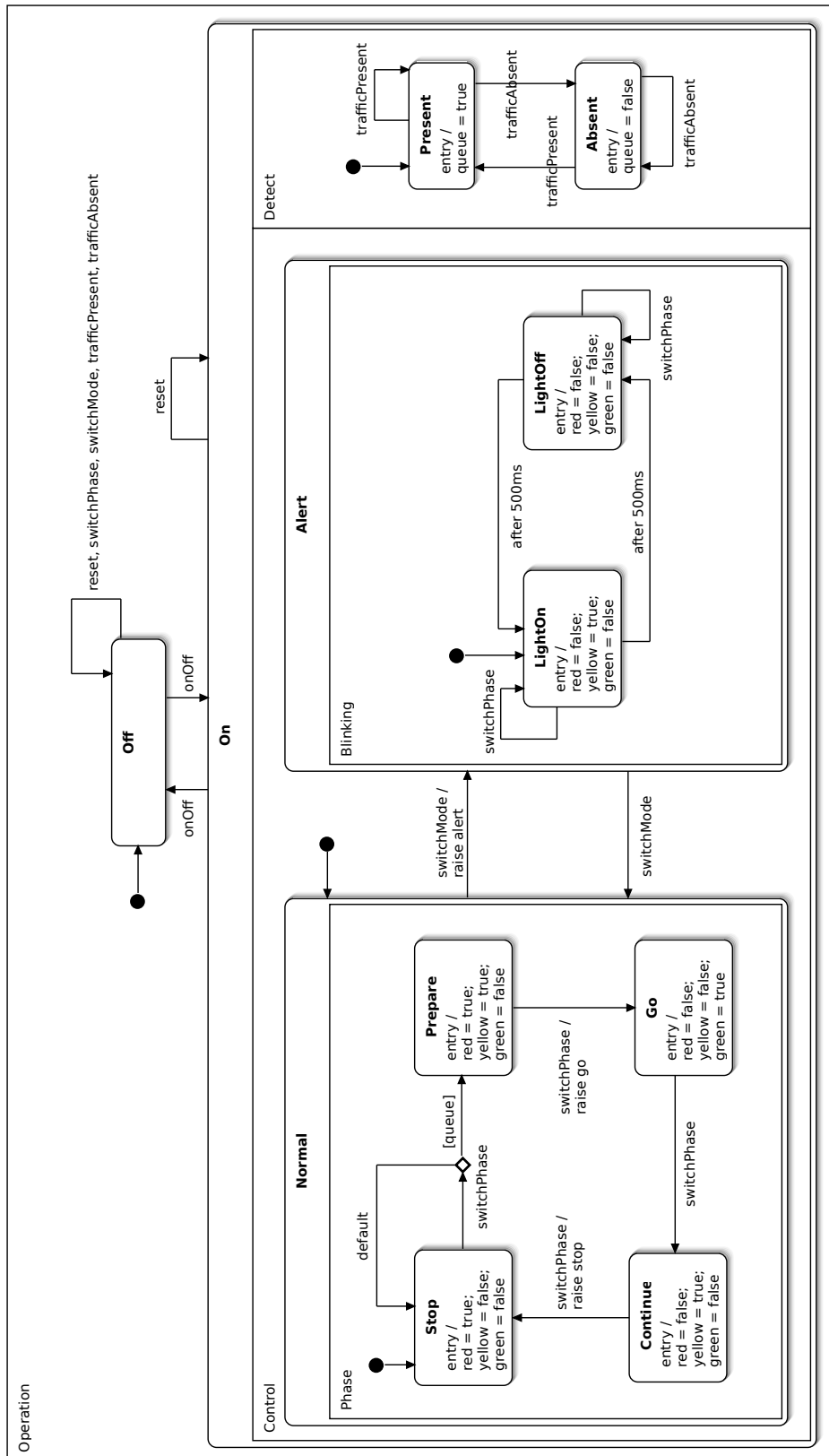
koncentrálni. Ekkor az eredeti modell pillanatnyi állapota **Stop**. Ahhoz, hogy a bővített modellben ilyenkor mind a **switchPhase**, mind a **trafficAbsent** és **trafficPresent** eseményeket megfelelően kezelni tudjuk, szükséges egy **Stop_Present** kezdőállapot és egy **Stop_Absent** állapot felvétele. 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 vizsgálánk, 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).

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.

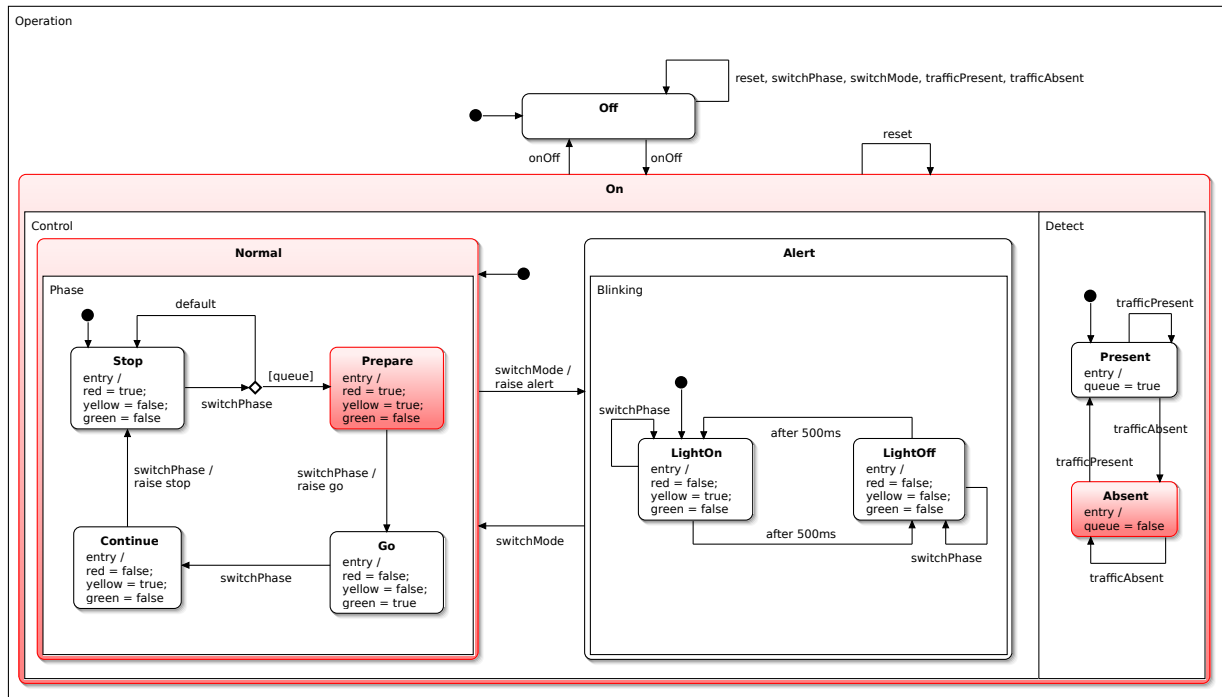
Az így elkészített állapotgépet a 12. ábra szemlélteti.



12. ábra. Járműérzékelős jelzőlámpa állapotgépe ortogonális állapottal

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 13. ábra látható állapotkonfigurációba kerül. Vegyük észre, hogy az aktív ortogonális állapot (On) mindkét ortogonális régiójában (Control, Detect) pontosan egy közvetlenül tartalmazott állapot aktív.



13. ábra. Ortogonális állapot aktuális állapotként

Ortogonalis 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 kizárólagosság).

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

$$\{\text{Off, Stop, Prepare, Go, Continue, LightOn, LightOff}\}$$

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

$$\{\text{Off, Present, Absent}\}$$

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 funkcionalitást?

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 kizárólag a **Detector** interfészt kell figyelembe venni, a többi interfész részleteivel nem kell megismerkedni, azok esetleges áttervezését nem kell nyomon követni. Speciális esetként megjelenik a kizárólag 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 interfészeknek lehetne azonos nevű eseménye ill. állapotváltozója, ezért Yakinuban mindig az interfész megadásával kell hivatkozni a változókra és eseményekre. A módosított modellt szemlélteti a 14. ábra.

7. Kooperáló állapotgépek szinkronizációja*

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

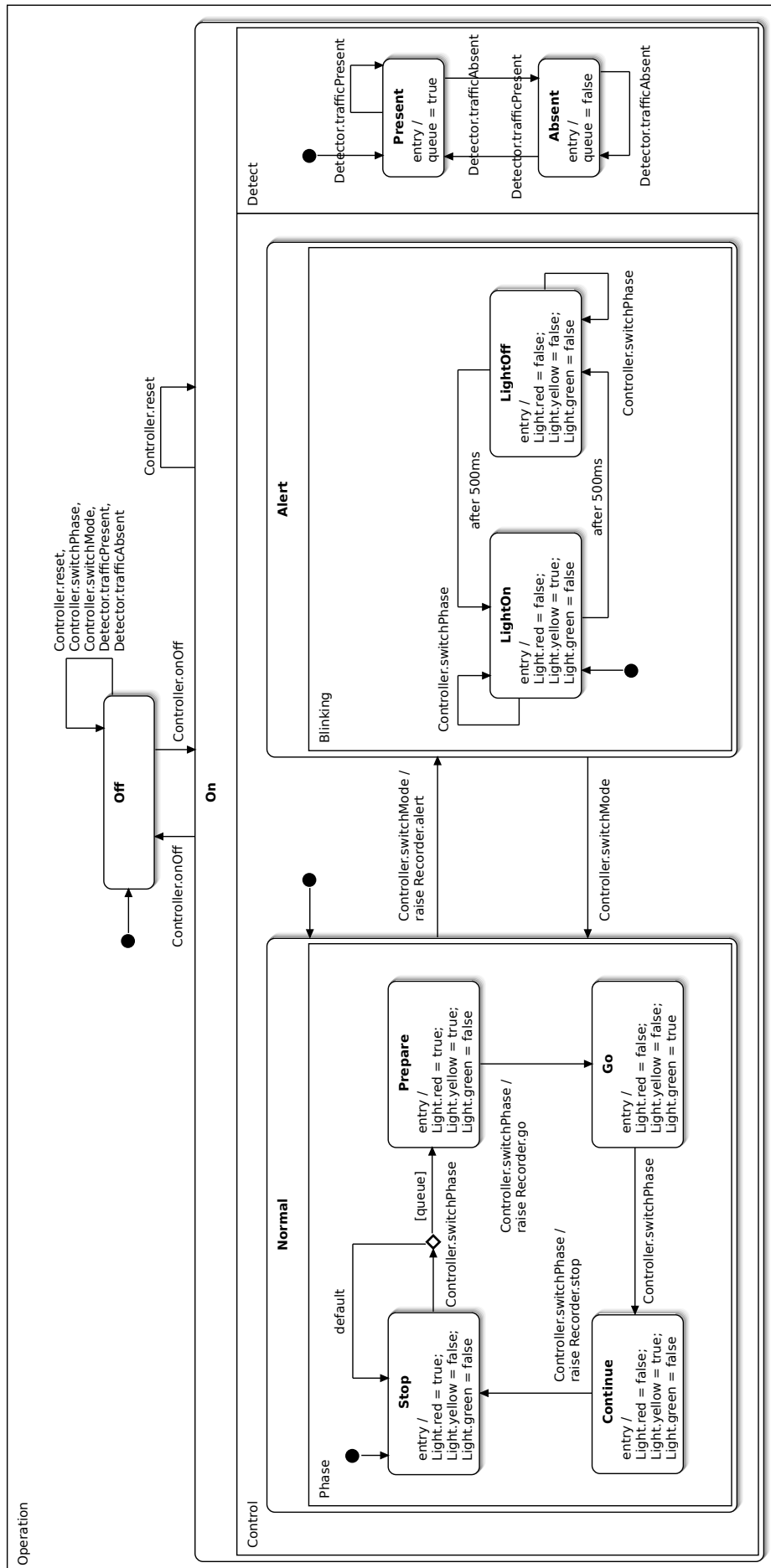
A szinkronizációval leírható, hogy a két állapotgépben megjelölt állapotátmenetek valójában a teljes rendszer egyetlen (összetett) állapotátmenetének vetületei, külön-külön nem, csak egyszerre végrehajthatók. A címke az összetett állapotátmenetre utal, így több helyen is előfordulhat ugyanaz a szinkronizációs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

Felhasznált irodalom

- Az állapottérkép modellezési módszer kidolgozása [3, 4]
- Az állapottérkép modell UML-ben [2]
- Állapottérkép modellek értelmezése (modellszemantika) [7, 1, 5]
- Állapottérkép alapú forráskód generálás [11]
- Pintér Gergely, *Model Based Program Synthesis and Runtime Error Detection for Dependable Embedded Systems* [10]
- UML állapottérképek használata biztonságkritikus rendszerekben [6, 8]
- Pap Zsigmond, *Biztonságossági kritériumok ellenőrzése UML környezetben* [9]



14. ábra. A jelzőlámpa modellje interfészekkel

Hivatkozások

- [1] Jori Dubrovin – Tommi A. Junttila: Symbolic model checking of hierarchical UML state machines. In Jonathan Billington – Zhenhua Duan – Maciej Koutny (szerk.): *8th International Conference on Application of Concurrency to System Design (ACSD 2008), Xi'an, China, June 23-27, 2008* (konferenciaanyag). 2008, IEEE, 108–117. p. ISBN 978-1-4244-1838-1.
URL <http://dx.doi.org/10.1109/ACSD.2008.4574602>.
- [2] Object Management Group: Information technology – Object Management Group Unified Modeling Language (OMG UML) – part 2: Superstructure. ISO/IEC 19505-2:2012. Jelentés, 2012, Object Management Group. URL <http://www.omg.org/spec/UML/ISO/19505-2/PDF/>.
- [3] David Harel: Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8. évf. (1987) 3. sz., 231–274. p. URL [http://dx.doi.org/10.1016/0167-6423\(87\)90035-9](http://dx.doi.org/10.1016/0167-6423(87)90035-9).
- [4] David Harel: Statecharts in the making: a personal account. In Barbara G. Ryder – Brent Hailpern (szerk.): *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007* (konferenciaanyag). 2007, ACM, 1–43. p. URL <http://doi.acm.org/10.1145/1238844.1238849>.
- [5] David Harel – Amir Pnueli – Jeanette P. Schmidt – Rivi Sherman: On the formal semantics of statecharts (extended abstract). In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987* (konferenciaanyag). 1987, IEEE Computer Society, 54–64. p.
- [6] John C Knight – Colleen L DeJong – Matthew S Gobble – Luis G Nakano: Why are formal methods not used more widely? In *Fourth NASA formal methods workshop* (konferenciaanyag). 1997, Citeseer.
- [7] Diego Latella – István Majzik – Mieke Massink: Towards a formal operational semantics of UML statechart diagrams. In Paolo Ciancarini – Alessandro Fantechi – Roberto Gorrieri (szerk.): *Formal Methods for Open Object-Based Distributed Systems, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), February 15-18, 1999, Florence, Italy*, IFIP Conference Proceedings konferenciasorozat, 139. köt. 1999, Kluwer. ISBN 0-7923-8429-6.
- [8] C. R. Nobe – William E. Warner: Lessons learned from a trial application of requirements modeling using statecharts. In *Proceedings of the 2nd International Conference on Requirements Engineering, ICRE '96, Colorado Springs, Colorado, USA, April 15-18, 1996* (konferenciaanyag). 1996, IEEE Computer Society, 86–93. p. ISBN 0-8186-7252-8.
URL <http://dx.doi.org/10.1109/ICRE.1996.491433>.
- [9] Zsigmond Pap: *Biztonságossági kritériumok ellenőrzése UML környezetben*. PhD értekezés (Budapest University of Technology and Economics). 2006.
URL <https://repozitorium.omikk.bme.hu/handle/10890/595>.
- [10] Gergely Pintér: *Model based program synthesis and runtime error detection for dependable embedded systems*. PhD értekezés (Budapest University of Technology and Economics). 2007. URL <https://repozitorium.omikk.bme.hu/handle/10890/636>.
- [11] Miro Samek: Practical UML statecharts in C/C++. *Event-Driven Programming for Embedded Systems. Second Edition*. Newnes, 2008.

Tárgymutató

