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

**Budapesti Műszaki és Gazdaságtudományi Egyetem**

Villamosmérnöki és Informatikai Kar

Pogácsasütöde Tanszék

Hálózati interfész   
modell alapú generálása

Készítette

Mikitovics MárkKonzulens

Semeráth Oszkár

2017

Tartalomjegyzék

[Összefoglaló 4](#_Toc480916760)

[Abstract 5](#_Toc480916761)

[1. Bevezetés 6](#_Toc480916762)

[2. Egyszerű funkcionális példa megvalósítása 7](#_Toc480916763)

[2.1. Yakindu Statechart tools [1] 7](#_Toc480916764)

[2.1.1. Kiberfizikai rendszer állapotgépekkel 7](#_Toc480916765)

[2.1.2. Osztályok generálása 8](#_Toc480916766)

[2.2. Message Queue Telemetry Transport (MQTT [2]) és Paho [3] 9](#_Toc480916767)

[2.2.1. MQTT protokollt használó osztály 10](#_Toc480916768)

[2.2.2. MQTT kompatibilis állapotgép futtatása 12](#_Toc480916769)

[3. Esettanulmány 14](#_Toc480916770)

[3.1. A kiberfizikai rendszer komponensei 14](#_Toc480916771)

[3.2. Konfigurációs fájl elkészítése 17](#_Toc480916772)

[4. Irodalomjegyzék 19](#_Toc480916773)

[5. Függelék 20](#_Toc480916774)

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, 2017.05.12.

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.

# Egyszerű funkcionális példa megvalósítása

## Yakindu Statechart tools [1]

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

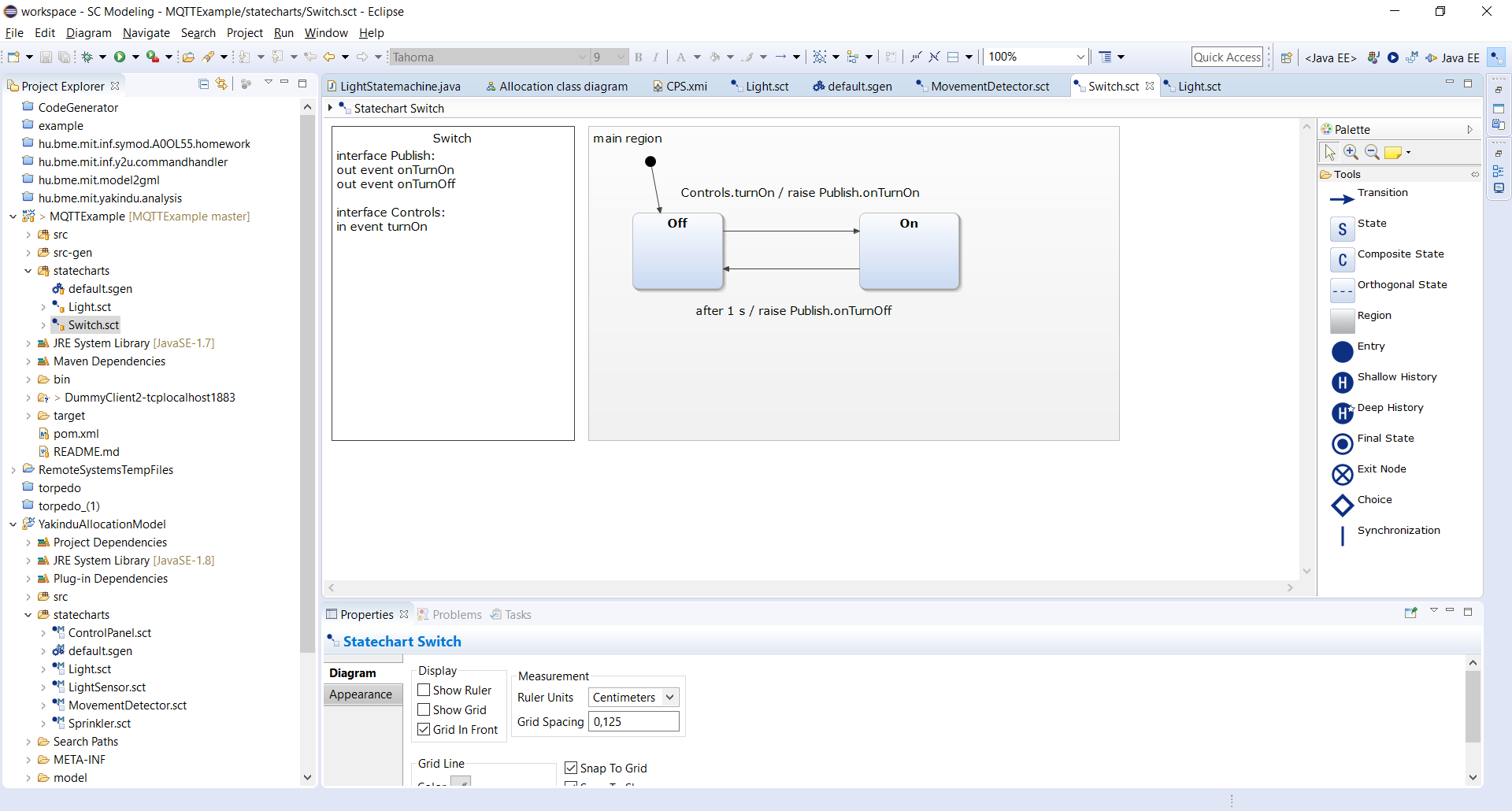
A feladatot az eszköz megismerésével kezdtem, egy kezdetleges példát készítettem el, amely rendelkezett a végső kiberfizikai rendszer minden szükséges funkcionalitásával. Ez olyan állapotgépek létrehozását követelte meg, amelyek képesek külső események hatására belső állapotot váltani, állapotváltásaik 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.

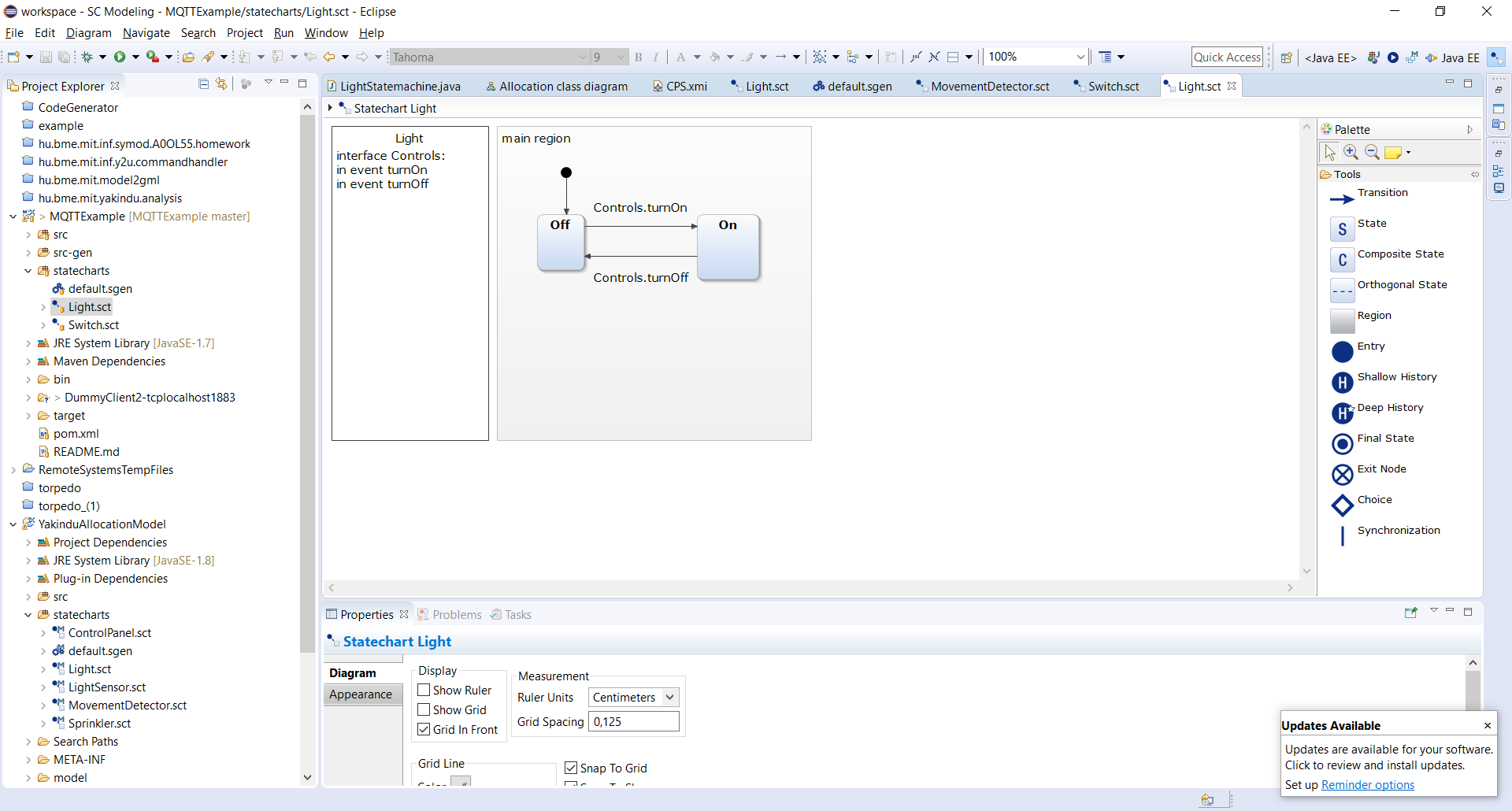
Általánosan elmondható, hogy minden állapotgép két interfésszel rendelkezik. A bejövő eseményeket a **Controls**, a kimenőket a **Publish** tartalmazza. A későbbiekben ezen események összekötésével lesznek képesek a komponensek a kommunikációra.

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

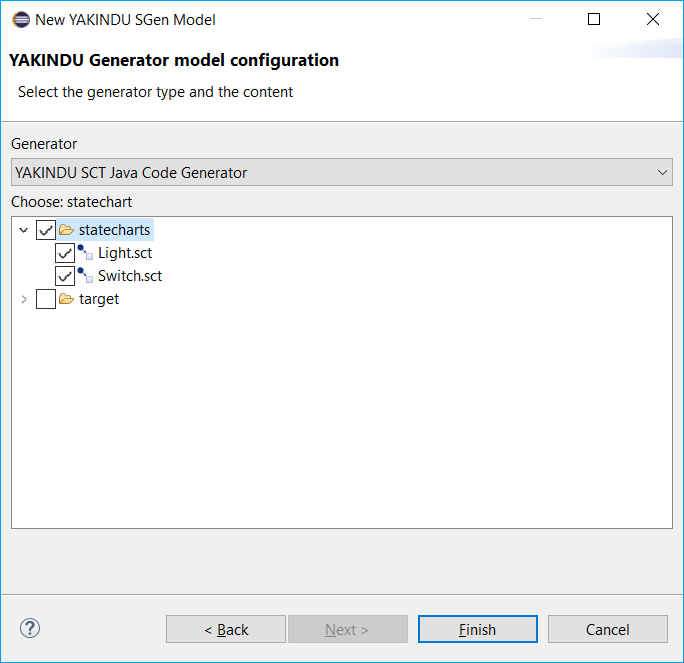
Ezt követően a lámpa osztály Yakindu állapotgépes implementációját hajtottam végre. Láthatóan sokkal egyszerűbb, csak **Controls** interfésszel rendelkezik, ugyanis ő nem oszt meg információt a külvilággal. Szintén leolvasható az ábráról, hogy a kezdeti állapota az **Off**. A **Controls** **turnOn** 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.



. ábra: Generator Model

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

}

}

}

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 [2]) és Paho [3]

Az MQTT 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 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 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>

### 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**;

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**;

}

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();

}

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**;

}

}

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

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"

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();

}

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) {

System.***out***.println("reason "+me.getReasonCode());

System.***out***.println("msg "+me.getMessage());

System.***out***.println("loc "+me.getLocalizedMessage());

System.***out***.println("cause "+me.getCause());

System.***out***.println("excep "+me);

me.printStackTrace();

}

}

# Esettanulmány

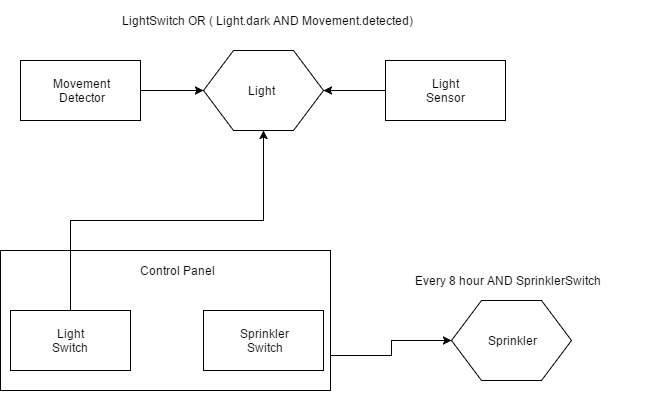
Az egyszerű, de funkcionális példa elkészítésével végig jártam azokat a lépéseket, melyek egy összetettebb kiberfizikai rendszer elkészítéséhez szükségesek. Azonban nem lenne célszerű minden állapotgéphez kézzel elkészíteni az MQTT használatához szükséges függvényeket, egy konfiguráció elkészítésével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

## A kiberfizikai rendszer komponensei

Az esettanulmányhoz egy kerti rendszert terveztem. Ennek részei:

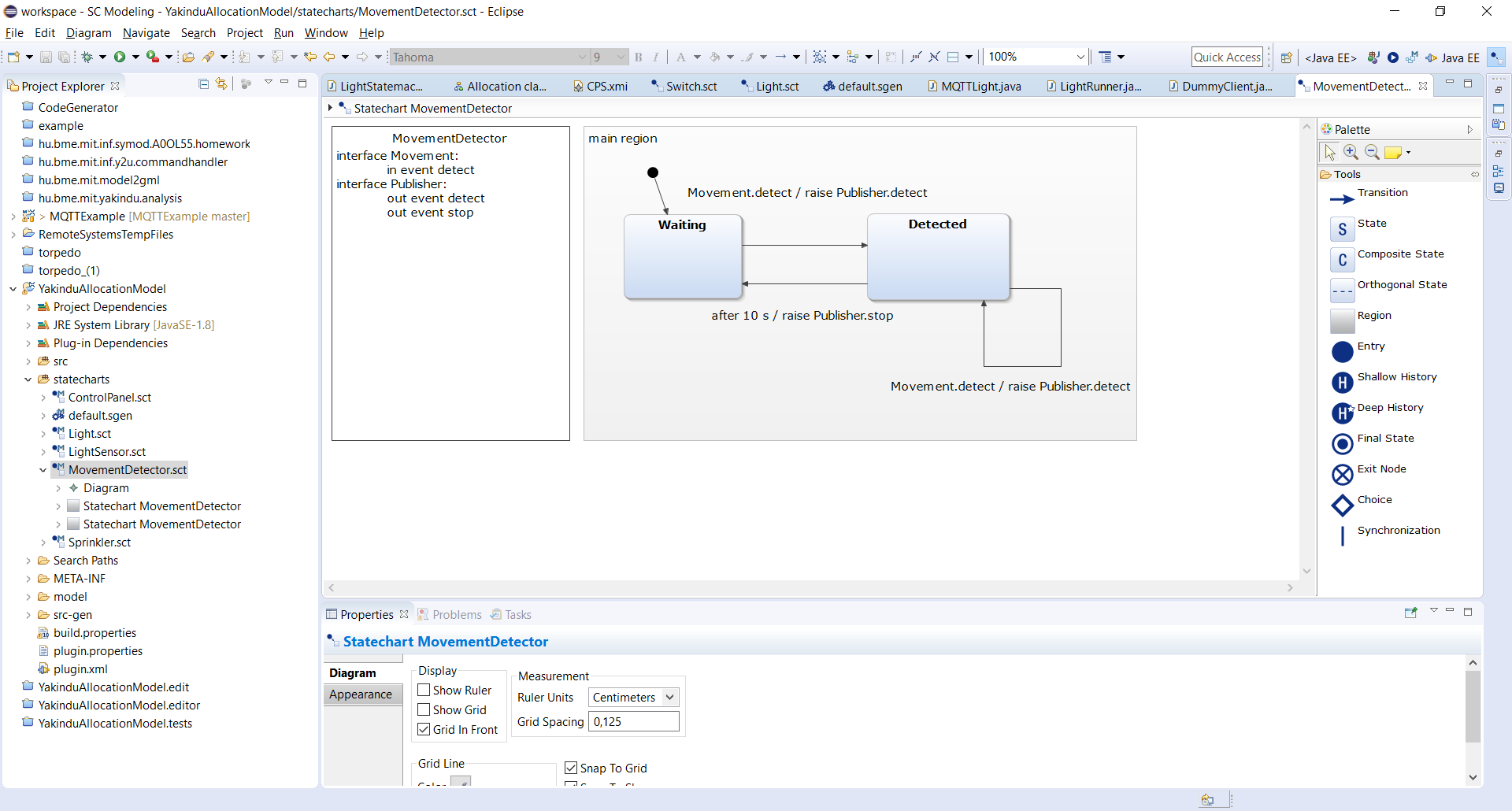
* mozgásérzékelő
* fényérzékelő
* lámpa
* locsoló
* vezérlőpanel, mely tartalmaz egy kapcsolót a lámpához és a locsolóhoz

A kiberfizikai rendszerben két subscriber található, a lámpa és a locsoló. A többi komponens publisherként viselkedik. A lámpa világít, amennyiben a hozzá tartozó kapcsoló be van kapcsolva, vagy sötét van, de mozgást érzékelt a rendszer. A locsoló minden nyolcadik órában működésbe lép, vagy akkor, ha a vezérlőpanelen található kapcsolót használják.



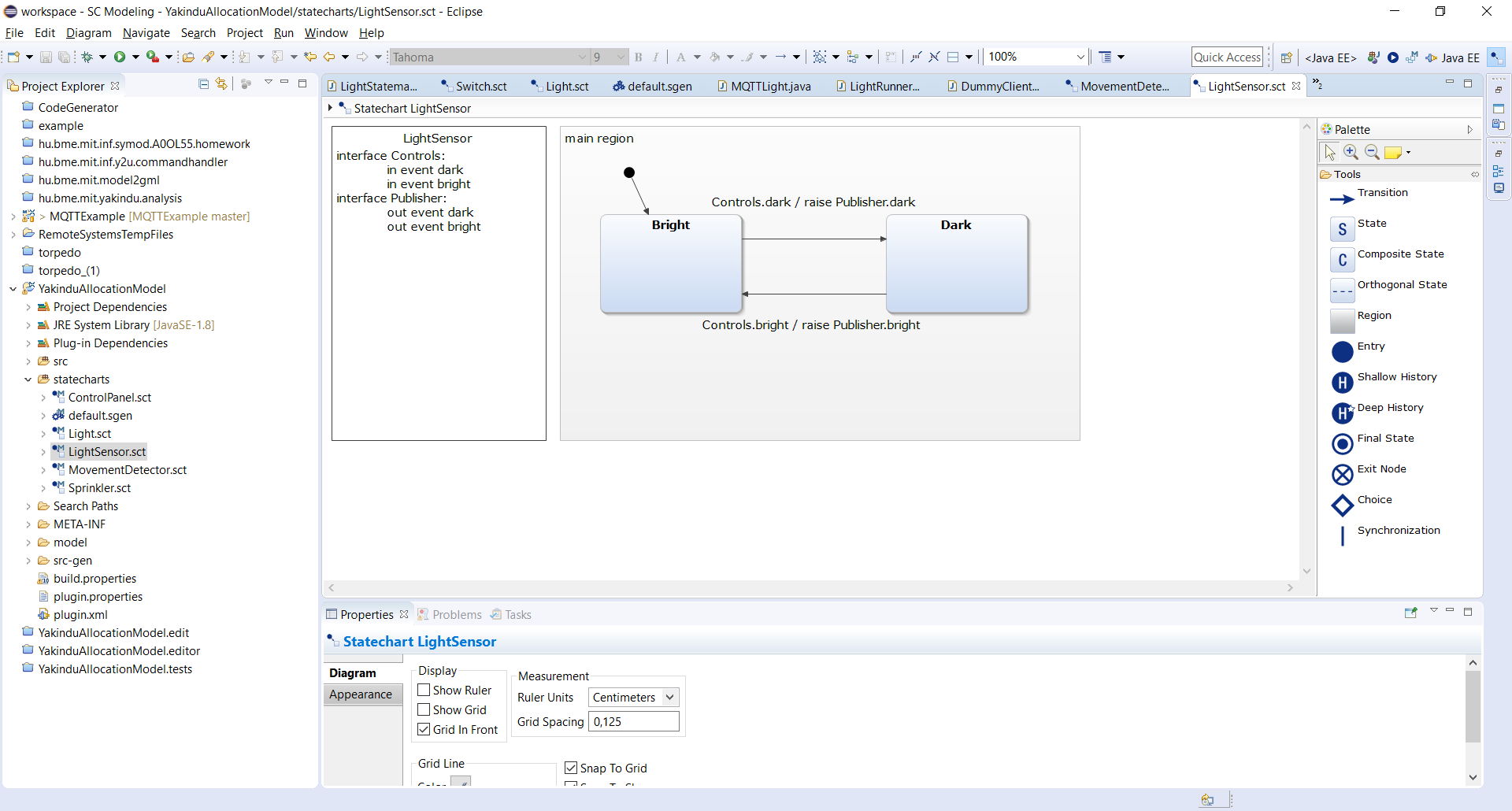
. ábra: A rendszer vázlata

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.



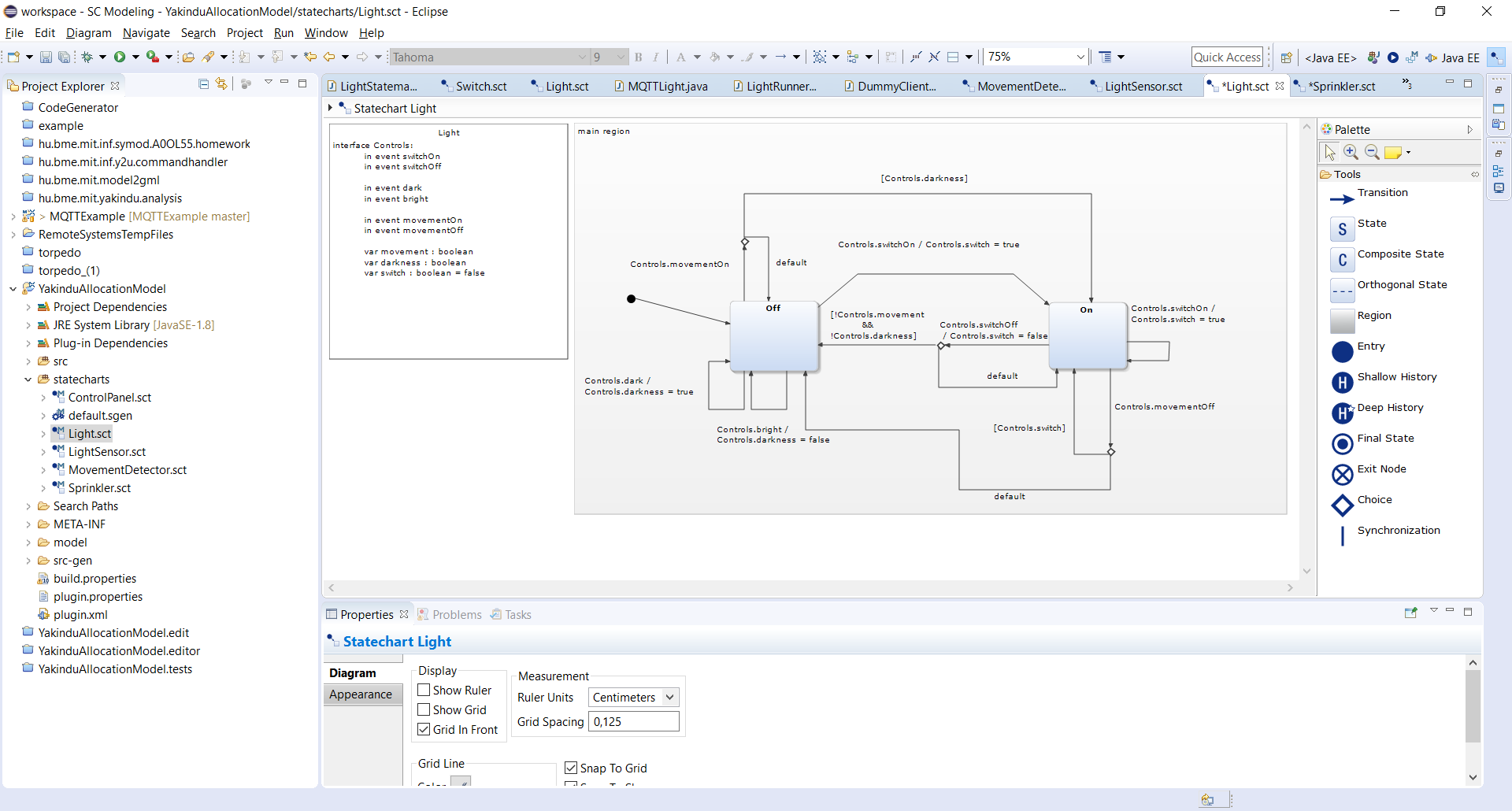
. ábra: A mozgásérzékelő blokkváza

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



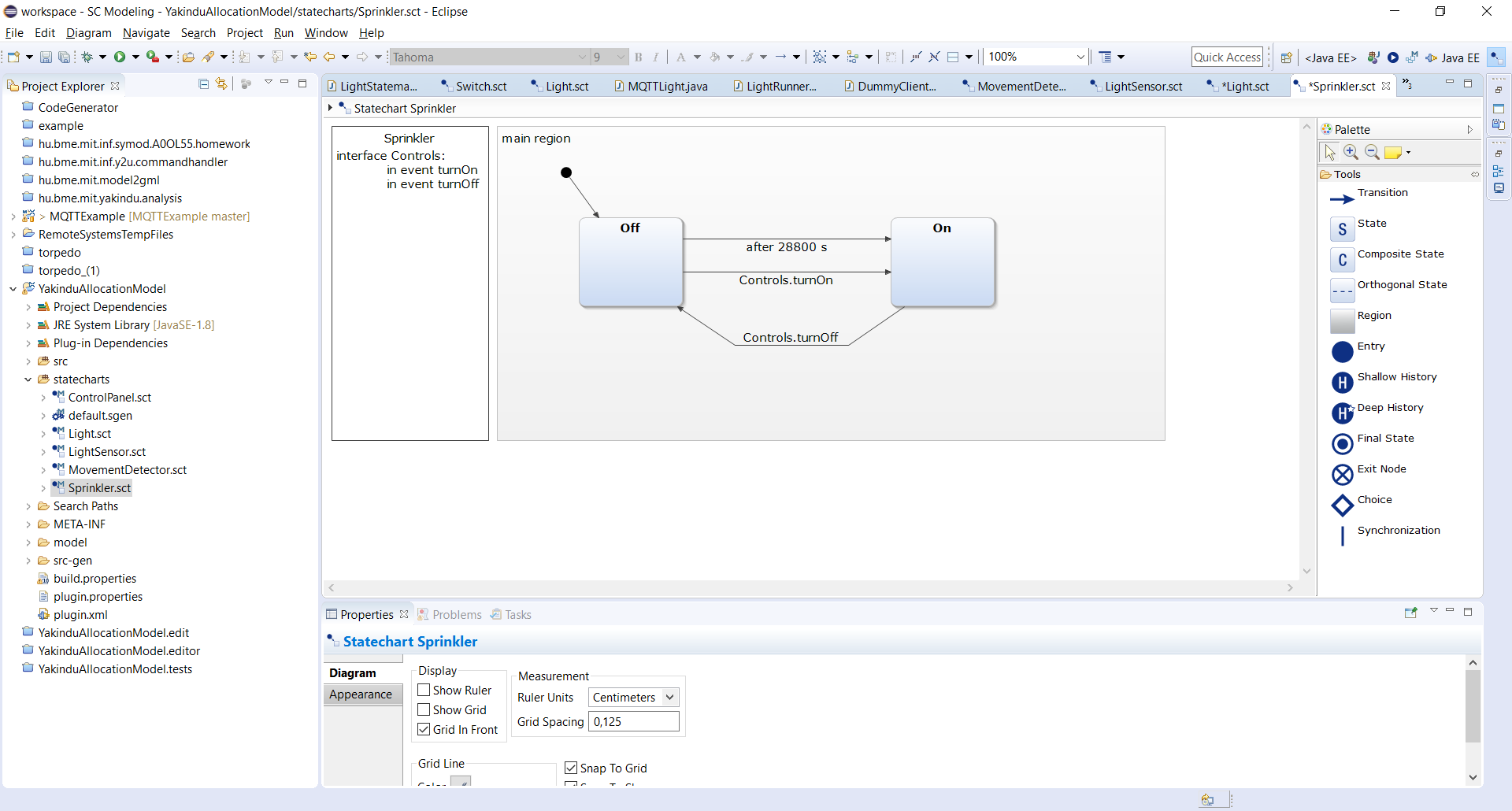
. ábra: A fényérzékelő blokkváza

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.



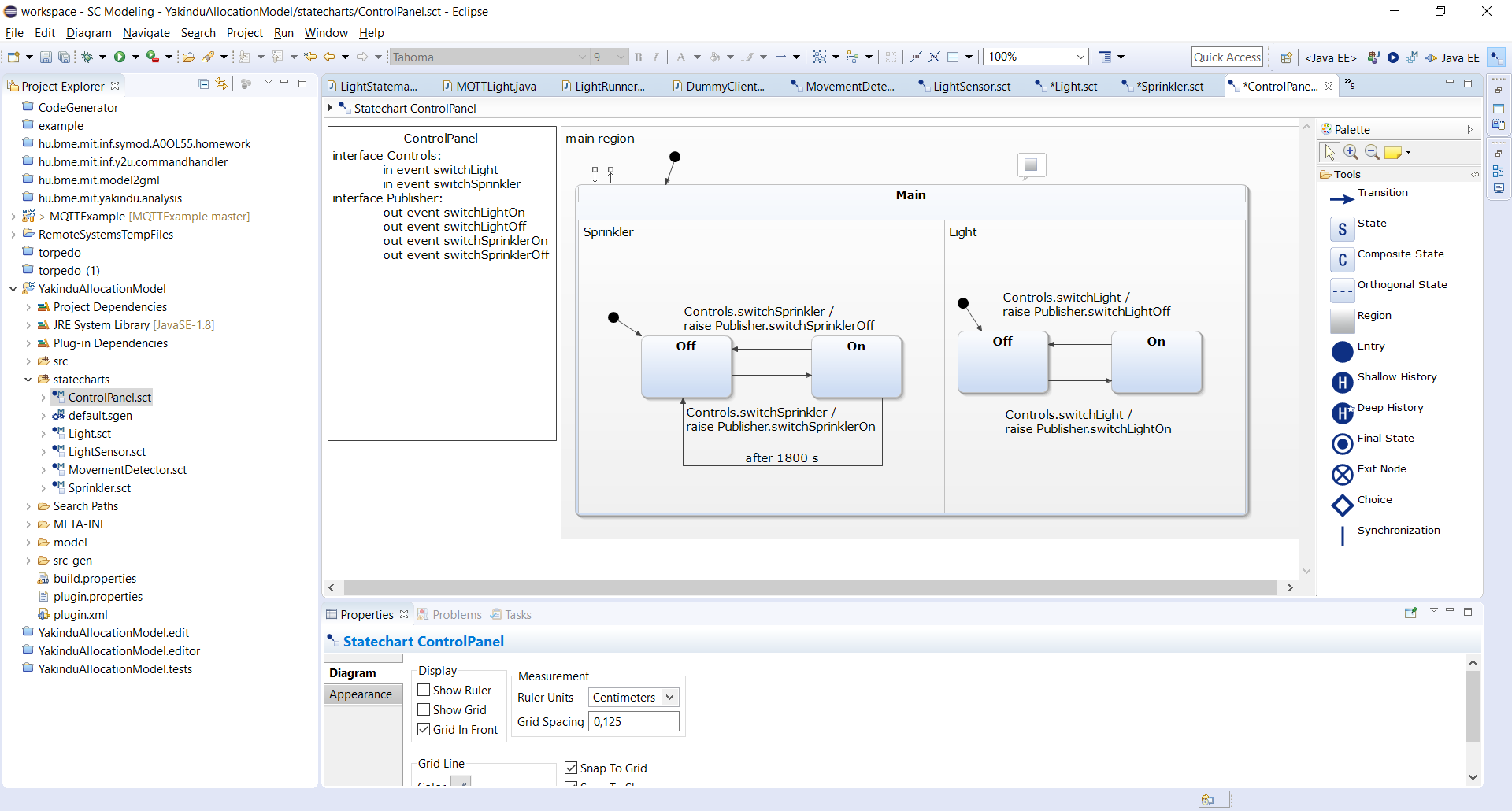
. ábra: A lámpát reprezentáló állapotgép

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.



. 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

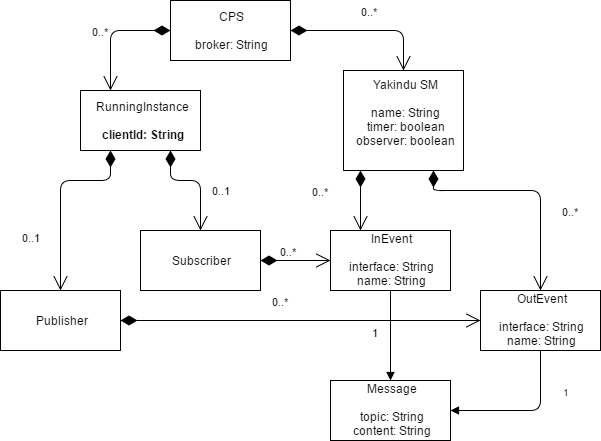
## Konfigurációs fájl elkészítése

Ahhoz, hogy ne kelljen kézzel elkészíteni minden állapotgépből származtatott osztályt, valamint a köztük lévő interfészek kapcsolatát implementálni, egy konfigurációs fájl létrehozása tűnt jó megoldásnak. 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.

A kiberfizikai rendszer (CPS) tárolja a közös bróker címét. Futtatható állományokból áll, melyekhez Yakindu állapotgépek tartoznak. A futtatható komponensekhez tartozhat publisher, subscriber vagy akár mindkettő is. Ezek tartalmazzák a Yakindu állapotgépek bejövő, vagy kimenő eseményeit egy adott interfészhez. Fontos, hogy klienseknek egyedi azonosítóval kell rendelkezniük.

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. Az állapotgépekben bejövő és kimenő események találhatók tetszőleges számban.

A legfontosabb, amiért szükséges a konfigurációs fájl, pedig az üzeneteket jelképező entitás, ami összekapcsolja a kimenő és bejövő eseményeket.



. ábra: Az allokációs diagram egyszerűsített változata

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

# 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/. |

# Függelék

A függelék szövege.