C:\Users\szarnyasg\Downloads\bme_logo_nagy.eps

**Budapesti Műszaki és Gazdaságtudományi Egyetem**

Villamosmérnöki és Informatikai Kar

Hálózati interfész   
modell alapú generálása

Készítette

Mikitovics MárkKonzulens

Semeráth Oszkár

2017

Tartalomjegyzék

[Összefoglaló 4](#_Toc482556915)

[Abstract 5](#_Toc482556916)

[1. Bevezetés 6](#_Toc482556917)

[2. Előismeretek 7](#_Toc482556918)

[2.1. Yakindu Statechart tools [1] 7](#_Toc482556919)

[2.1.1. Kiberfizikai rendszer állapotgépekkel 7](#_Toc482556920)

[2.1.2. Osztályok generálása 8](#_Toc482556921)

[2.2. Message Queue Telemetry Transport (MQTT [2]) és Paho [3] 9](#_Toc482556922)

[2.2.1. MQTT protokollt használó osztály 9](#_Toc482556923)

[2.2.2. MQTT kompatibilis állapotgép futtatása 11](#_Toc482556924)

[3. Hálózati kommunikációt megvalósító állapotgépek 13](#_Toc482556925)

[3.1. A kiberfizikai rendszer komponensei 13](#_Toc482556926)

[3.2. Konfigurációs fájl elkészítése 16](#_Toc482556927)

[3.3. A kódgeneráló működése 18](#_Toc482556928)

[3.4. Tesztelés 20](#_Toc482556929)

[3.5. Tesztesetek bemutatása, a rendszer működés közben 20](#_Toc482556930)

[4. Irodalomjegyzék 24](#_Toc482556931)

[5. Függelék 25](#_Toc482556932)

Hallgatói nyilatkozat

Alulírott Mikitovics Márk, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2017.05.21.

Mikitovics Márk

# Összefoglaló

A szakdolgozat, vagy diplomaterv elkészítése minden egyetemi hallgató életében egy fontos mérföldkő. Lehetőséget ad arra, hogy az egyetemi évei során megtanultakat kamatoztassa és eredményeit szélesebb közönség előtt bemutassa, s mérnöki rátermettségét bizonyítsa. Fontos azonban, hogy a dolgozat elkészítésének folyamata számos csapdát is rejt magában. Rossz időgazdálkodás, hiányos szövegszerkesztési ismeretek, illetve a dolgozat készítéséhez nélkülözhetetlen „műfaji” szabályok ismeretének hiánya könnyen oda vezethetnek, hogy egy egyébként jelentős időbefektetéssel készült kiemelkedő szoftver is csak gyengébb minősítést kapjon a gyenge minőségű dolgozat miatt.

E dokumentum – amellett, hogy egy általános szerkesztési keretet ad a dolgozatodnak – összefoglalja a szakdolgozat/diplomaterv írás írott és íratlan szabályait. Összeszedjük a Word kezelésének legfontosabb részeit (címsorok, ábrák, irodalomjegyzék stb.), a dolgozat felépítésének általános tartalmi és szerkezeti irányelveit. Bár mindenkire igazítható sablon természetesen nem létezik, megadjuk azokat az általános arányokat, oldalszámokat, amelyek betartásával jó eséllyel készíthetsz egy színvonalas dolgozatot. A részletes és pontokba szedett elvárás-lista nem csupán a dolgozat írásakor, de akár más dolgozatok értékelésekor is kiváló támpontként szolgálhat.

Az itt átadott ismeretek és szemléletmód nem csupán az aktuális feladatod leküzdésében segíthet, de hosszútávon is számos praktikus fogással bővítheti a szövegszerkesztési és dokumentumkészítési eszköztáradat.

# Abstract

English translation of the abstract of the thesis work. This summarises the content of the thesis in 0.5–1 pages and is uploaded to the Thesis Work Portal as well.

# Bevezetés

Jelen sablon célja, hogy a Mérnök informatikus BSc/MSc, a villamosmérnök BSc/MSc, az gészségügyi mérnök MSc, az gazdaságinformatikus MSc képzések szakdolgozat és diplomaterv munkáihoz sablont nyújtson. A dokumentumot korábbi sablonokból kiindulva Kővári Bence és Szárnyas Gábor állította össze.

# Előismeretek

## Yakindu Statechart tools

A Yakindu Statechart tools [1] segítségével az általam megtervezett kiberfizikai rendszer állapotgépekkel való leírása volt a cél.

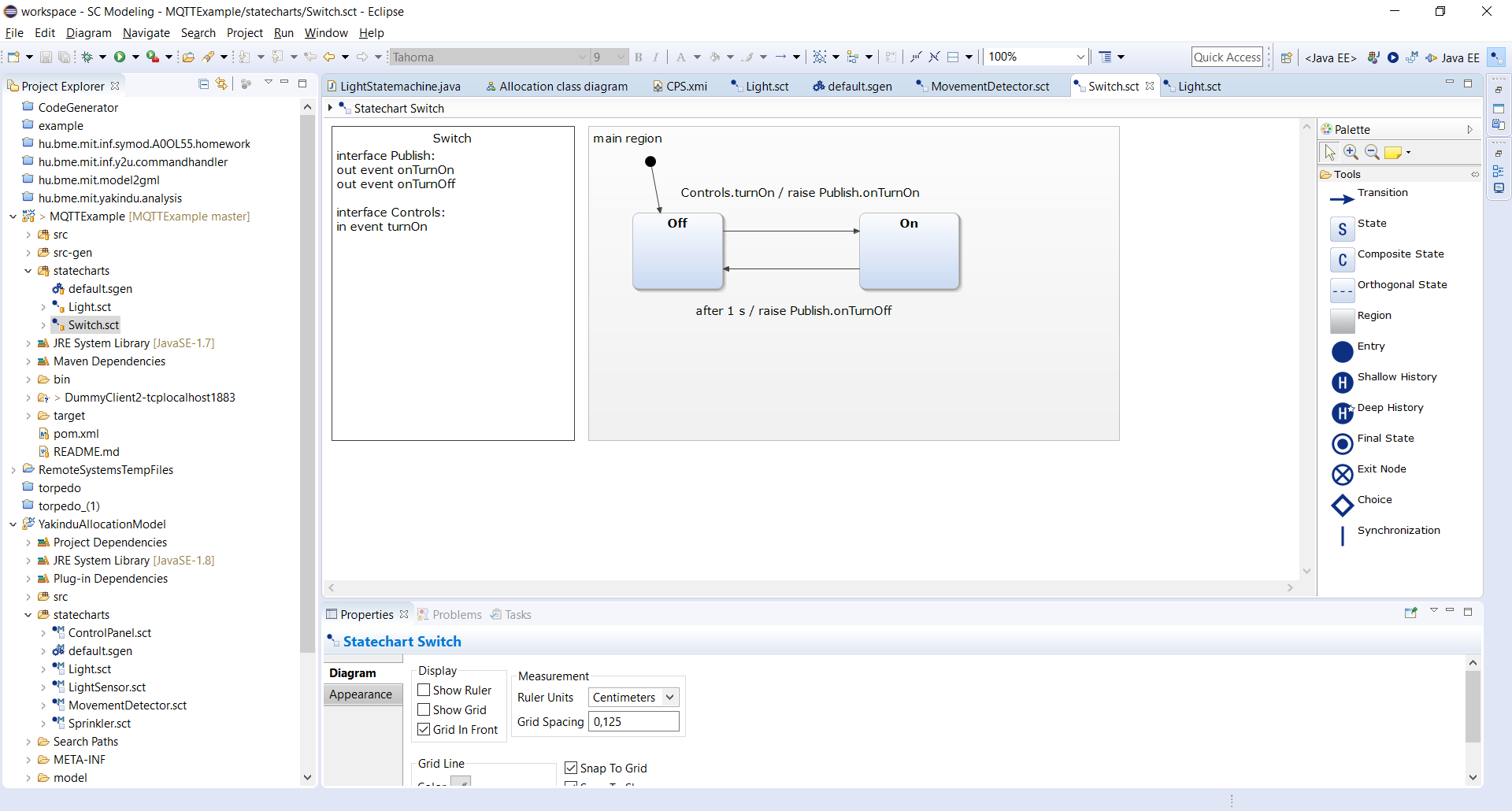
A feladatot az eszköz megismerésével kezdtem, egy kezdetleges példát készítettem el, amely rendelkezett a végső kiberfizikai rendszer minden szükséges funkcionalitásával. Ez olyan állapotgépek létrehozását követelte meg, amelyek képesek külső események hatására belső állapotot váltani, állapotváltásaik hatására kimenő eseményt generálni, illetve rendelkeznek belső időzítő mechanizmussal, amely szintén képes állapotváltásokat előidézni.

### Kiberfizikai rendszer állapotgépekkel

Erre a célra megterveztem egy olyan rendszert, amely két komponensből áll, egy kapcsolóból és egy lámpából. A kapcsoló alapesetben ki van kapcsolva. Külső esemény hatására bekapcsolt állapotba kerül, majd egy másodperc múlva automatikusan kikapcsolt állapotba jut. A lámpa szintén ki van kapcsolva alapértelmezetten, viszont ő minden állapotváltását külső esemény hatására végzi.

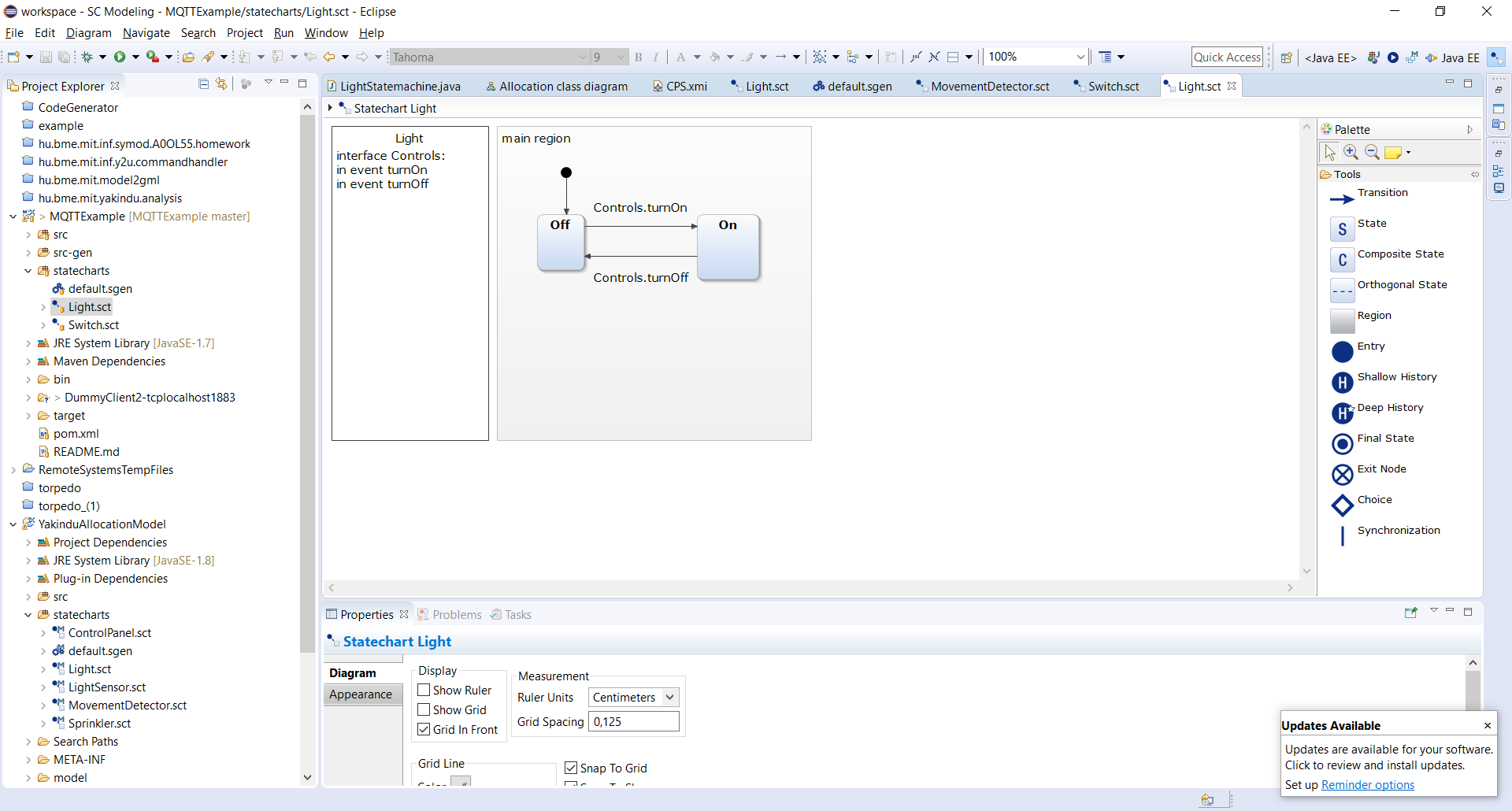
Általánosan elmondható, hogy minden állapotgép két interfésszel rendelkezik. A bejövő eseményeket a **Controls**, a kimenőket a **Publish** tartalmazza. A későbbiekben ezen események összekötésével lesznek képesek a komponensek a kommunikációra.

A kapcsoló megvalósítása az alábbi képen látható. A kiindulási pont az **Off** állapot, ebből lehetséges az eljutás az **On** állapotba, majd onnan vissza is. A **Controls** interfész **turnOn** eseményének hatására a **Publish** interfész **onTurnOn** eseménye kerül kiküldésre az **On** állapotba jutáskor. A visszatérés az **after 1 s** kifejezés hatására egy másodperc késleltetéssel történik, melynek hatására szintén a **Publish** interfésznek az **onTurnOff** eseménye eszkalálódik.



. ábra: A kapcsoló megvalósítása Yakindu segítségével

Ezt követően a lámpa osztály Yakindu állapotgépes implementációját hajtottam végre. Láthatóan sokkal egyszerűbb, csak **Controls** interfésszel rendelkezik, ugyanis ő nem oszt meg információt a külvilággal. Szintén leolvasható az ábráról, hogy a kezdeti állapota az **Off**. A **Controls** **turnOn** eseményének következtében **On** állapotba kerül, visszafelé hasonlóan a **turnOff** hatására történik az állapotváltás.



. ábra: A lámpa megvalósítása Yakindu segítségével

### Osztályok generálása

A Yakindu lehetőséget ad az állapotgépeket megvalósító Java osztályok generálására. Ehhez egy új Generator Modelt hoztam létre. A felhasználói felületen kiválasztható, hogy mely statechartokhoz szeretném generálni a forráskódot.

Természetesen mindkét osztályt kiválasztottam. Az így létrehozott *default.sgen* Generator Modelt még ki kellett egészítenem.

|  |
| --- |
| **GeneratorModel** **for** yakindu::java {  **statechart** Light {  **feature** Outlet {  targetProject = "MQTTExample"  targetFolder = "src-gen"  libraryTargetFolder = "src"  }  **feature** GeneralFeatures{  RuntimeService = true  }  }  **statechart** Switch {  **feature** Outlet {  targetProject = "MQTTExample"  targetFolder = "src-gen"  libraryTargetFolder = "src"  }  **feature** GeneralFeatures{  TimerService = true  RuntimeService = true  InterfaceObserverSupport = true  }  }  } |

. ábra: Az állapotgépek, és a hozzájuk tartozó funkciók konfigurációja

A Generator Model alapértelmezetten tartalmazta az **Outlet** featuret mindkét osztály esetében, valamint a **GeneralFeatures** **RuntimeService** attribútumát is. Azonban ahhoz, hogy kapcsoló képes legyen belső időzítésre, illetve a kimenő eseményeinek hatására saját programkód futtatására, hozzá kellett adnom rendre a **TimerService** és az **InterfaceObserverSupport** attribútumokat.

## Message Queue Telemetry Transport (MQTT) és Paho

Az MQTT [2] egy publish-subscribe alapú TCP/IP protokoll. Kisebb üzenetek küldésére és fogadására lett tervezve. Minden kliens meghatározott topikra küldi az üzenetét vagy iratkozik fel. A működéséhez szükség van egy brókerre, aki felelős az üzenetek továbbításáért, valamint elosztásáért a kliensek között.

Az MQTT protokoll logikus választásnak tűnt, ugyanis a komponensek interfészeket definiálnak, melyek üzenetek küldésére és fogadására képesek. Az interfészek esetemben a topikok, így, ha valamelyik kliens számára az adott témához kapcsolódó üzenetek relevánsak, akkor feliratkozik rá. Amennyiben egy kliens az adott témával kapcsolatos megfigyeléseket végez, azokat meg tudja osztani az érdekelt felekkel a brókeren keresztül.

A Paho [3] projekt nyílt forráskódú implementációt nyújt MQTT üzenetküldésre. Ennek segítségével ki tudtam bővíteni úgy a Yakindu által generált forráskódot, hogy képesek legyenek a protokoll használatával a kommunikációra.

Ehhez szükség volt olyan osztályok létrehozására, amelyek egy MQTT kliens és egy állapotgép kompozíciója. Ezen felül implementálniuk kell az **MqttCallback** interfészt ahhoz, hogy az üzenetek küldésének és érkezésének végén lévő callback függvényeket megvalósíthassam.

A projektet szükséges volt Maven [4] projektté konvertálni, hogy ezt követően dependenciaként lehessen hivatkozni a jelenlegi Paho implementációra. Ehhez csak a következő sorokat kellett felvenni a pom.xml-be:

|  |
| --- |
| **<dependency>**  **<groupId>org.eclipse.paho</groupId>**  **<artifactId>org.eclipse.paho.client.mqttv3</artifactId>**  **<version>1.1.0</version>**  **</dependency>** |

. ábra: A függőség megadása a Maven számára

### MQTT protokollt használó osztály

Példaként bemutatom az **MQTTSwitch** osztályom, ugyanis ez rendelkezik az összes olyan feature implementációjával, amelyeknek részhalmazával tetszőleges kiberfizikai komponens előállítható egy ilyen rendszerben.

Az osztály négy attribútummal rendelkezik:

|  |
| --- |
| SwitchSMStatemachine statemachine;  MqttClient myClient;  MqttConnectOptions connOpts;  **boolean** started = **false**; |

. ábra: Az állapotgép attribútumai

Ezek egymást követően a belső Yakindus kapcsolónkant megvalósító állapotgép példánya, egy MQTT kliens és a hozzá kapcsolódó beállítások, valamint egy változó, amely az állapotgép elindulását ellenőrzi.

Az osztály konstruktorában ezeket a változókat inicializálom (amennyiben még nem történt meg), illetve csatlakozom a brókerhez. Az inicializáló függvényében az állapotgép időzítőjének adok egy új időzítőt és a **Publish** interfész listenerjeihez adok olyan függvényeket, amelyek a kimenő események hatására MQTT üzeneteket küldenek. Természetesen itt megadtam az üzeneteknek, hogy milyen topikra és milyen üzenetet küldjenek

|  |
| --- |
| **public** **void** init() {  **if** (!started) {  statemachine.setTimer(**new** TimerService());  statemachine.getSCIPublish()  .getListeners().add(new SCIPublishListener() {  **public** **void** onOnTurnOnRaised() {  String topic = "DummyTopic";  String content = "Turned On!";  MqttMessage message = **new** MqttMessage(content.getBytes());  **try** {  myClient.publish(topic, message);  } **catch** (MqttPersistenceException e) {  e.printStackTrace();  } **catch** (MqttException e) {  e.printStackTrace();  }  }  /\*  További kimenő eseményekre hasonló módon lehet feliratkozni és MQTT üzenetet küldeni  \*/  });  statemachine.init();  statemachine.enter();  RuntimeService.*getInstance*() .registerStatemachine(statemachine, 100);  started = **true**;  } |

6. ábra: A kibővített állapotgép inicializálása

Végül az állapotgépet is inicializálom, belépek a kezdeti állapotba, regisztrálom, és megjelölöm, hogy elindult.

Mivel a bekapcsolás külső hatásra történik, ezért kiajánlom a bekapcsoló függvényt.

|  |
| --- |
| **public** **void** turnOn() {  statemachine.getSCIControls().raiseTurnOn();  statemachine.runCycle();  } |

. ábra: Egy kiajánlott függvény az állapotgép kezeléséhez

Az **MqttCallback** implementálásakor szükséges felüldefiniálni a **connetcionLost, deliveryComplete,** és **messageArrived** függvényeket. Első kettőnél csak a konzolra kiírom az eseményeket, hogy futtatás közben értesüljek az esetlegesen megszakadt kapcsolatról, vagy hogy sikeresen továbbítva lett az üzenet a bróker felé. A **messageArrived** két paraméterrel rendelkezik *topic* és *msg*. A függvény implementációjában itt egy switch-case utasítással megadtam, hogy a **Controls** interfész mely bemenő eseményét érvényesítsem. Az **MqttSwitch** esetében ilyen nincs, de az **MqttLight** esetében a következő a megvalósítás:

|  |
| --- |
| **public** **void** messageArrived(String topic, MqttMessage msg) **throws** Exception {  **switch** (msg.toString()) {  **case** "Turned On!":  turnOn();  **break**;  **case** "Turned Off!":  turnOff();  **break**;  **default**:  System.***out***.println ("Unhandled message received: " + msg.toString());  **break**;  }  } |

. ábra: A bejövő üzenetek kezelése, majd a szükséges függvény meghívása

Ezt követően még három függvényt valósítottam meg. Ezek wrapperek az MQTT kliens **subscribe, publish,** és **disconnect,** metódusaihoz,melyek rendre a topikra való feliratkozáshoz, publikáláshoz, és a brókerrel való kapcsolat megszakításához szükségesek.

### MQTT kompatibilis állapotgép futtatása

Először is létre kellett hoznom egy brókert. Ehhez egy Mosquitto [4] szervert használtam lokálisan, melynek elindítása parancssorosan történik a telepítési mappából a következő paranccsal:

|  |
| --- |
| $ ./mosquitto.exe |

. ábra: Mosquitto bróker konzolos indítása

Lehetséges különálló publisher vagy subscriber indítása is, ezek a fejlesztés során kifejezetten hasznosak. Egy „MyTopic” topikot figyelő subscriber és egy ugyanerre a topikra „MyMessage” üzenetet küldő publisher a következőképpen indítható:

|  |
| --- |
| $ ./mosquitto\_sub.exe -t "MyTopic"  $ ./mosquitto\_pub.exe -t "MyTopic" -t "MyMessage" |

. ábra: Parancssoros subscriber és publisher indítása

Visszatérve az MQTT kompatibilis Yakindu által generált állapotgép kódját használó osztályhoz, készítettem egy **SwitchRunner** nevezetű osztályt, ami tartalmaz minden szükséges információt ahhoz, hogy egy **MqttSwitch** futtatható legyen.

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** InterruptedException {  String broker = "tcp://localhost:1883";  String clientId = "DummyClient";  MemoryPersistence persistence = **new** MemoryPersistence();  MQTTSwitch sampleClient1 = **new** MQTTSwitch(broker, clientId + '1', persistence);  sampleClient1.init();  sampleClient1.turnOn();  } |

. ábra: Az MQTTSwitchet futtató main függvény

Az alapértelmezetten az 1883-as porton futó Mosquitto brókerhez csatlakozik. Szükséges neki egy kliens azonosító, valamint opcionálisan egy tároló, ha esetleg újraindulna a kliens. Ezt követően már csak a kliens inicializálása van hátra, valamint a teszteléshez a kapcsolót is működésbe hozom.

Ehhez képest a **LightRunner** kiegészítésre szorul, ugyanis szüksége van egy topikra is, valamint egy Quality of Service-t meghatározó konstansra is. Ez három értéket vehet fel, melyek az üzenet továbbítására vonatkoznak:

* 0 – legfeljebb egyszer
* 1 – legalább egyszer
* 2 – pontosan egyszer

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** InterruptedException {  String topic = "DummyTopic";  **int** qos = 2;  String broker = "tcp://localhost:1883";  String clientId = "DummyClient";  MemoryPersistence persistence = **new** MemoryPersistence();  **try** {  MQTTLight sampleClient2 = **new** MQTTLight(broker, clientId + '2', persistence);  sampleClient2.init();  sampleClient2.subscribe(topic, qos);  } **catch**(MqttException me) {  /\* hibakezelés \*/  }  } |

. ábra: Subscriberrel rendelkező állapotgép inicializálása

# Hálózati kommunikációt megvalósító állapotgépek

Az egyszerű, de funkcionális példa elkészítésével végig jártam azokat a lépéseket, melyek egy összetettebb kiberfizikai rendszer elkészítéséhez szükségesek. Azonban nem lenne célszerű minden állapotgéphez kézzel elkészíteni az MQTT használatához szükséges függvényeket, egy konfiguráció elkészítésével ezek akár generálhatóak is lennének.

Az állapotgépek ilyen formában azonban nem rendelkeznek minden szükséges információval ahhoz, hogy képesek legyenek hálózati kommunikációra. Ebben a formában a rendszert fejlesztő személynek kellene kiegészíteni a generált kódokat, ami szintén időigényes és repetitív feladat. A megoldás egy konfigurációs állomány elkészítése, ami rendelkezik minden olyan paraméterrel, ami a működő rendszer generálásához szükséges.

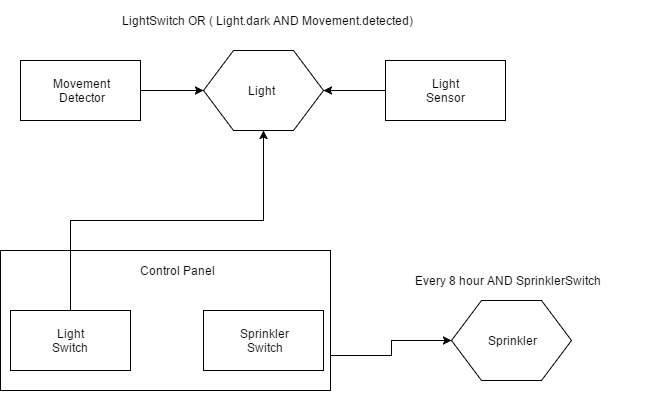
## A kiberfizikai rendszer komponensei

Az esettanulmányhoz egy kerti rendszert terveztem. Ennek részei:

* mozgásérzékelő
* fényérzékelő
* lámpa
* locsoló
* vezérlőpanel, mely tartalmaz egy kapcsolót a lámpához és a locsolóhoz

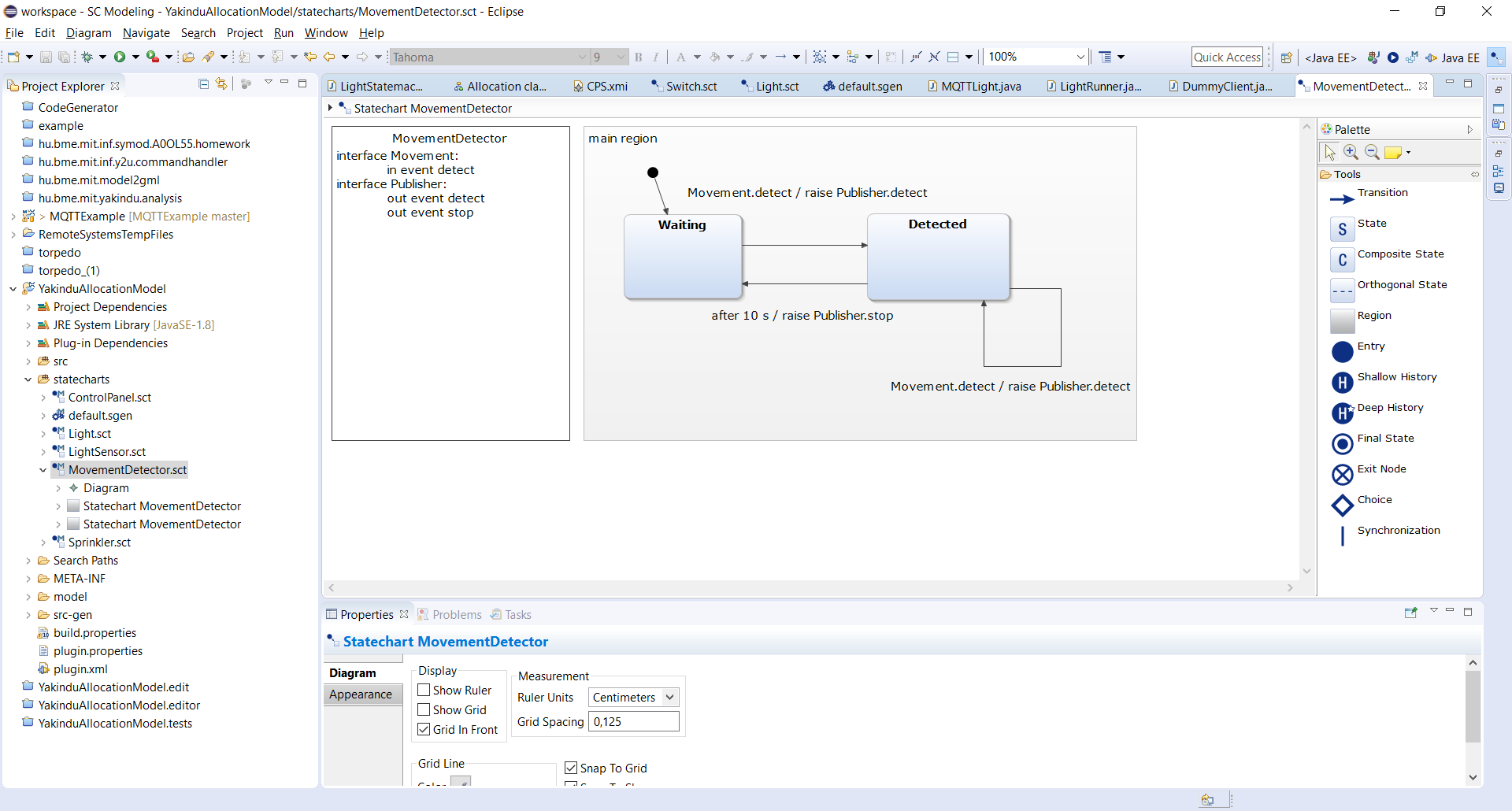
A kiberfizikai rendszerben két subscriber található, a lámpa és a locsoló. Ők csak megfigyelőként vesznek részt a kiberfizikai rendszerben, mások számára információt nem biztosítanak. A többi komponens publisherként viselkedik. Megfigyelik a körülöttük lévő világot, annak változásaira, illetve a vezérlőpanel esetében az emberi interakcióra üzenetet továbbítanak a bróker felé.

A lámpa világít, amennyiben a hozzá tartozó kapcsoló be van kapcsolva, vagy sötét van, de mozgást érzékelt a rendszer. Ezen feltételek kezelését a benne található állapotgép kezeli. A locsoló minden nyolcadik órában fél órára működésbe lép, vagy akkor, ha a vezérlőpanelen található kapcsolót használják. Ezt a működést is könnyen le lehet írni a Yakindu által biztosított eszközökkel.



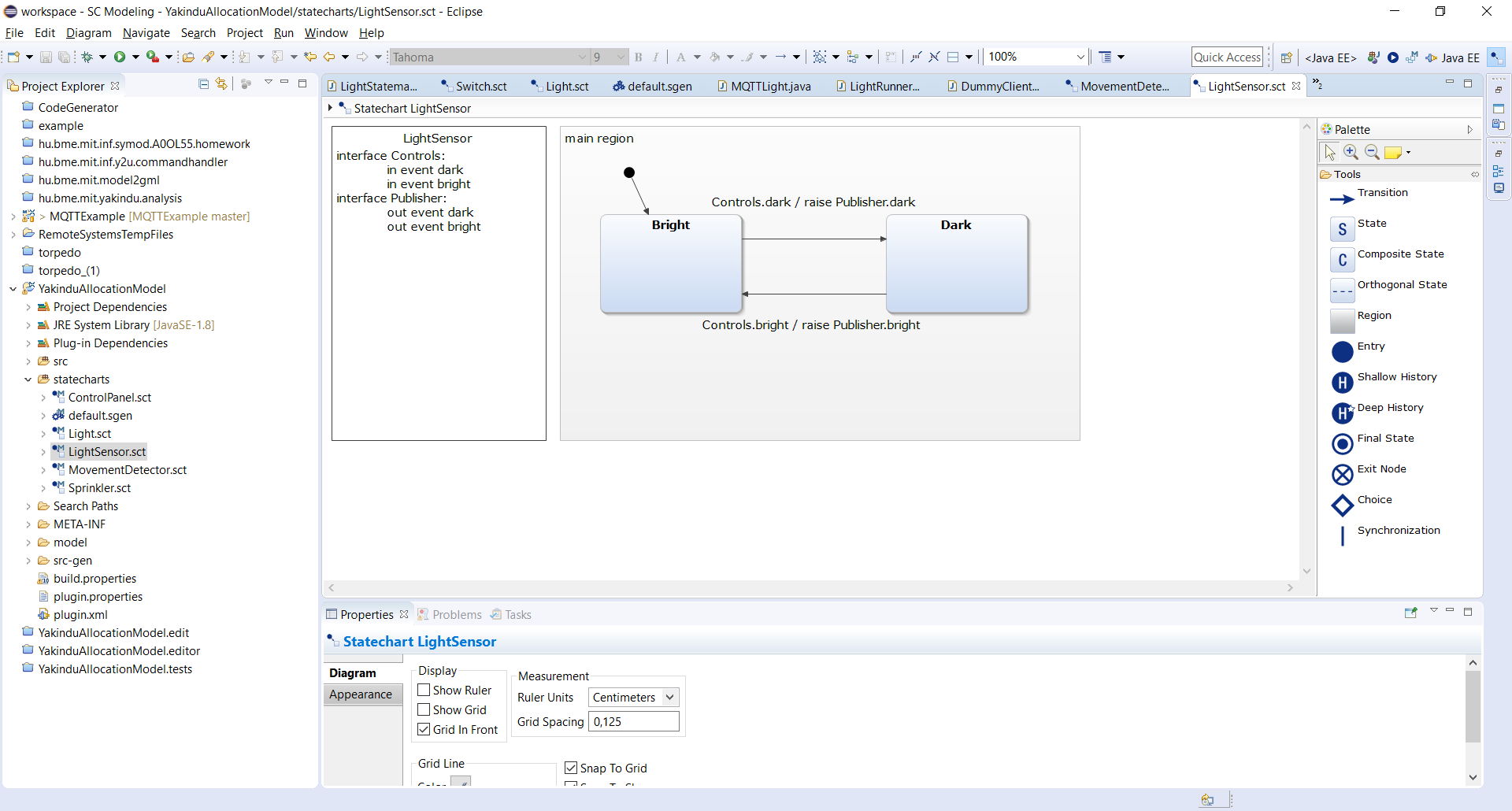
. ábra: A rendszer vázlata

A komponenseket egyenként elkészítettem a Yakindu segítségével. A mozgásérzékelő alapesetben várakozik, érzékelés hatására, majd ezt követő további mozgásokra a „mozgás érzékelve” állapotába jut. Ha a következő tíz másodpercben nem történik semmi, akkor visszatér a várakozó stádiumba. Az állapotváltások esetén mindig kiváltódik az adott esemény, mellyel tájékoztatja a feliratkozókat.



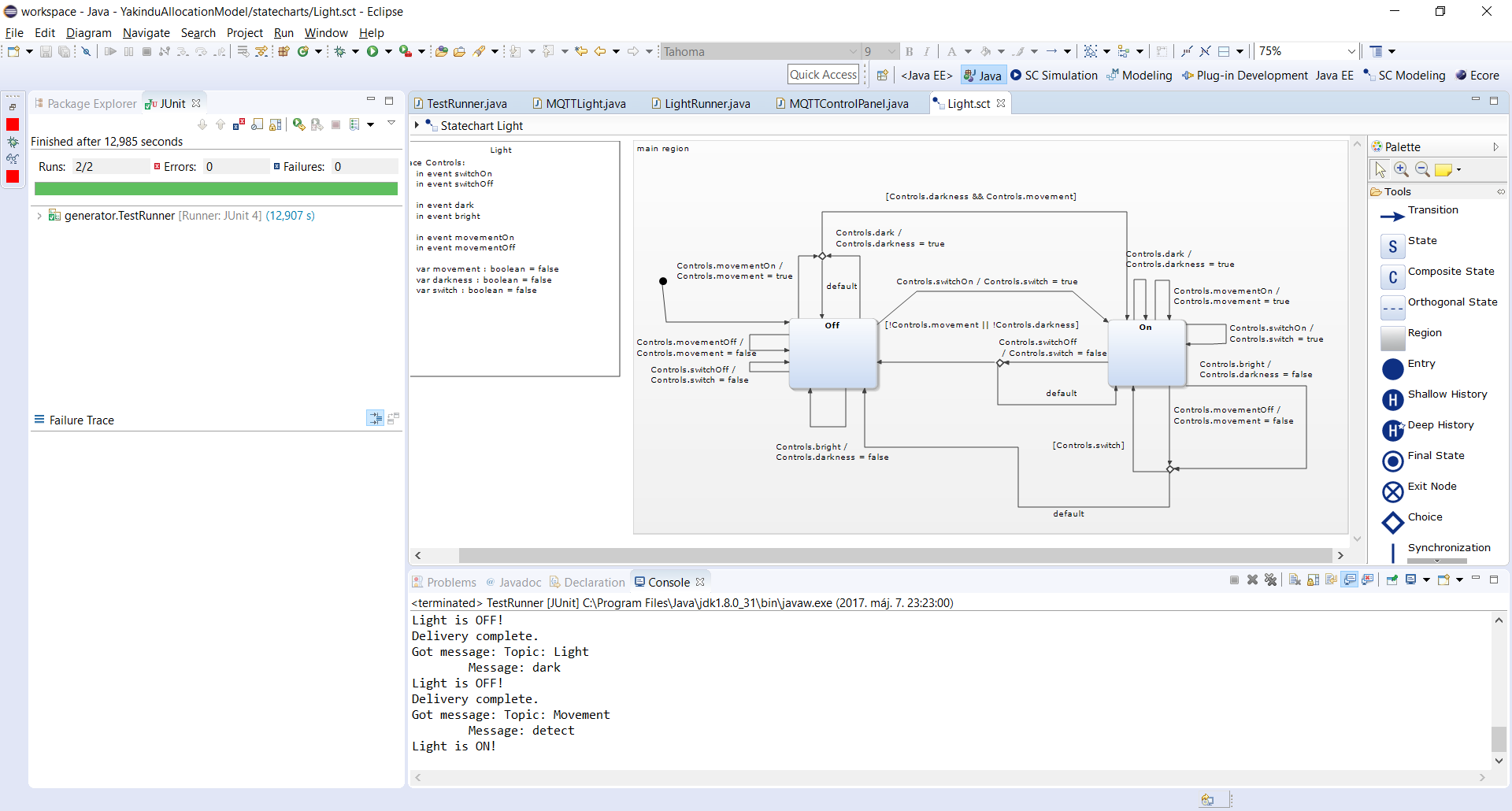
. ábra: A mozgásérzékelő blokkváza

A fényérzékelő két állapottal rendelkezik, vagy sötét van, vagy világos. Az ezek közötti átmenetek esetén kifelé küld erről egy eseményt.



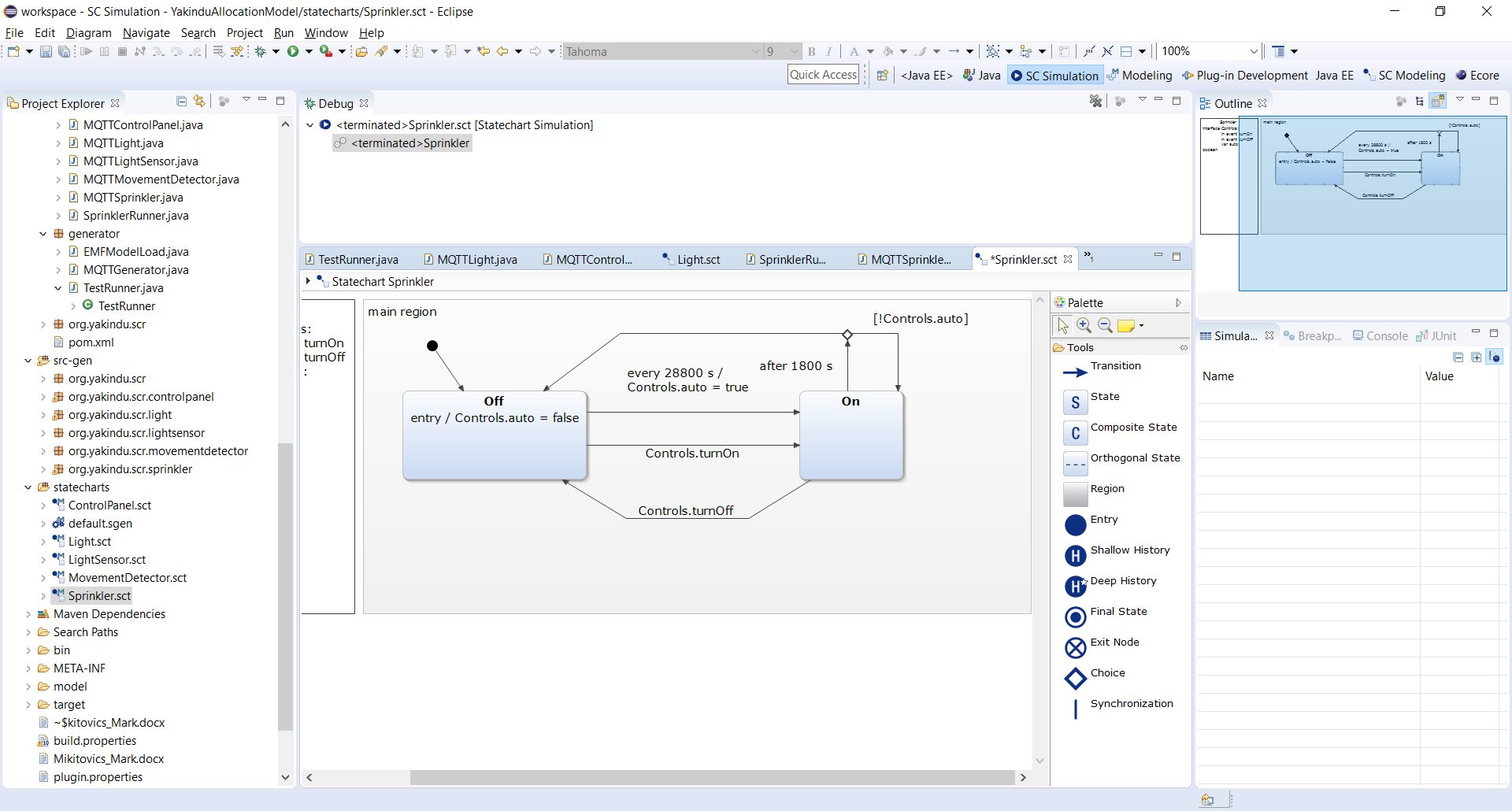
. ábra: A fényérzékelő blokkváza

A lámpa működése a legösszetettebb a rendszerben. Alapesetben ki van kapcsolva. Amennyiben bekapcsolják, vagy sötét van és mozgást érzékel, fel kell kapcsolódnia. A bejövő események nem várt sorrendben is következhetnek, erre is fel kellett készülnöm. Erre egy példa, hogy akkor is felkapcsolva kell maradnia, amikor a kapcsoló fel van nyomva, de eközben kivilágosodik, és már mozgás sincs. Ahhoz, hogy ezeket az eseteket nyomon tudjam követni, a bejövő eseményeken kívül három változót is fel kellett vennem. Ezek a változók rendre azt mutatják, hogy sötét van-e, történt-e mozgásérzékelés, vagy a kapcsoló hatására kerültünk a bekapcsolt állapotba.



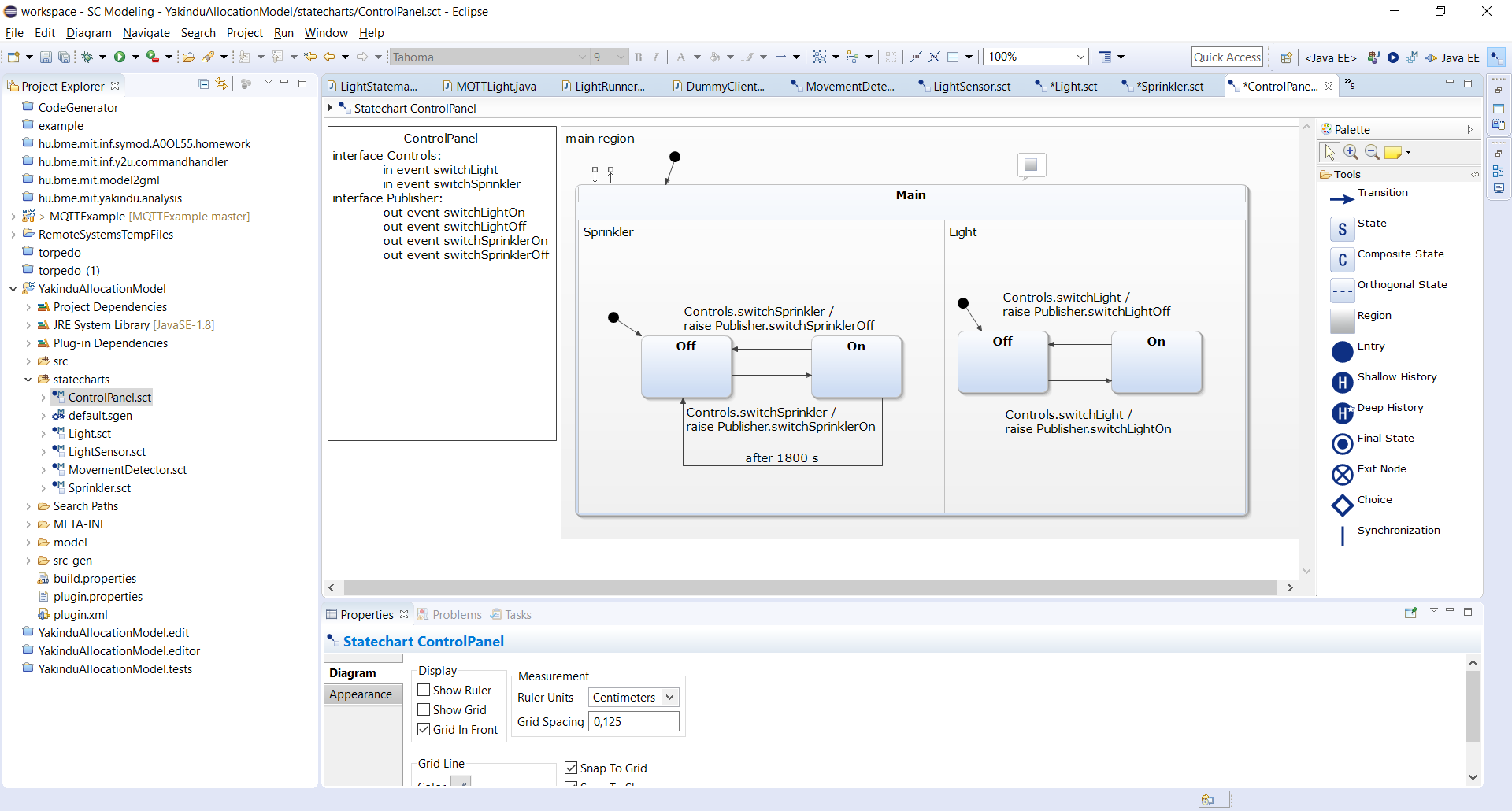
. ábra: A lámpát reprezentáló állapotgép

A locsoló felépítése egyszerű: vagy működésben van, vagy nem. Egyedül a belső időzítéshez szükséges 8 órát kellett átváltani másodpercbe, ugyanis ez a legkisebb időegység, amit a Yakindu kezel. Azt érdemes még figyelni, hogy a bekapcsolás felhasználói interakcióra történt a panel segítségével, vagy automatikus volt-e. Utóbbi esetben fél óra elteltével magától lekapcsol a locsoló. Amennyiben a panel utasítására kezdődött el a locsolás, várakozunk az azt követő kikapcsolásra utasító üzenetre.



. ábra: A locsoló blokkváza

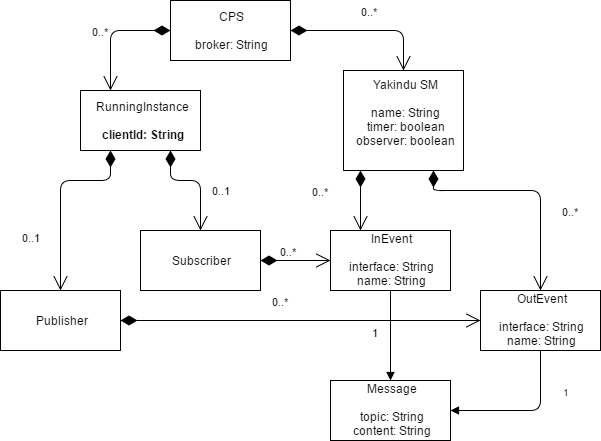
A vezérlőpanel egy összetett belső állapottal rendelkezik. Egymástól függetlenül működik a locsoló és a lámpa működését befolyásoló kapcsolószerkezet. Természetesen itt is minden átmenet esetében kiküldésre kerül az adott esemény.



. ábra: A vezérlőpanel működése

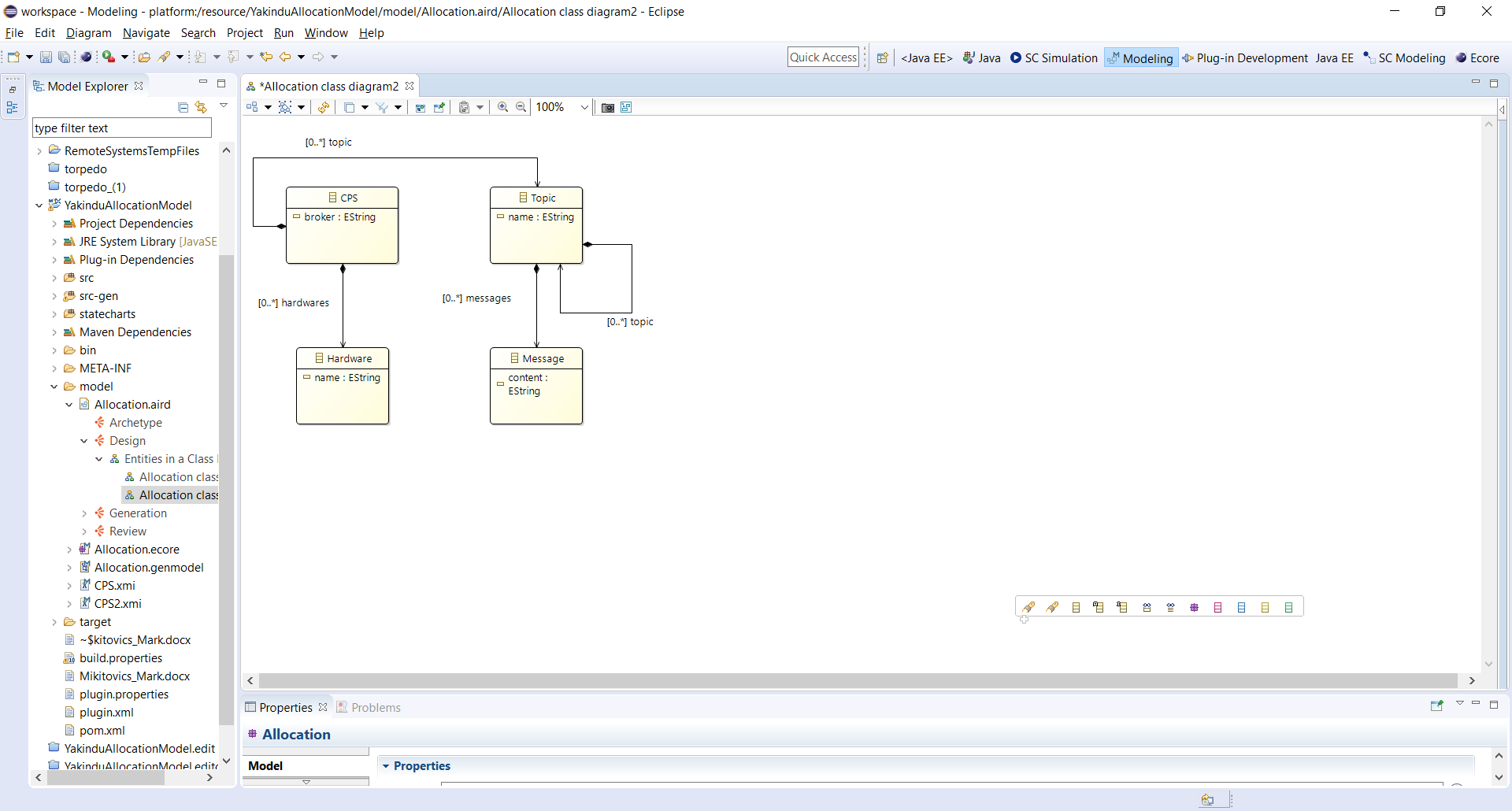
## Konfigurációs fájl elkészítése

Ahhoz, hogy ne kelljen kézzel elkészíteni minden állapotgépből származtatott osztályt, valamint a köztük lévő interfészek kapcsolatát implementálni, egy konfigurációs fájl létrehozása mellett döntöttem. Ehhez egy Eclipse Entity Modeling Framework (EMF) [5] core diagramot használtam, melyben megadtam ez előforduló entitásokat a köztük lévő kapcsolatokat, valamint a végleges kód generálásához nélkülözhetetlen attribútumokat is felvettem hozzájuk.



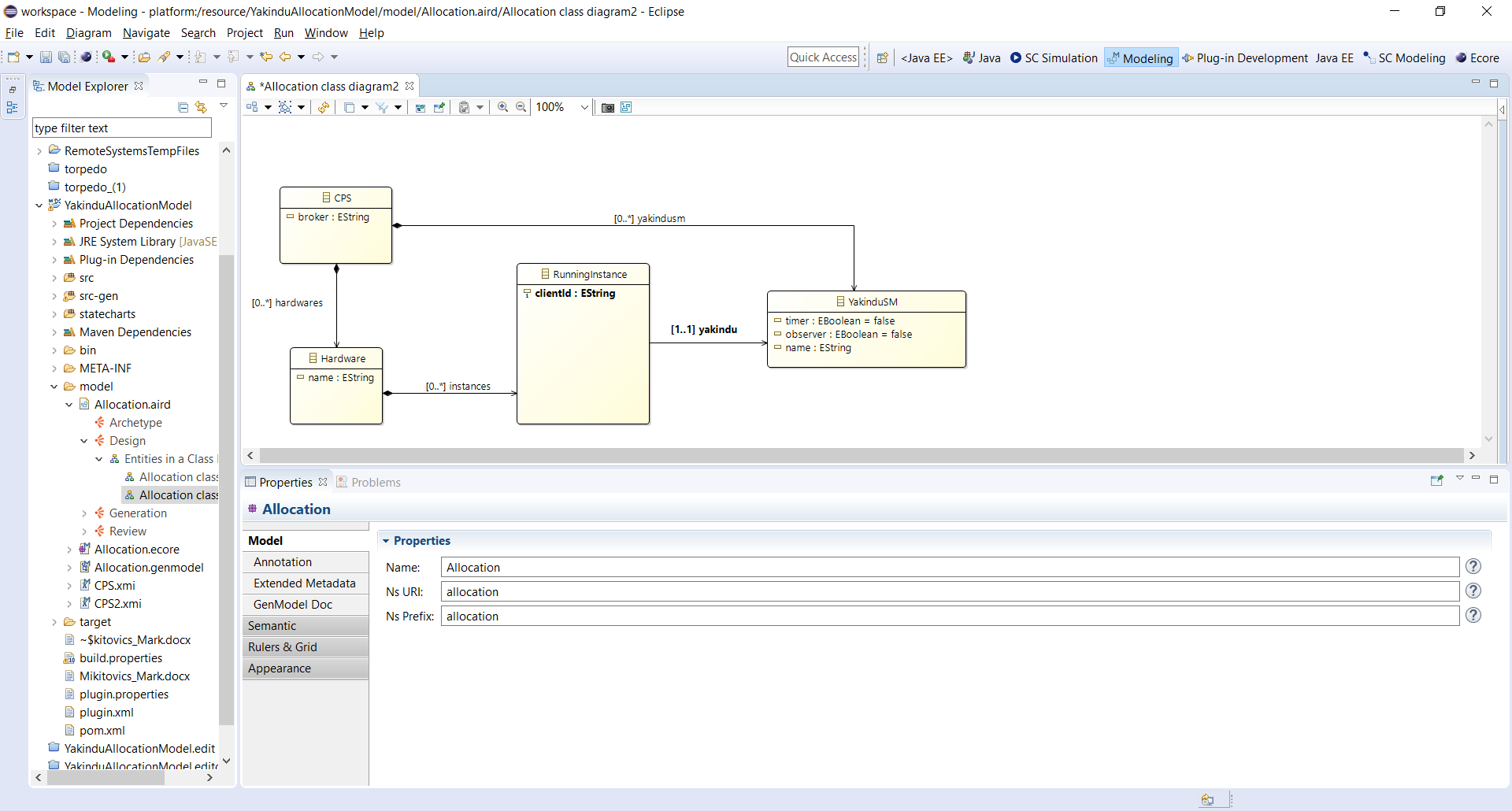
. ábra: Az allokációs diagram egyszerűsített változata

A kiberfizikai rendszerben való kommunikáció alapja az MQTT által megkövetelt topikok és üzenetek használata. Ehhez a rendszer konfigurációjának idejében ismernünk kell ezeket. Az MQTT lehetőséget biztosít altopikok létrehozására is. Ezek akkor hasznosak, ha léteznek komponensek, amelyek egy adott témának nem minden információjára kíváncsiak, így nem terheljük őket fölöslegesen.



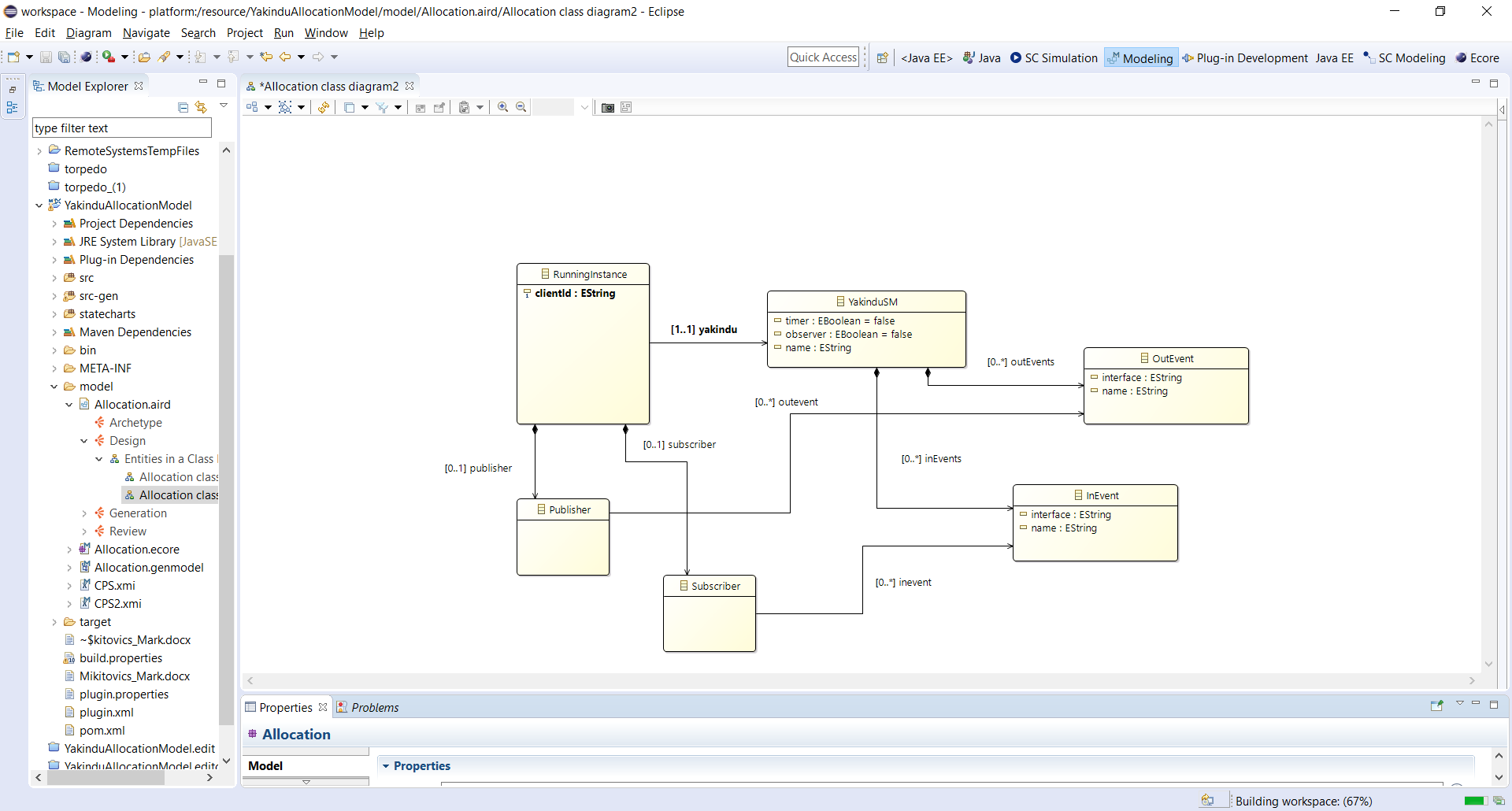
. ábra: A diagram egy részlete, amely ábrázolja a rendszer, a topikok és az üzenetek kapcsolatát

A kiberfizikai rendszer (CPS) tárolja a közös bróker címét. Megadása a konfigurációs állományban elengedhetetlen a kódgenerálás szempontjából, ugyanis minden generált komponens ezzel a címmel fog kommunikálni. Futtatható állományokból áll, melyekhez Yakindu állapotgépek tartoznak. A hardverek akár magukban foglalhatnának több futtatható állományt, ezáltal állapotgépet is, ugyanis előfordulhat, hogy egy hardver több feladatot is ellát. Egy könnyen elképzelhető példa egy olyan eszköz, amely egyszerre vizsgálja a hőmérsékletet és a levegő páratartalmát is.



. ábra: A kiberfizikai rendszerszer hardvereinek összerendelése állapotgépekkel

A futtatható komponensekhez tartozhat publisher, subscriber vagy akár mindkettő is. Ezek tartalmazzák a Yakindu állapotgépek bejövő, vagy kimenő eseményeit egy adott interfészhez. Itt szükséges meghatározni, hogy ezek az események milyen topikra milyen üzenetet küldenek. Ez elengedhetetlen ahhoz, hogy a Yakindu által generált interfészek felüldefiniálásakor megadjuk, hogy milyen témában publikálja az adott komponens az üzenetét. A fogadó fél szempontjából a bejövő üzenetek kezelésekor pedig ezek alapján döntjük el, hogy mely bemenő esemény legyen kiváltva az állapotgépnél. Fontos továbbá, hogy klienseknek egyedi azonosítóval kell rendelkezniük.



. ábra: A futtatható példány interfészeinek kapcsolata az állapotgép kimenő és bejövő eseményeivel

A Yakindu állapotgépeknek van neve, valamint jelezni kell a generált kódhoz, hogy támaszkodnak-e az időzítőre, vagy a kimenő eseményeikhez szükséges-e hozzáadni az MQTT kommunikációhoz szükséges kódsorokat. Ezen felül az állapotgépekben bejövő és kimenő események is jelen lehetnek tetszőleges számban.

Miután elkészítettem a rendszer metamodelljét, és hozzáadtam a végleges kód generálásához szükséges attribútumokat, megvalósítottam ennek a konfigurációnak egy példányát.

### Az állapotgépekhez tartozó konfigurációk

A konfigurációs fájlban minden állapotgépnél fel kell tüntetni nevüket. Ez alapján történik az importálás a Yakindu által generált java forráskódból. Ezen felül két további paraméter beállítása fontos a kódgeneráló megfelelő működéséhez.

A **Timer** változó azt jelzi, hogy az adott komponensnek szüksége van-e időzítő funkcióra. A mozgásérzékelő, a vezérlőpanel és a locsoló állapotgépek esetében ezt igazra állítottam. Az **Observer** változót abban az esetben kell igazra állítani, amennyiben az adott állapotgép eseményeit szeretnénk kívülről bővíteni. Ennek segítségével oldom meg, hogy a kimenő eseményeket felüldefiniáljam, hozzáadjam az MQTT üzenetküldést.

Továbbá minden állapotgéphez tartozó konfigurációban fel kell venni a kimenő és bejövő eseményeket, és azoknak megadni, hogy mi a nevük, az állapotgép mely interfészét használják, és melyik üzenettel vannak kapcsolatban. Ahol egyezik az üzenet (és ez által a topik is) két különböző állapotgép kimenő és bejövő eseményénél, ott fog reagálni a subscriber a publisher által továbbított információra.

### A kiberfizikai rendszer hardverei

A konfiguráció egy kiberfizikai rendszer, amelyhez tartozó bróker a tcp://localhost:1883 címen érhető el, ez a mosquitto alapértelmezett címe. Öt hardvert foglal magában:

* Fényérzékelő
* Mozgásérzékelő
* Vezérlőpanel
* Lámpa
* Locsoló

A fényérzékelő hardver magában foglalja a futtatható példányát a hozzá tartozó állapotgépnek, valamint egy publisherrel rendelkezik, ami a **Light** topikra küld **dark** és **bright** üzenetet, attól függően, hogy sötét van vagy világos.

A mozgásérzékelő szintén egy publishert tartalmaz a futó példányon és az ahhoz tartozó állapotgépen kívül. A **Movement** topikban megjelenő **detect** és **stop** üzenetek jelzik, hogy éppen mit érzékel.

A vezérlőpanel az előző két hardverhez hasonlóan rendelkezik publisher funkcionalitással, és az állapotgépének egy futtatható példányával. Négy kimenő eseménye van, melyek a lámpa és a locsoló irányítására hivatottak. Ezek a **switchSprinklerOff, switchSprinklerOn, switchLightOff** és **switchLightOn** események, amelyek a **ControlPanel** topikra kerülnek kiküldésre.

A lámpához tartozik egy futtatható példány, amely csak subscribert tartalmaz. Ez feliratkozik a fényérzékelő sötét és világos eseményére, a mozgásérzékelő beindulására és kikapcsolására, valamint a vezérlőpanel lámpát be és kikapcsoló eseményére.

A locsoló, akárcsak a többi hardver tartalmazza a futtható példányát az állapotgépnek, amely feliratkozik a vezérlőpanel eseményeire, és figyeli, hogy kiváltódott-e számára releváns esemény.

### A rendszerben található topikok

Három topikkal rendelkezik a rendszer, a fénnyel, a mozgással, és a vezérlőpanellel kapcsolatosak egyenként. A fény topikon arról jöhet üzenet, hogy világos, vagy sötét van-e. A mozgás témában az érzékelésről, illetve a mozgás megszűnéséről kapnak tájékoztatást a feliratkozó komponensek. A vezérlőpanel topikjában pedig a lámpa és a locsoló fel- illetve lekapcsolására történik utasítás. A konfigurációban azt is meg kell adni, hogy a topikokon milyen üzeneteket várunk, ugyanis ezzel tudják egyértelműen beazonosítani a feliratkozó komponensek, hogy milyen változások történtek körülöttük.

A topikok és a tartalmazott üzenetek a következők:

* Light
  + dark
  + bright
* Movement
  + detect
  + stop
* ControlPanel
  + switchSprinklerOff
  + switchSprinklerOn
  + switchLightOff
  + switchLightOn

## A kódgeneráló működése

Az előbb létrehozott EMF modell egy xmi fájlban van, amelynek elemein az EMF segítségével könnyen végig tudtam iterálni. Ehhez két osztályt hoztam létre, az első az **EMFModellLoad.java**, amely az előbb említett konfiguráció betöltését végzi.

Először is inicializálni kell a modelt, ami ebben az esetben az **AllocationPackage**. Ezt követően beolvasom a *CPS.xmi*fájlomat, ami tartalmazza az elkészített konfigurációt. Elkérem az első elemet, és a megfelelő típussá konvertálom. Ez lesz a kiberfizikai rendszer, a gyökérelem a listában. Ennek az elemnek a gyerekei tartalmazzák az összes szükséges beállítást ahhoz, hogy működő, egymással MQTT kommunikációra képes állapotgépeket tudjak generálni.

|  |
| --- |
| **public** **class** EMFModelLoad {  **public** CPS load() {  AllocationPackage.***eINSTANCE***.eClass();  Resource.Factory.Registry reg = Resource.Factory.Registry.***INSTANCE***;  Map<String, Object> m = reg.getExtensionToFactoryMap();  m.put("xmi", **new** XMIResourceFactoryImpl());  ResourceSet resSet = **new** ResourceSetImpl();  Resource resource = resSet.getResource(URI  .*createURI*("model/CPS.xmi"), **true**);  CPS cps = (CPS) resource.getContents().get(0);  **return** cps;  }  } |

. ábra: EMF model betöltése

Ezt használja fel a kódgenerálást végző **MQTTGenerator.java**. A betöltés után a hardvereken iterálok végig, amik becsomagolják a már említett futtatható állományokat. Egy hardver tartalmazhatna több ilyet is, de a példában mindegyik csak eggyel rendelkezik. Minden futtatható állományhoz két összetevőre van szükség, a **main** függvénnyel rendelkező **Runner** és az MQTT funkcionalitással bővített állapotgépet megvalósító **MQTTClass** osztályra.

|  |
| --- |
| **public** **static** **void** main(String[] args) {  EMFModelLoad loader = **new** EMFModelLoad();  CPS cps = loader.load();  **for** (Iterator<Hardware> hardwareIterator =  cps.getHardwares().iterator();  hardwareIterator.hasNext();) {    Hardware hw = hardwareIterator.next();  **for** (Iterator<RunningInstance> instanceIterator =  hw.getInstances().iterator(); instanceIterator.hasNext();) {    RunningInstance runningInstance = instanceIterator.next();  *generateRunner*(runningInstance, cps.getBroker());  *generateMQTTClass*(runningInstance.getYakindu());  }  }  } |

. ábra: A kiberfizikai rendszer gyerekein iteráló kódgenerátor

A működő kód generálásához ezért két további függvényt hoztam létre, a **generateRunner** függvény a futtató osztályokat, a **generateMQTTClass** pedig kiegészített állapotgépeket generálja. A kódgenerálás mindkét esetben egyszerű fájlba írással történik, először a szükséges külső osztályok importálásával. A fájlok neve az adott állapotgép, és a neve elé fűzött MQTT, vagy az utána fűzött Runner, attól függően, hogy melyik osztály az.

A **generateRunner** metódusban létrehozom azt a függvényt, amely futtatja az MQTT kommunikációra képes állapotgépeket. Először importálom a szükséges osztályokat, majd beállítom a bróker és a kliens azonosítókat. Az MQTT osztály létrehozása és inicializálása után, ha az adott osztály tartalmaz subscribert is, feliratkozok azokra a topikokra, amelyek az adott komponens működése szempontjából relevánsak. Ezt követi a hibakezelés, majd a fájlba írás megszűntetése a kimenet lezárásával.

A **generateMQTTClass** függvényben a Yakindu állapotgépet használom fel az osztály létrehozásához. Amennyiben az adott osztály támaszkodik időzítőre, vagy szükséges felüldefiniálni a kimenő eseményeit a kommunikáció érdekében, az ehhez szükséges osztályokat is hozzáadom az importálásokhoz. Bármilyen MQTT funkcionalitással kibővített osztályt generálok, szükség van egy inicializáló részre. A bejövő eseményeket minden esetben elérhetővé teszem kívülről is. Ez fontos a tesztelés, illetve a külső vezérelhetőség szempontjából is.

Az elkészült main függvénnyel rendelkező osztályokat elindítva csatlakoznak a brókerhez, és készek arra, hogy üzeneteket küldjenek és fogadjanak.

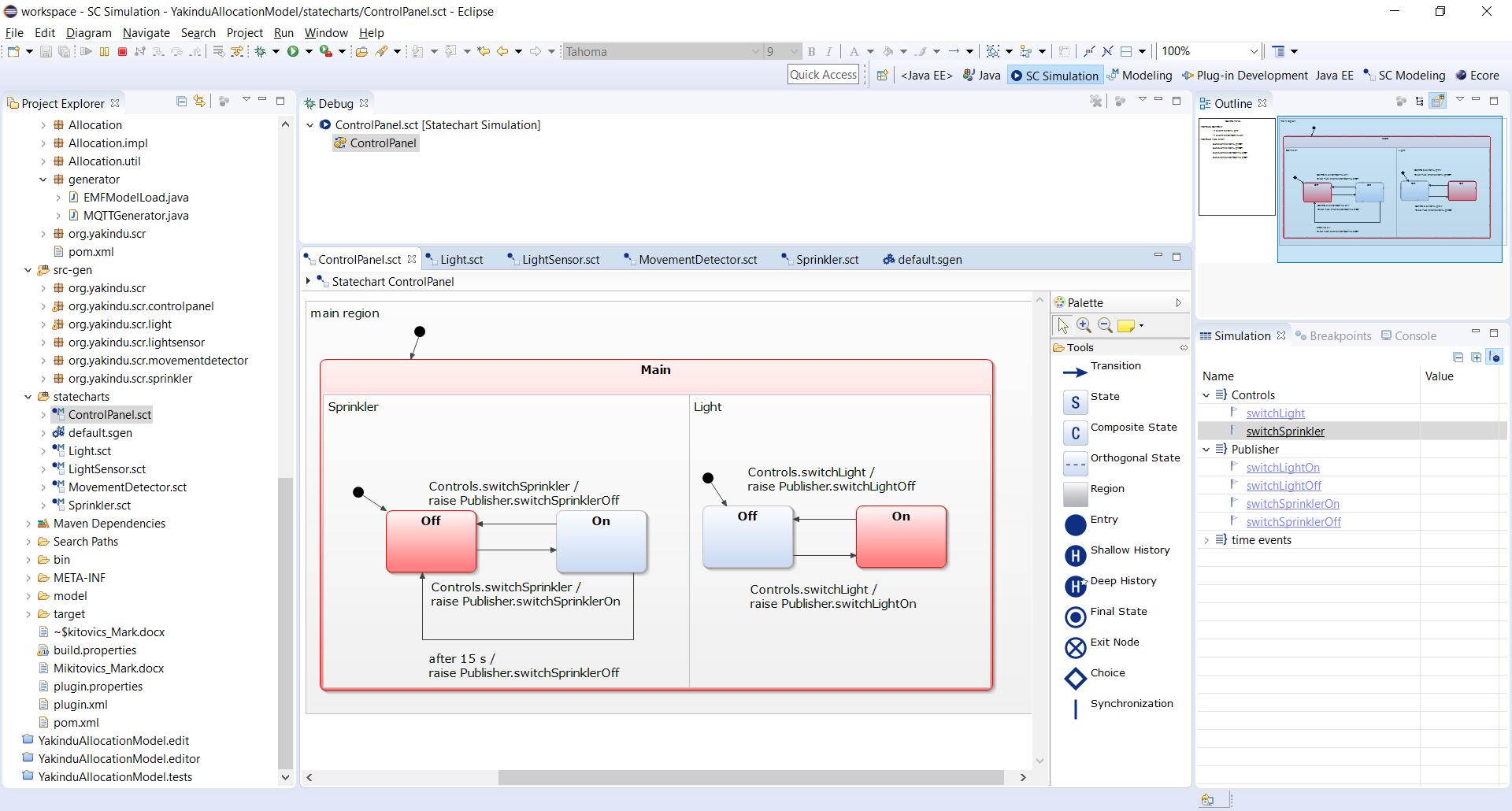
## Tesztelés

Ahhoz, hogy megbizonyosodjak a generált kód megfelelő működéséről, különböző teszteseteket készítettem, és vizsgáltam a komponensek tevékenységét és állapotváltásait. Ezeket a teszteseteket először a Yakindu által biztosított szimulációban hajtottam végre először, majd ezt követően az általam generált futtató osztályokban idéztem elő ugyanezt a működést, és vetettem össze az eredményeket. Az első manuális teszt elkészítése után nyilvánvalóvá vált számomra, hogy a bonyolultabb esetek teljes lefedése érdekében automatikusan futtatható teszteket kell létrehoznom. Ehhez a JUnit [6] keretrendszert választottam. Tesztelés során annyi változtatást hajtottam végre a locsolón és a vezérlőpanelen, hogy a fél óra után bekövetkező kikapcsolást megváltoztattam 5 másodpercre, valamint a 8 óránként bekövetkező locsolást is átállítottam 10 másodpercre.

## Tesztesetek bemutatása, a rendszer működés közben

Egyenként kipróbáltam az állapotgépeket szimulációban, hogy megbizonyosodjak arról, hogy önálló futásuk során nem történik hiba. A locsoló, a fény- és mozgásérzékelő kevés lehetséges esettel rendelkezik, hamar meggyőződtem arról, hogy jól működnek.

A vezérlőpanel esetén megvizsgáltam, hogy az összetett állapotban függetlenek-e egymástól a locsoló és a lámpa kapcsolói. Mind a négy lehetséges állapot könnyen előidézhető, a locsoló kapcsolója az elvárt működés szerint 5 másodperc elteltével automatikusan lekapcsolt.



. ábra: Kompozit állapotok szimuláció közben

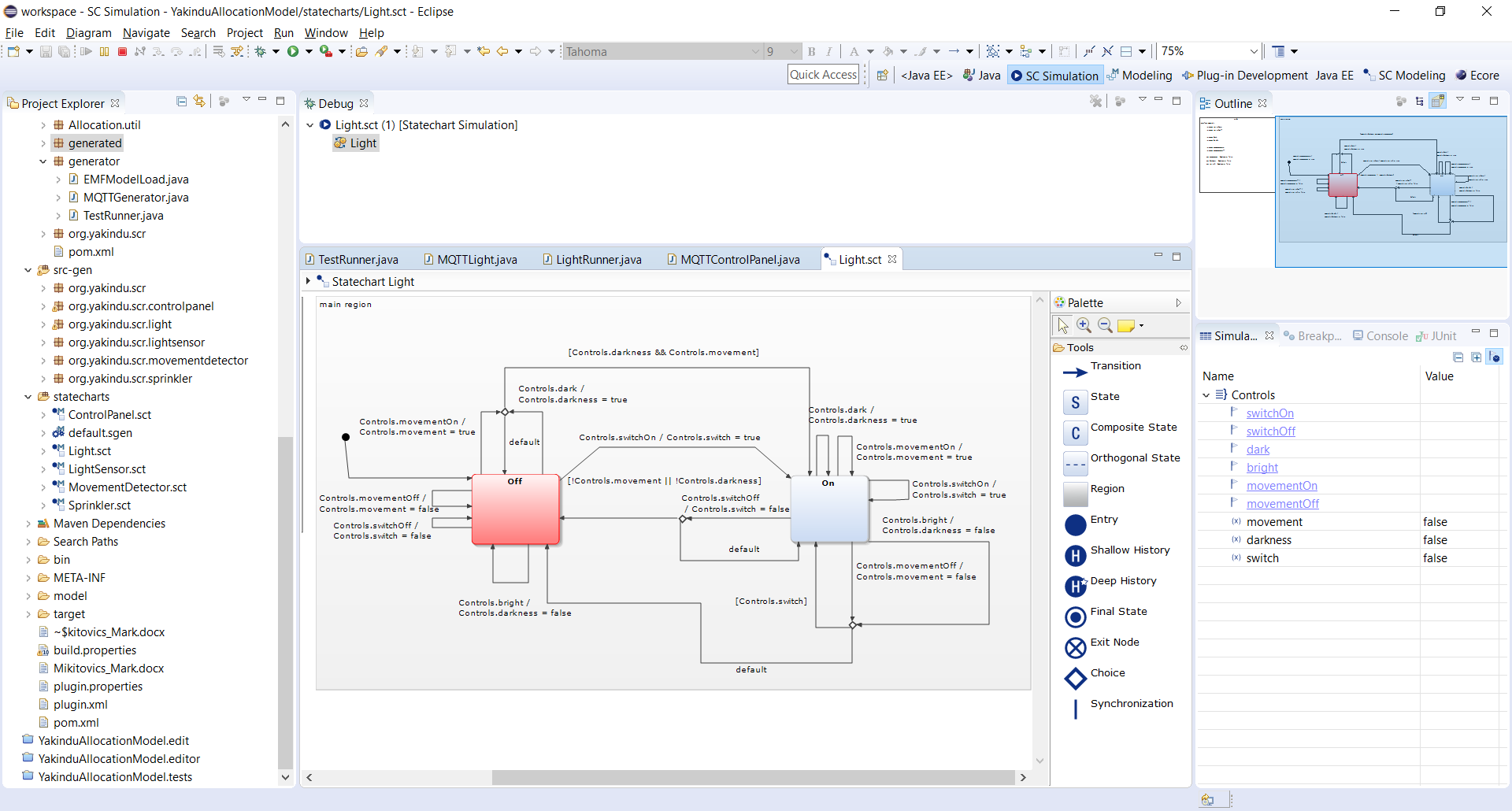
A legtöbb lehetséges kimenetel a lámpánál fordulhat elő. A vezérlőpanel mindig felülírja az érzékelők által előidézhető bekapcsolást, viszont amennyiben az le van kapcsolva, a sötétség és a mozgás érzékelése esetén szintén beindul. Az események hatására az elvárt állapotok a kapcsoló, a fényérzékelő és a mozgásérzékelő függvényében a következők:

. táblázat: A lámpa működése a három változó függvényében

|  |  |  |  |
| --- | --- | --- | --- |
|  | | Mozgás – be | Mozgás – ki |
| Kapcsoló – be | Fény – be | Bekapcsolva | |
| Fény – ki |
| Kapcsoló – ki | Fény – be | Bekapcsolva | Kikapcsolva |
| Fény – ki | Kikapcsolva | Kikapcsolva |

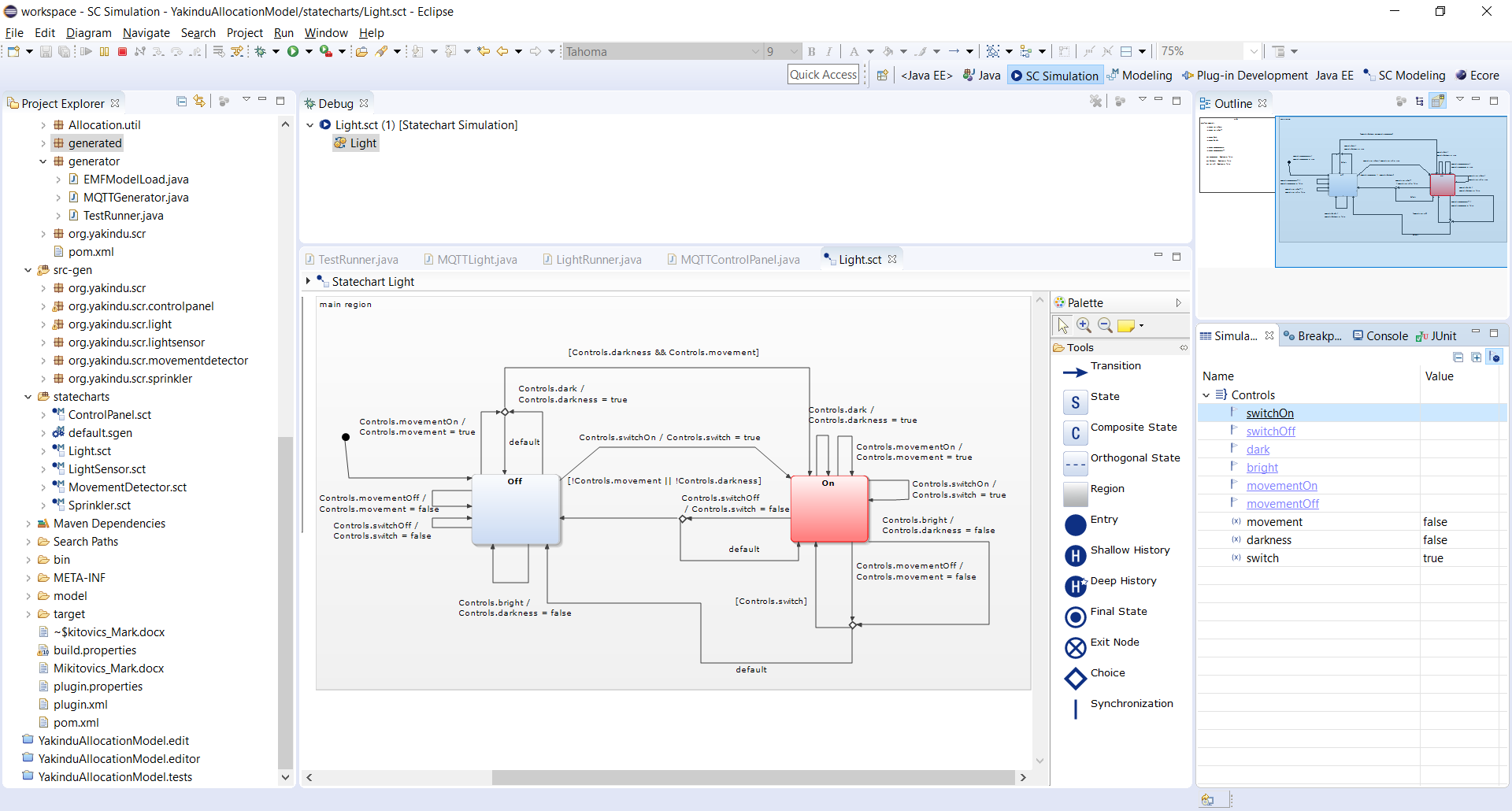
A felsorolt lehetőségeket szimulációban egyszerűen ki tudtam próbálni, a **Controls** interfész egyes eseményeit kiváltva. Miután minden esetben az elvárt működést tapasztaltam, kipróbáltam az eseteket működő állapotgépeken is. Egy példát be is mutatok a rendszer együttes tesztelése során.

Az rendszer együttes tesztelését a lámpa működésének vizsgálatával kezdtem. Először is megnéztem, hogy amennyiben a fényérzékelő jelez, hogy besötétedett, majd a mozgásérzékelő is jelez, a lámpa helyesen működik-e. A szimuláció során csak egy állapotgépet tudok megfigyelni, a bejövő eseményeket viszont tudom generálni. Kezdetben a lámpa kikapcsolt állapotban van.



. ábra: A lámpa kezdeti állapota

Ezt követően generáltam egy jelzést, melynek hatására a belső, sötétséget indikáló változó az elvárt működés szerint igaz lett. Ekkor a lámpa még mindig kikapcsolt állapotban van. A következő mozgás esemény hatására felkapcsolódik, illetve ki is kapcsol, ha azt az üzenetet kapja, hogy megszűnt az.



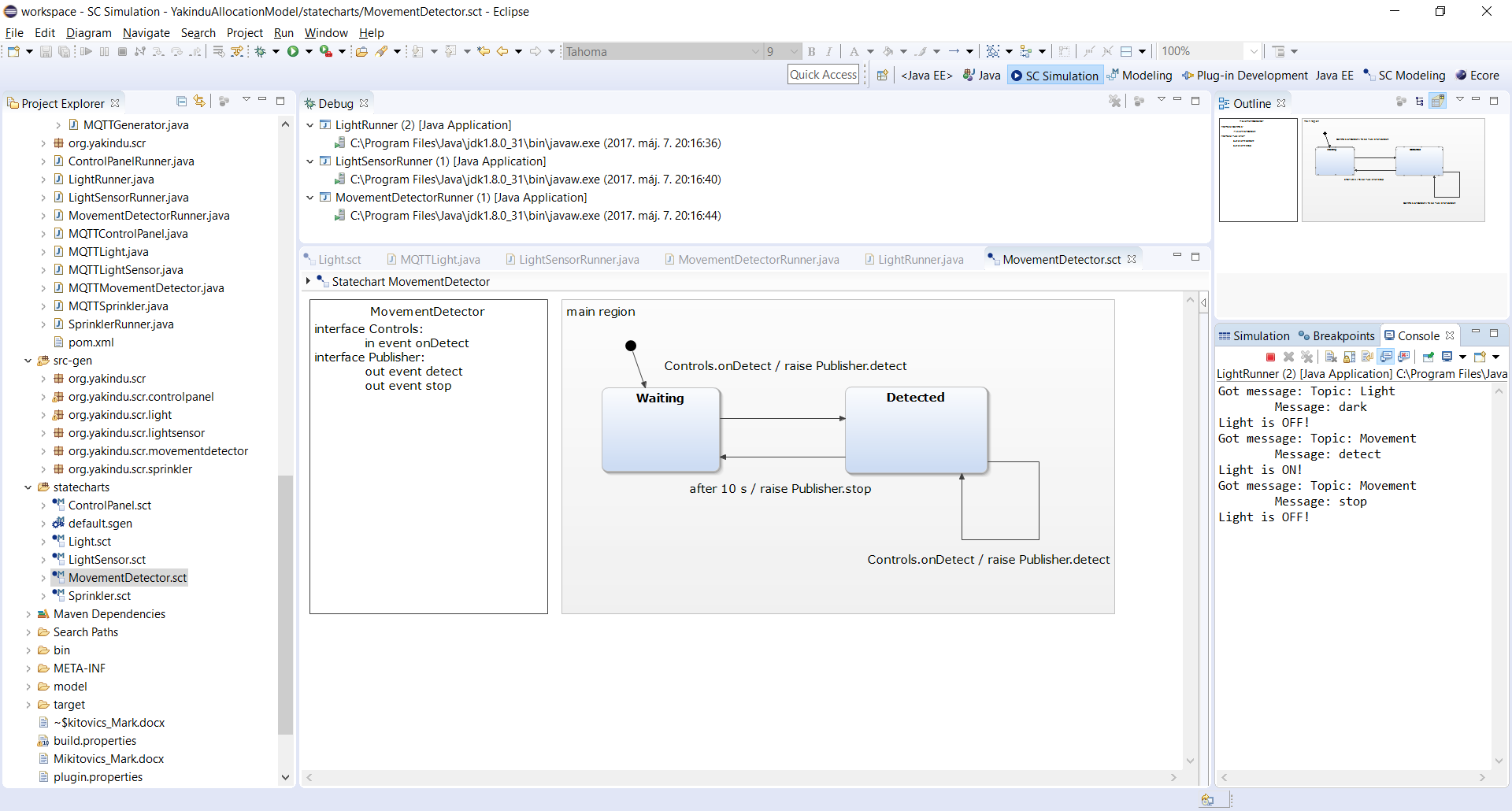
. ábra: A mozgást követő bekapcsolt állapot

Ahhoz, hogy a generált kódban is ugyanezt az esetet ki tudjam próbálni, módosítanom kellett az így előállított osztályokat. A **LightSensorRunner** osztályban az inicializálás után kiváltom a sötétséget jelző eseményt, a **MovementDetectorRunner** osztályban pedig a mozgás eseményt szimulálom hasonlóan. Ahhoz, hogy követni tudjam az **MQTTLight** belső állapotváltásait, kiegészítettem úgy, hogy minden beérkező üzenet hatására kiírja azt. Sajnos az nem kérhető le, hogy melyik állapot aktív, de az igen, hogy egy tetszőlegesen kiválasztott aktív-e. Ezért a következő változtatást hajtottam végre:

|  |
| --- |
| **public** **void** messageArrived(String topic, MqttMessage msg) **throws** Exception {  String message = **new** String(msg.getPayload());  System.***out***.println("Got message: Topic: " + topic + "\n\tMessage: " + message);  **if**(message.equals("switchLightOn") && topic.equals("ControlPanel")) {  switchOn();  }  **if**(message.equals("switchLightOff") && topic.equals("ControlPanel")) {  switchOff();  }  /\* további üzenetek ellenőrzése \*/  **if** (statemachine.isStateActive(State.***main\_region\_On***)) {  System.***out***.println("Light is ON!");  }  **if** (statemachine.isStateActive(State.***main\_region\_Off***)) {  System.***out***.println("Light is OFF!");  }  } |

. ábra: Bejövő üzenetek kezelése és a jelenlegi állapot ellenőrzése

Mivel egymás után indítottam a **LightRunner, LightSensorRunner** majd a **MovementDetectorRunner** osztályokat, így a szimulációval megegyező sorrendben következtek az események. A **LightRunner** kimenetén megfigyelhető az elvárt működés. A sötét üzenet megérkezésekor marad kikapcsolt állapotban a lámpa. Ezt követően megkérkezik a mozgásérzékelőtől az üzenet, melynek hatására a lámpa rögtön be is kapcsol. Továbbá az is észrevehető, hogy az **MQTTMovementDetector** helyesen, 10 másodperc elteltével jelzi, hogy megszűnt a mozgás, amire a lámpa ismét lekapcsol.



. ábra: A LightRunner kimenete

Miután megbizonyosodtam arról, hogy egy egyszerű esetben megfelelően működik a lámpa, megvizsgáltam a kapcsoló, a mozgás- és a fényérzékelő eseményeinek egymásra hatását. Fontos, hogy bármilyen sorrendben érkezzenek, mindig a kívánt állapotban legyen a feliratkozó állapotgép.

A további komplexebb tesztesetekhez készítettem egy különálló JUnit [6] tesztet, ahol az inicializálás után az egyes eseményeket időzítve váltottam ki, és közben figyeltem, hogy mindig az elvárt eredményt kapom-e. Minden teszt előtt lefuttatom az inicializáló részt, ami lényegében az MQTT állapotgépeket futtató állományokban található inicializálások összessége. Ezen kívül minden teszt után lefuttatok egy kódrészletet, amely lecsatlakoztatja a klienseket a brókerről.

|  |
| --- |
| lightSensorClient.onBright(); // Switch on, bright, movement  Thread.*sleep*(200);  *assertTrue*(lightClient.statemachine.isStateActive(State.***main\_region\_On***));    Thread.*sleep*(11000);    //Switch Off, only one true  controlClient.switchLight(); // Switch off, bright, no movement  Thread.*sleep*(200);  *assertFalse*(lightClient.statemachine.isStateActive(State.***main\_region\_On***)); |

. ábra: Különböző tesztesetek

Ugyanígy elkészítettem az automatikus tesztet a locsolóhoz is, annak, és a kapcsolónak az időzítéseit is figyelembe véve, szintén az elvárt működést produkálta.

# Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | „Yakindu Statechart tools,” [Online]. Available: https://www.itemis.com/en/yakindu/state-machine/. |
| [2] | „MQTT Version 3.1.1 Plus Errata 01.,” OASIS Standard Incorporating, 10 12 2015. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html. |
| [3] | „Eclipse Paho,” 2017. [Online]. Available: http://www.eclipse.org/paho/. |
| [4] | „Mosquitto, An Open Source MQTT v3.1/v3.1.1 Broker,” [Online]. Available: https://mosquitto.org/. |
| [5] | „Eclipse Modeling Project,” [Online]. Available: http://www.eclipse.org/modeling/emf/. |
| [6] | „JUnit 4,” 2017. [Online]. Available: http://junit.org/junit4/. |

# Függelék

A 2. fejezet példája a <https://github.com/mikitovicsmark/MQTTStateMachine> címen érhető el, illetve az esettanulmány és a teljes forráskód megtalálható a <https://github.com/mikitovicsmark/MQTTConfig> url alatt.

A projekteket Eclipse Modeling Neon verziójával készítettem. Az állapotgépek szerkesztéséhez és a kódgeneráláshoz szükséges Yakindu Statechart tools telepítése, valamint a függőségek kezeléséhez Maven plugin.

Mindkét projekt futtatásához 1.8-as verziószámú Java futtatókörnyezetre van szükség.