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

**Budapesti Műszaki és Gazdaságtudományi Egyetem**

Villamosmérnöki és Informatikai Kar

Hálózati interfész   
modell alapú generálása

Készítette

Mikitovics MárkKonzulensek

Vörös András

tudományos segédmunkatárs

Semeráth Oszkár

tudományos segédmunkatárs

2017

Tartalomjegyzék

[Kivonat 5](#_Toc482963934)

[Abstract 6](#_Toc482963935)

[1. Bevezetés 7](#_Toc482963936)

[1.1. Problémafelvetés 7](#_Toc482963937)

[1.2. Célkitűzés 7](#_Toc482963938)

[1.3. Kontribúció 8](#_Toc482963939)

[1.4. Hozzáadott érték 8](#_Toc482963940)

[1.5. Dolgozat felépítése 8](#_Toc482963941)

[2. Előismeretek 9](#_Toc482963942)

[2.1. Állapotgép modellezés Yakindu Statechart Tools segítségével 9](#_Toc482963943)

[2.1.1. Kiberfizikai rendszer komponenseinek modellezése 10](#_Toc482963944)

[2.1.2. Automatikus kódgenerálás állapotgépekből 11](#_Toc482963945)

[2.2. Message Queue Telemetry Transport kommunikációs protokoll 12](#_Toc482963946)

[2.3. Apache Maven 12](#_Toc482963947)

[2.4. MQTT bróker használata 12](#_Toc482963948)

[3. Összetett kiberfizikai rendszer modellezése 14](#_Toc482963949)

[3.1. Kommunikációs és allokációs modell 14](#_Toc482963950)

[4. A rendszert leíró állapotgépek 17](#_Toc482963951)

[5. A kiberfizikai rendszer konfigurációja és az erre alapuló kódgeneráló 21](#_Toc482963952)

[5.1. A rendszer komponensei 21](#_Toc482963953)

[5.2. A rendszerben található topikok 21](#_Toc482963954)

[5.3. A kódgeneráló működése 22](#_Toc482963955)

[6. A kommunikációért felelős generált kód bemutatása 25](#_Toc482963956)

[6.1. MQTT protokollt használó osztály 25](#_Toc482963957)

[6.2. A programkódot futtató osztályok bemutatása 27](#_Toc482963958)

[7. Előzetes tesztelés 29](#_Toc482963959)

[7.1. Unit tesztek 29](#_Toc482963960)

[7.2. Kommunikációs tesztek 31](#_Toc482963961)

[7.3. Integrációs tesztek 32](#_Toc482963962)

[7.4. Tesztlefedettség vizsgálata 33](#_Toc482963963)

[8. Összefoglaló 34](#_Toc482963964)

[8.1. Eredmények 34](#_Toc482963965)

[8.2. Feladatok teljesítése 34](#_Toc482963966)

[8.3. A rendszer fejlesztésének lehetőségei 35](#_Toc482963967)

[9. Irodalomjegyzék 36](#_Toc482963968)

[10. Függelék 37](#_Toc482963969)

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

# Kivonat

A manapság széles körben elterjedt kiberfizikai rendszerek olyan hardverekkel és szoftverekkel rendelkező egységek, melyek segítségével az élet változatos területein oldhatók meg problémák legyen szó akár önvezérelt járművekről, akár gyártósorokról. A modellalapú szoftvertervezés a számítástechnikán belül egy jól bevált szokás, amely magasabb produktivitást és jobb minőségű munkát eredményez.

A gyakorlatban jelenleg kevés olyan modellező eszköz érhető el, melynek segítségével állapotgépekkel is leírható kiberfizikai rendszereket kényelmesen tervezhetnénk. A komponensek egymással való kommunikációja, valamint hozzájuk tartozó interfészek leírása kényelmetlen.

Dolgozatom célja ezen probléma megoldása, egy létező, állapotgépeket leíró eszköz olyan szintű kibővítése, hogy az alkalmas legyen a komponensek összekötésére, és azok közti kommunikációra.

Elkészítettem egy modellezési módszert, amellyel egy kiberfizikai rendszerek állapotgépes leírására alkalmas formalizmust adok meg. Továbbá a modellezési nyelvhez elkészítettem egy kódgenerátort, ami által futtatható java nyelvű programkód generálható. A kódgenerátor működését egy esettanulmányon keresztül szemléltetem, majd ezt követően tesztekkel ellenőrzöm annak helyességét.

Az általam nyújtott megoldással kibővített modellező eszközzel és kódgenerátorral állapotgépekkel leírható, moduláris, könnyen módosítható kiberfizikai rendszer tervezhető. Ennek következményeként olyan részei is automatikusan generálhatók egy alkalmazásnak, amelyet eddig manuálisan kellett elvégezni. Az eszköz segítségével egyszerűen újra konfigurálható a rendszer és a karbantarthatóság is javul.

# Abstract

Today’s widespread cyber-physics systems help to solve problems in diverse areas of our lives, ranging from self-guided vehicles to production lines. They include software and hardware units, which observe their surroundings, and respond to them, to complete their tasks. Model-based software design is a well-established custom in computer science, which results in higher productivity and better quality work.

There are currently few available modelling tools currently, which can design cyber-physical systems, that can be described with state-machines. The description of the communication and the associated interfaces between the components is inconvenient.

The purpose of my thesis is to solve this problem. I extended an existing state-machine engineering tool for this task. The connection and the communication between the components is easily solved this way.

I have prepared a modelling method that gives me a formalism to the description of cyber-physical systems with state-machines. Additionally, I have created a code generator for the modelling language, which generates a runnable java program. I will demonstrate the operation of the code generator through a case study and then check the accuracy of it with tests.

With the help of this extended modelling tool, it is possible to create modular, easily modifiable cyber-physical systems, that can be described with state-machines. Therefore, the application can be automatically generated. With this tool, it is easy to reconfigure the system, and it also comes with improved maintainability.

# Bevezetés

Napjainkban a kiberfizikai rendszerek széles körben elterjedtek. Olyan hardverekkel és szoftverekkel rendelkeznek, amelyek együttműködve változatos problémákra képesek önműködő megoldást nyújtani, legyen szó akár önvezérelt járművekről, akár gyártósorokról. Fontos tulajdonságuk, hogy valós idejű megfigyeléseket végeznek, amelyekhez alkalmazkodik az egész rendszer, hogy a feladatát teljesítse.

A számítástechnika egyik meghatározó területe a modellezés. Lehetőséget ad absztrakt problémák leírására tudományok széles spektrumán. Feladatokat szimulálhatunk, oldhatunk meg anélkül, hogy költségesebb, valós erőforrásokkal kellene kísérleteznünk.

A modellezéssel szorosan összefüggő témakör a kódgenerálás. Segítségével a modelljeink könnyen ellenőrizhetővé válnak. Egy új konfiguráció által akár az egész rendszerre kiterjedő változtatásaink eredményét is hamar kézhez kapjuk. Nagyban javítja a fejlesztő produktivitását, illetve a szoftver minőségét is pozitívan befolyásolja. A tervezés során olyan hibákra is fény derülhet a beépített validációval, amelyek egyébként csak futási időben jönnének elő.

Állapotgépekkel [1] könnyen leírhatunk véges számú állapottal rendelkező eszközöket. Összetett rendszer esetén is érthetően reprezentálható azok működése. Egy ilyen szoftver a Yakindu Statechart Tools [2] is, mellyel foglalkozok a szakdolgozatom során.

## Problémafelvetés

Azonban a gyakorlatban jelenleg nem érhető el jól használható modellező eszköz, melynek segítségével állapotgépekkel is leírható kiberfizikai rendszereket tervezhetnénk. A Yakindu Statechart Tools nem ad lehetőséget arra, hogy a tervezés során több hasonló példányt készítsünk egy állapotgépből, amelyek csak konfiguráció szintjén térnek el. Ehhez le kell másolnunk egy komponenst, majd módosítani az adott paramétereit.

A kiberfizikai rendszer fontos tulajdonsága, hogy annak részei egymással hálózatot alkotva kommunikációra képesek [3]. Állapotgépek esetén nincs rá módunk, hogy meghatározzuk, hogy azok milyen interfészeken kapcsolódnak össze, illetve, hogy egyes állapotgépek mely hardverekhez tartoznak.

## Célkitűzés

Dolgozatom célja, hogy kiegészítsem a már említett és jól bevált modellezőeszközt, hogy képes legyen kiberfizikai rendszer modellezésére is. Ehhez szükséges, hogy a kiberfizikai rendszer hálózatát is lehessen modellezni. A célom ezeken felül olyan újra felhasználható állapotgépek megalkotása, amelyek egy rendszerként képesek működni, a rendszeren belül azonos működéssel, de eltérő konfigurációval több példánnyal is rendelkezhetnek.

## Kontribúció

Ebben a dolgozatban bemutatok egy általam elkészített új modellezési módszert, amivel össze lehet kötni különböző viselkedésmodellből felépített komponenseket. Ezek az elemek egy kiberfizikai rendszeren belül többször, eltérően definiált működéssel is felhasználhatók. A állapotgépekből álló kiberfizikai rendszer komponenseit kibővítettem úgy, hogy hálózati kommunikációra legyenek képesek a rendszeren belül tetszőleges számú résztvevővel.

Egy olyan formalizmust adok meg, amellyel állapotgépekből álló kiberfizikai rendszerek leírhatók. Elkészítettem egy olyan kódgenerátort, amelynek segítségével ebből a modellből futtatható kód generálható. A komponenseket felkészítettem Message Queue Telemetry Transport (MQTT) [4] üzenetküldő protokollon való kommunikációra. A leíró nyelv elkészítését és a kódgenerátor működését, valamint az az által létrehozott programok tesztelését egy esettanulmányon keresztül mutatom be.

## Hozzáadott érték

A megoldásom segítségével egy állapotgépekkel leírható, moduláris, könnyen bővíthető kiberfizikai rendszert leíró modellezési nyelvet készítettem. A kiberfizikai rendszer elemeinek konfigurációi, azoknak kommunikációs interfészei egyszerűen módosíthatóvá váltak.

Az általam nyújtott megoldással olyan problémákat is meg lehet oldani, amiket a Yakindu Statechart Tools eszközkészletével nem. A Yakinduval hasonló, azonban eltérő kommunikációs interfészekkel rendelkező állapotgépek modellezése esetén ugyan azt többször kellene elkészíteni.

## Dolgozat felépítése

A dolgozat során a 2. fejezetben bemutatom azokat az előismereteket, amelyekre alapozva megterveztem és elkészítettem a modellezési nyelvet. Egy egyszerű példán keresztül megvalósítok egy olyan rendszert, amely rendelkezik minden olyan funkcionalitással, amivel a generált kódnak is rendelkeznie kell végül (**Hiba! A hivatkozási forrás nem található.**. fejezet). Ezt követően egy esettanulmányon keresztül szemléltetem a formalizmust, amellyel a kiberfizikai rendszert alkotó komponenseket el lehet készíteni (3. fejezet). Az rendszert alkotó állapotgépek Yakindu Statechart Tool segítségével történő leírása után elkészítek egy olyan konfigurációs állományt, amely tartalmaz minden olyan információt, amely szükséges a működő programkód generálásához (5. fejezet). A 7. fejezetben az így előállított rendszert ezután tesztekkel ellenőrzöm, bemutatva annak helyes működését.

# Előismeretek

Ebben a fejezetben bemutatom a dolgozat megértéséhez szükséges technológiákat. Az állapotgépek elkészítéséhez, és az azokból származó osztályok generálásához elengedhetetlen Yakindu Statechart Tools fejlesztőeszközt. Ezt követően a kommunikációs protokollról, a Message Queue Telemetry Transport-ról (MQTT) [4] és annak nyílt forráskódú implementációjáról, a Paho [5] projektről lesz szó.

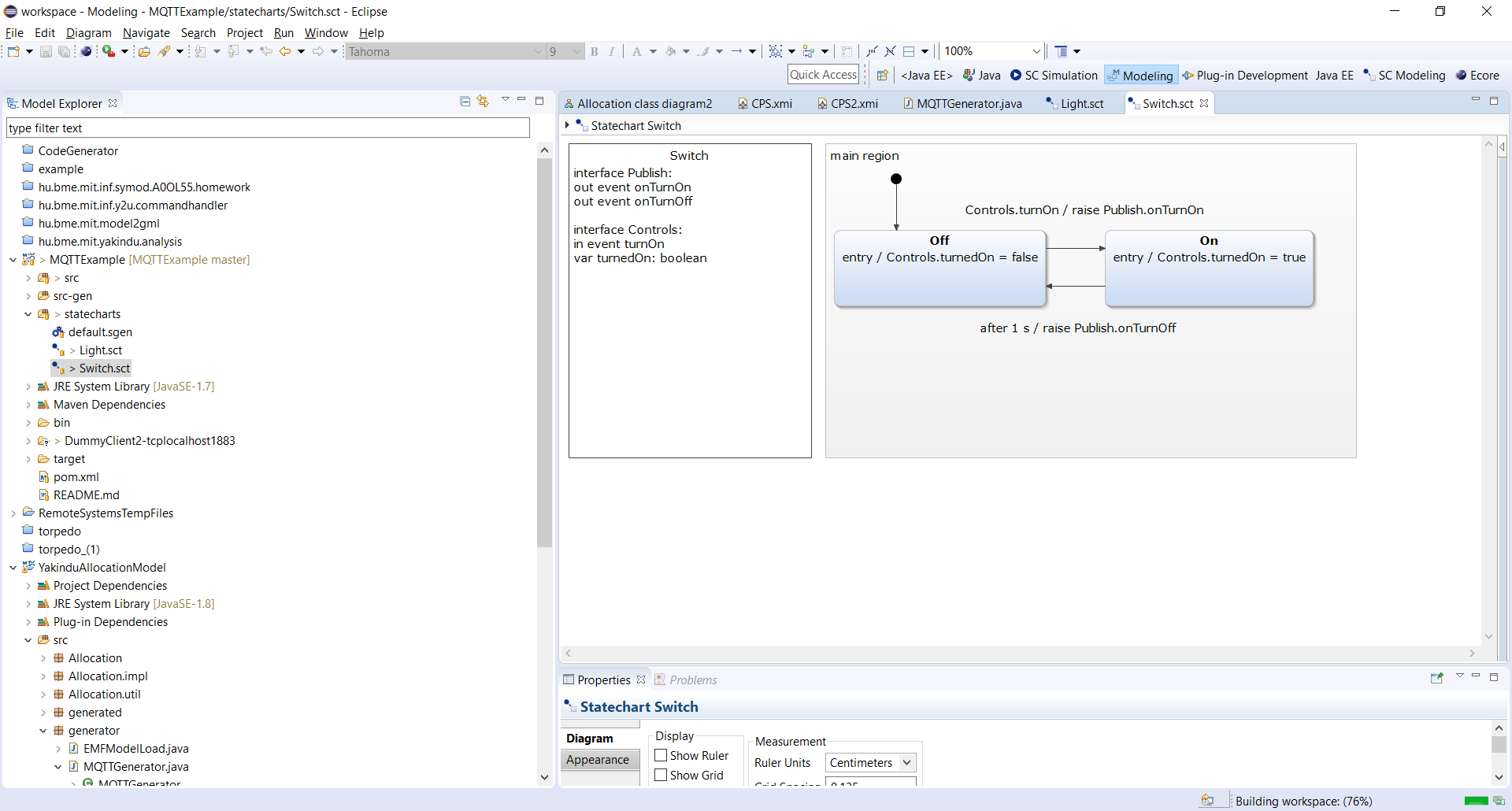
## Állapotgép modellezés Yakindu Statechart Tools segítségével

A Yakindu Statechart Tools [2] használatával az általam megtervezett kiberfizikai rendszer állapotgépekkel való leírása volt a cél.

A Yakindu Statechart tools egy olyan grafikus és szöveges modellező nyelv, amellyel állapotdiagramokat lehet leírni. Az állapotokat és átmeneteket grafikus elemekkel lehet elkészíteni, a leírásokat és a tevékenységeket pedig szövegesen.

Minden állapotgép rendelkezik egy kezdeti állapottal, erre mutat a fekete pötty. A régiókat dobozok jelölik, melyeknek neve felül olvasható. Ezek között az átmeneteket nyilak reprezentálják.

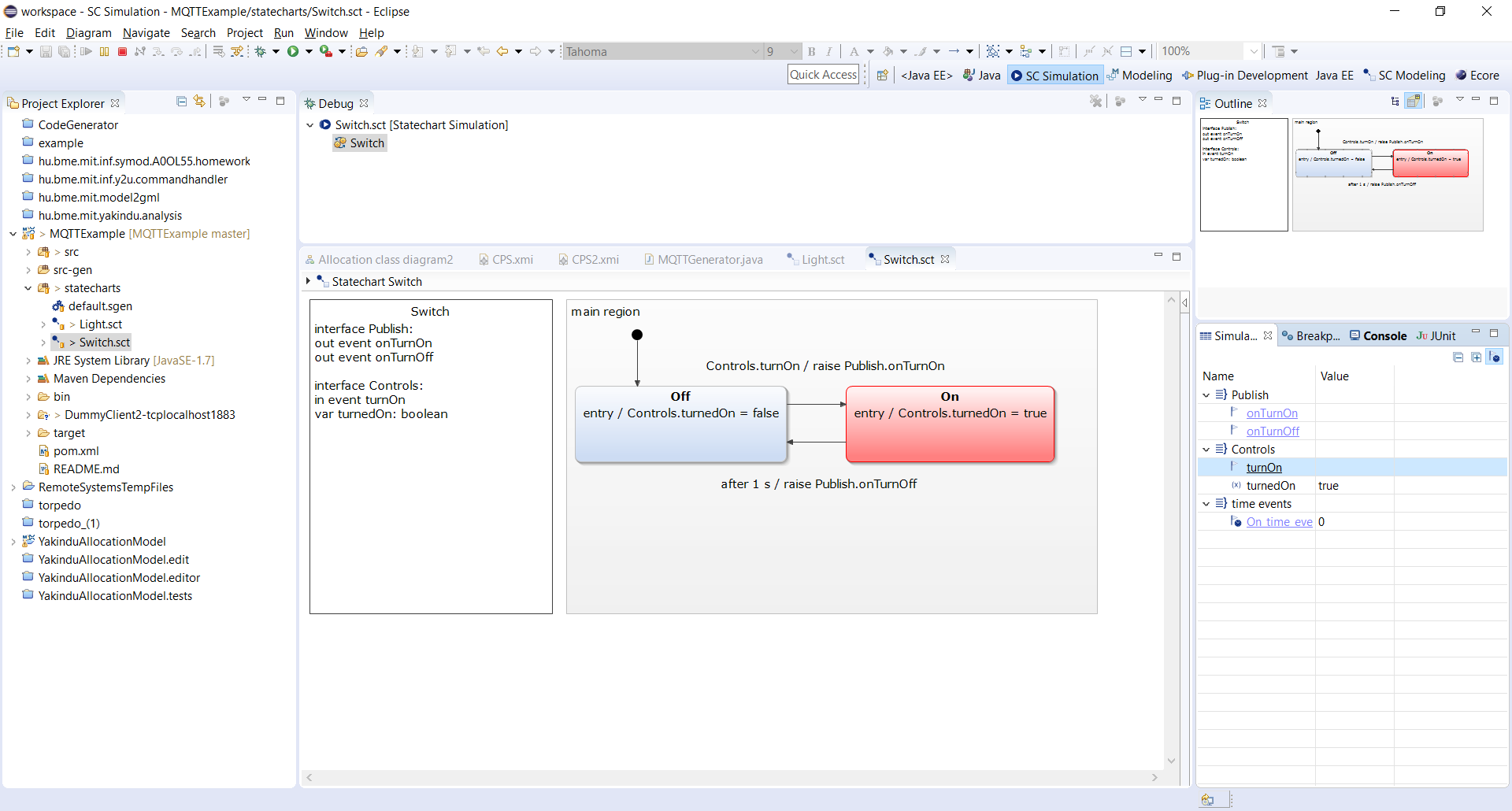
Az 1. ábra melletti területen szövegesen definiálhatók interfészek, hozzájuk tartozó kimenő és bejövő események, valamint változók is. Ezeket lehet felhasználni állapotváltásokhoz, vagy akár ki is lehet váltani ilyenkor eseményeket. Lehetőségünk van az állapotokba érkezéskor is ilyen tevékenységeket végrehajtani. Állapotváltásokat kiválthatunk időzítve is, erre egy példa az **after** kulcsszó. Az alapvető funkciókon kívül használtam még elágazásokat, valamint osztott belső állapotokat is.



1. ábra: Egy Yakindu Statechart toolssal készített egyszerű állapotgép

Az állapotgépek elkészítésekor további hasznos funkciók is segítik a fejlesztést. Az eszköz folyamatosan validálja az állapotgépet, figyelmeztet, ha nem elérhető állapotot hozok létre, vagy elírok egy kifejezést.

A tesztelés során számos hiba felfedezhető a szimulációval (2. ábra) is, amely jelzi, hogy mely állapotban tartózkodunk, illetve mutatja a belső változók értékét is. Az interfészeken definiált eseményeket kézzel kiválthatjuk, és a paramétereket is állíthatjuk közben.



2. ábra: Szimuláció során kiválthatunk eseményeket, megfigyelhetjük a változók értékeit is

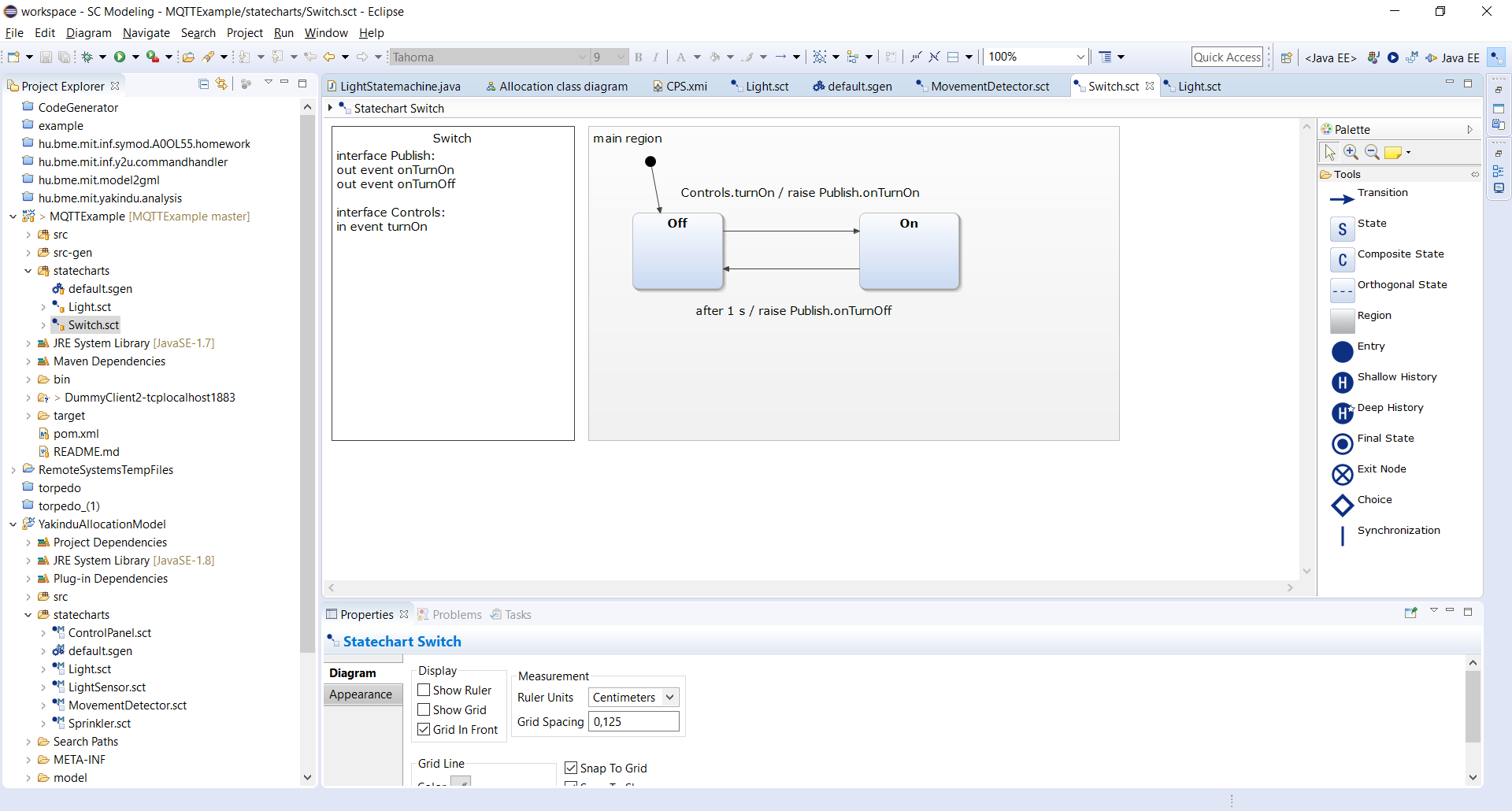
A feladatot az eszköz megismerésével kezdtem, egy kezdetleges példát készítettem el, amely rendelkezett a végső kiberfizikai rendszer minden szükséges funkcionalitásával. Ez olyan állapotgépek létrehozását követelte meg, amelyek képesek külső események hatására belső állapotot váltani, állapotváltásaik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

### Kiberfizikai rendszer komponenseinek modellezése

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.

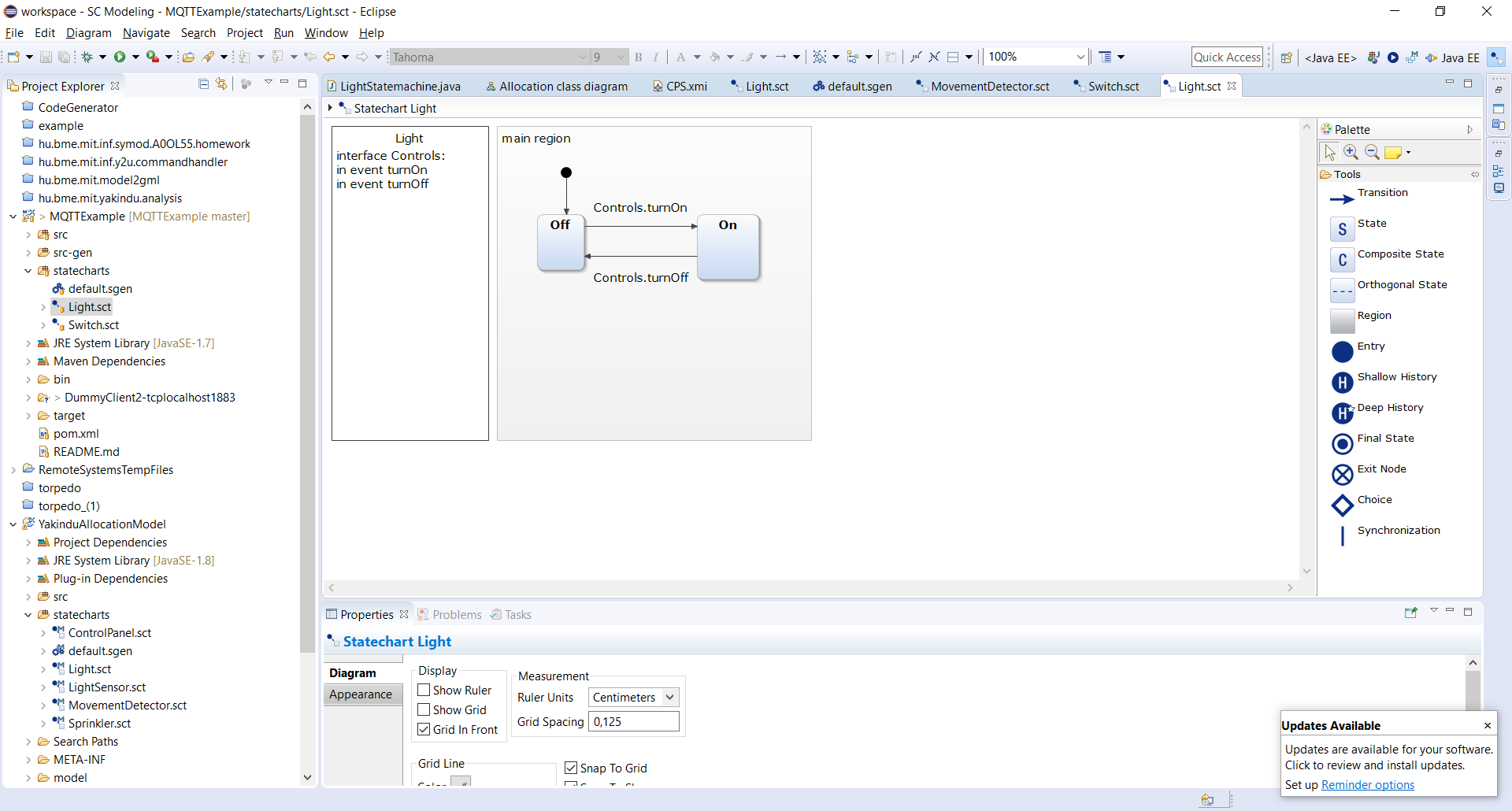
Általánosan elmondható, hogy minden állapotgép két interfésszel rendelkezik. A bejövő eseményeket a **Controls**, a kimenőket a **Publish** tartalmazza. A későbbiekben ezen események összekötésével lesznek képesek a komponensek a kommunikációra.

A 3. ábra mutatja a kapcsoló megvalósítását. 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.



3. ábra: A kapcsoló megvalósítása Yakindu segítségével

Ezt követően a lámpa osztály Yakindu állapotgépes implementációját (4. ábra) 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.



4. ábra: A lámpa megvalósítása Yakindu segítségével

### Automatikus kódgenerálás állapotgépekből

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 Generator Modelt még ki kellett egészítenem, amelynek forráskódját az 5. ábra mutatja.

|  |
| --- |
| **GeneratorModel** **for** yakindu::java {  /\*  Egyéb állapotgépek konfiguráció is itt foglalnak helyet  \*/  **statechart** Switch {  **feature** Outlet {  targetProject = "MQTTExample"  targetFolder = "src-gen"  libraryTargetFolder = "src"  }  **feature** GeneralFeatures{  TimerService = true  RuntimeService = true  InterfaceObserverSupport = true  }  }  } |

5. 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 kommunikációs protokoll

Az MQTT [4] egy publish-subscribe alapú TCP/IP protokoll. Kiberfizikai rendszerek számára üzenetek küldésére és fogadására lett tervezve. A protokoll meghatároz egy hierarchikus téma-struktúrát (topikokat). 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, ráadásul kiberfizikai rendszerekhez tartozó szabvány is. 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 [5] projekt nyílt forráskódú implementációt nyújt MQTT üzenetküldésre.

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.

## Apache Maven

Az Apache Maven [6] egy szoftver menedzsment eszköz, melynek segítségével egy központi fájl (pom.xml) használatával konfigurálható a projekt, kezelhetők a függőségek.

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>** |

. ábra: A függőség megadása a Maven számára

## MQTT bróker használata

A rendszerben található összes résztvevő kapcsolatban áll a brókerrel. Ehhez egy Mosquitto [7] 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:

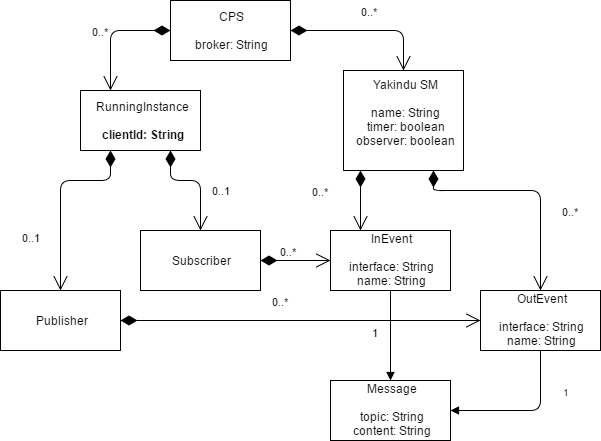
|  |
| --- |
| $ ./mosquitto\_sub.exe -t "MyTopic"  $ ./mosquitto\_pub.exe -t "MyTopic" -t "MyMessage" |

. ábra: Parancssoros subscriber és publisher indítása

# Összetett kiberfizikai rendszer modellezése

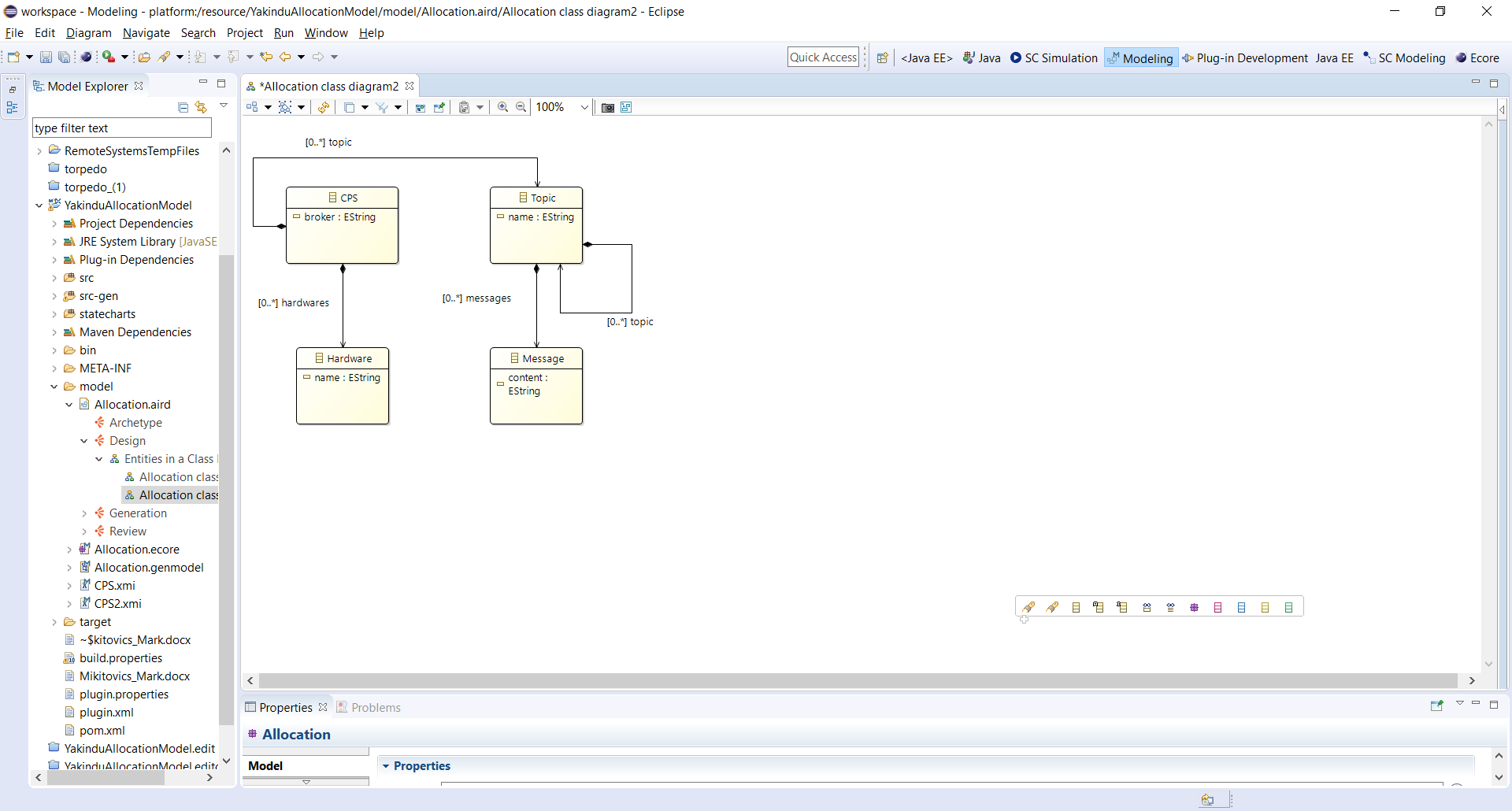
## Kommunikációs és allokációs modell

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) [8] Ecore diagramot használtam, melyben megadtam ez előforduló entitásokat a köztük lévő kapcsolatokat, valamint a végleges kód generálásához nélkülözhetetlen attribútumokat is felvettem hozzájuk.



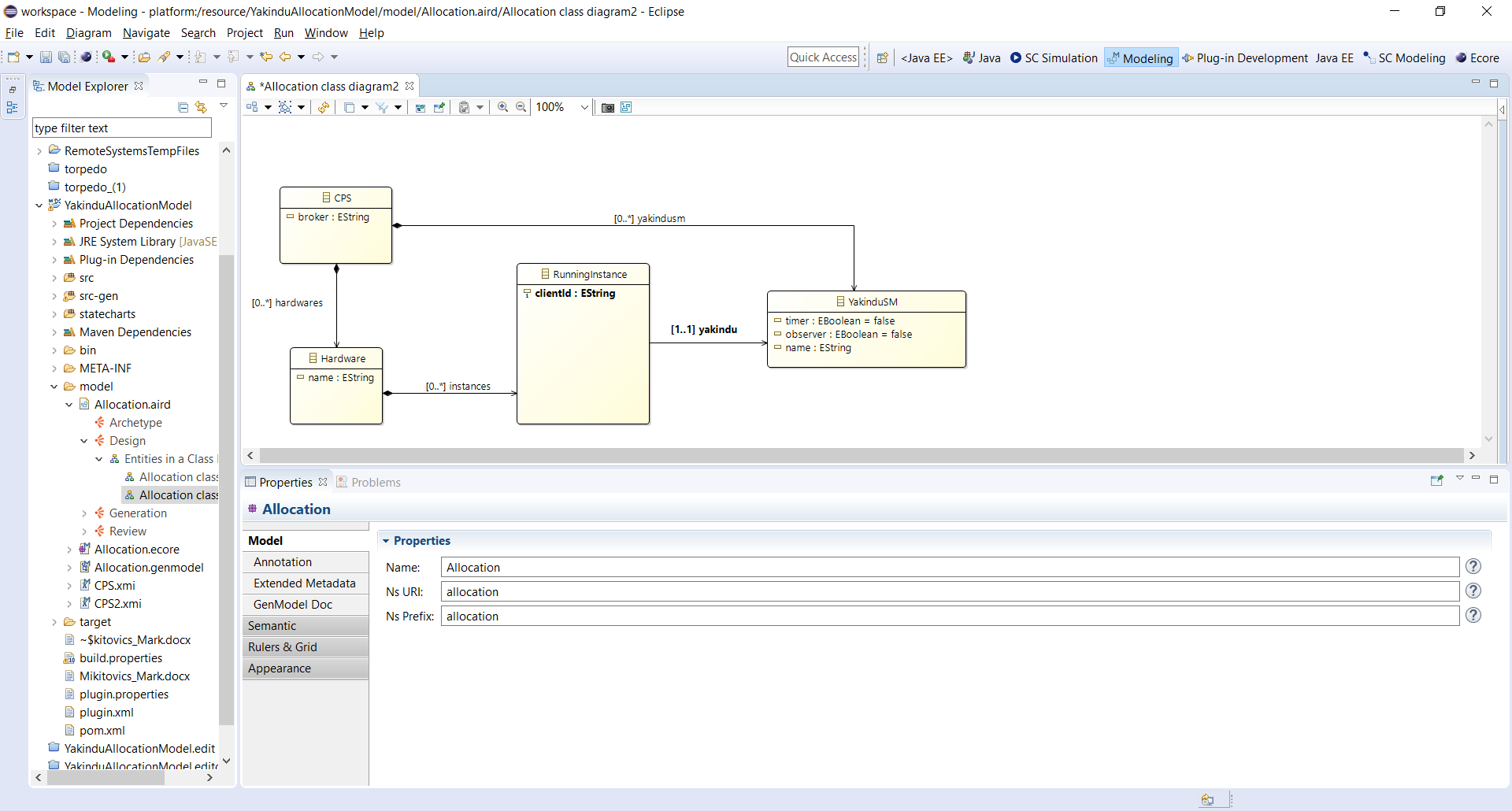
. ábra: Az allokációs diagram egyszerűsített változata

A kiberfizikai rendszerben való kommunikáció alapja az MQTT által megkövetelt topikok és üzenetek használata (10. ábra). Ehhez a rendszer konfigurációjának idejében ismernünk kell ezeket.



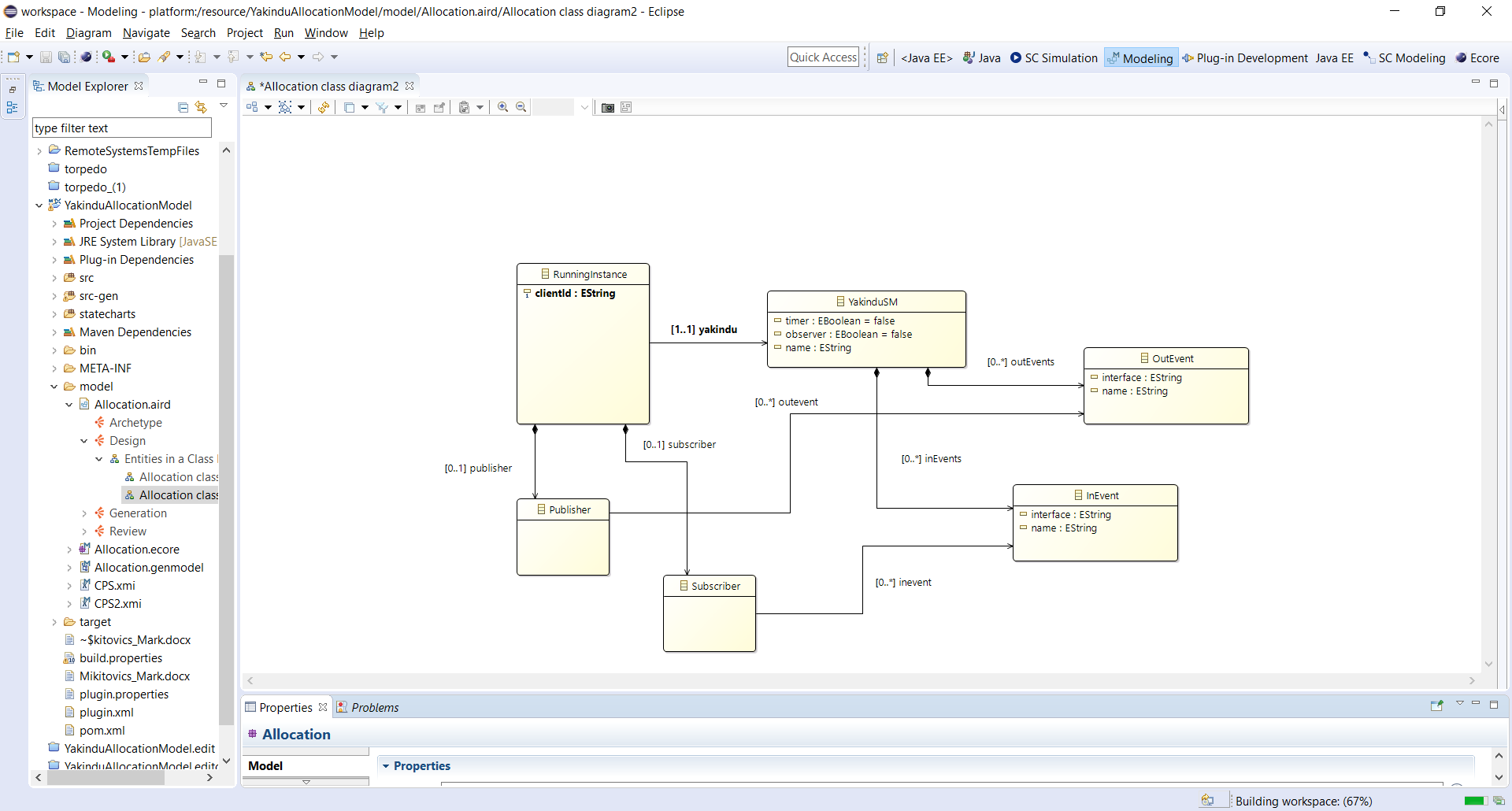
10. ábra: A diagram egy részlete, amely ábrázolja a rendszer, a topikok és az üzenetek kapcsolatát

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 (11. ábra). 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.



11. 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 (12. ábra). 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. Ezáltal 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.



12. 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.

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.

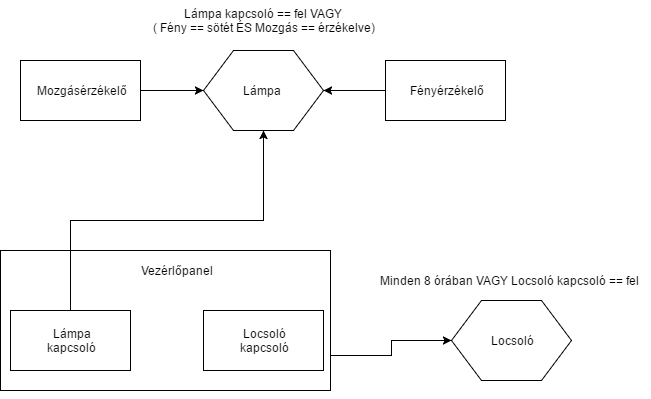
# A rendszert leíró állapotgépek

A kódgenerátor bemutatása érdekében készítettem egy esettanulmányt, ami egy kerti kiberfizikai rendszert reprezentál. A komponenseket, és azok kapcsolatát a 13. ábra mutatja. A kiberfizikai rendszer hardverei a következők:

* Adatforrások és szenzorok
  + mozgásérzékelő
  + fényérzékelő
  + vezérlőpanel, mely tartalmaz egy kapcsolót a lámpához és a locsolóhoz
* Adatokat feldolgozó komponensek
  + lámpa
  + locsoló

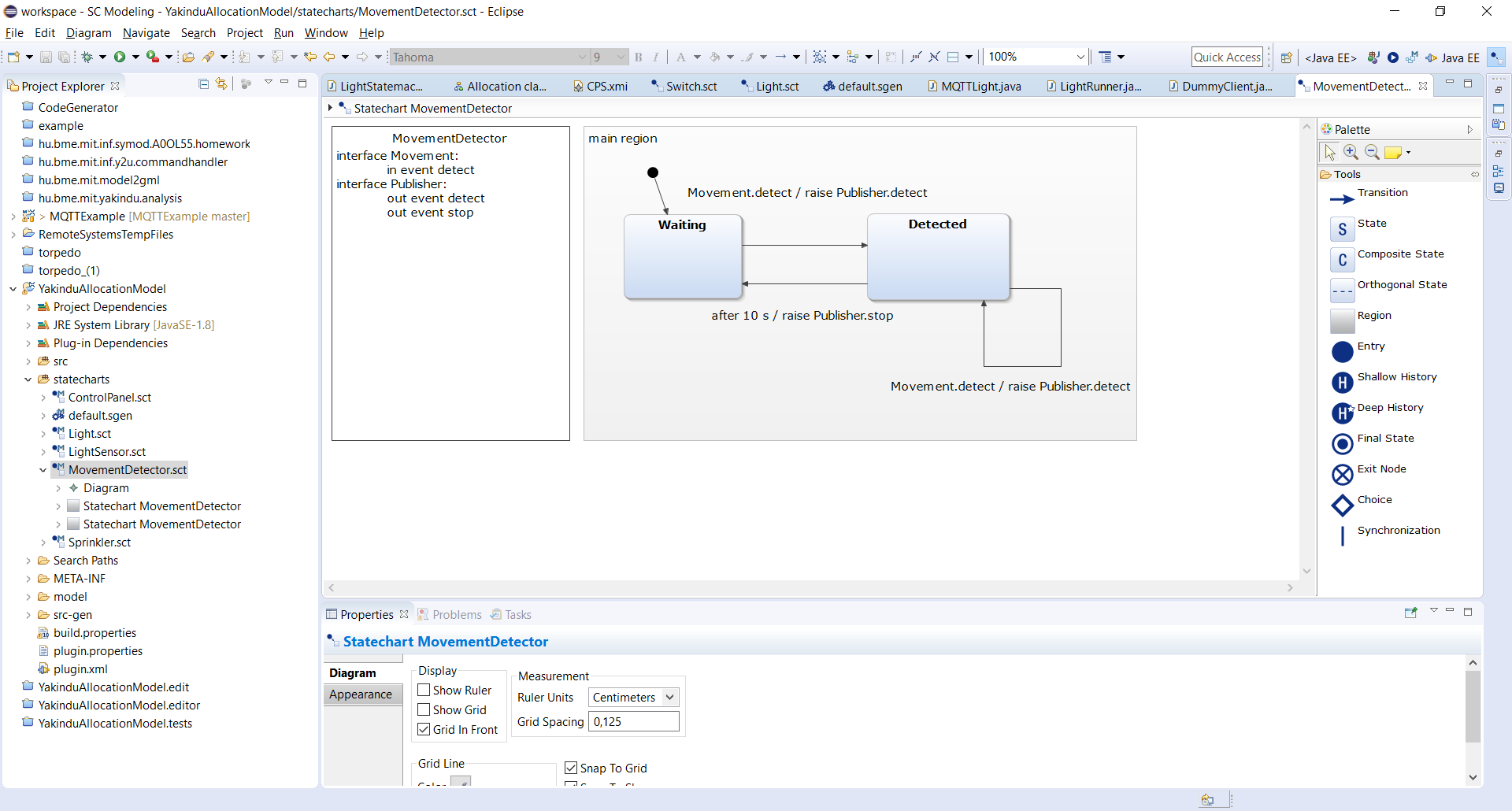
A kiberfizikai rendszerben két subscriber található, a lámpa és a locsoló. Ők csak megfigyelőként vesznek részt a kiberfizikai rendszerben, mások számára információt nem biztosítanak. A többi komponens publisherként viselkedik. Megfigyelik a körülöttük lévő világot, annak változásaira, illetve a vezérlőpanel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



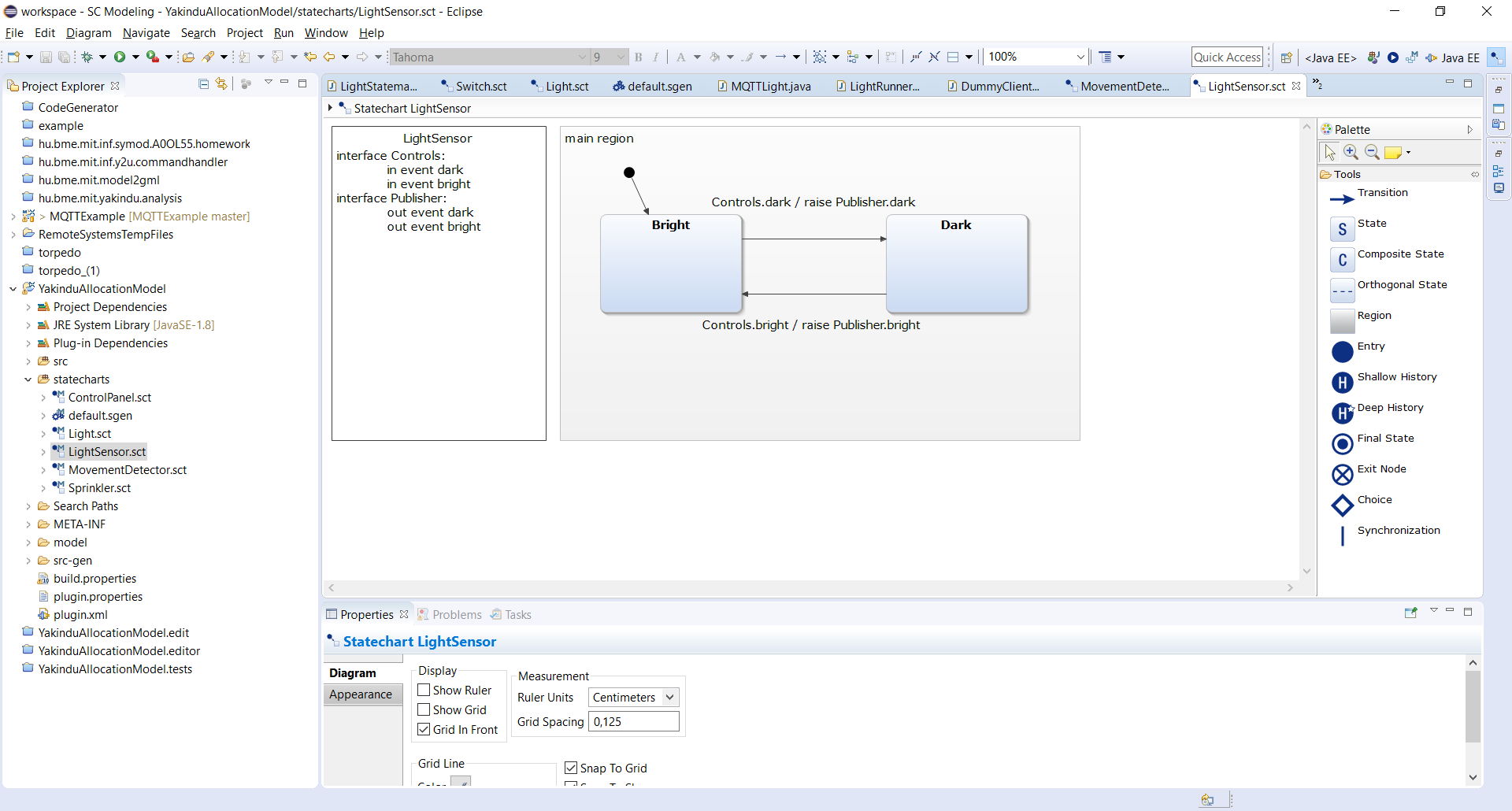
13. ábra: A rendszer vázlata

A komponenseket egyenként elkészítettem a Yakindu segítségével. A mozgásérzékelő alapesetben várakozik, érzékelés hatására, majd ezt követő további mozgásokra a „mozgás érzékelve” állapotába jut (14. ábra). 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.



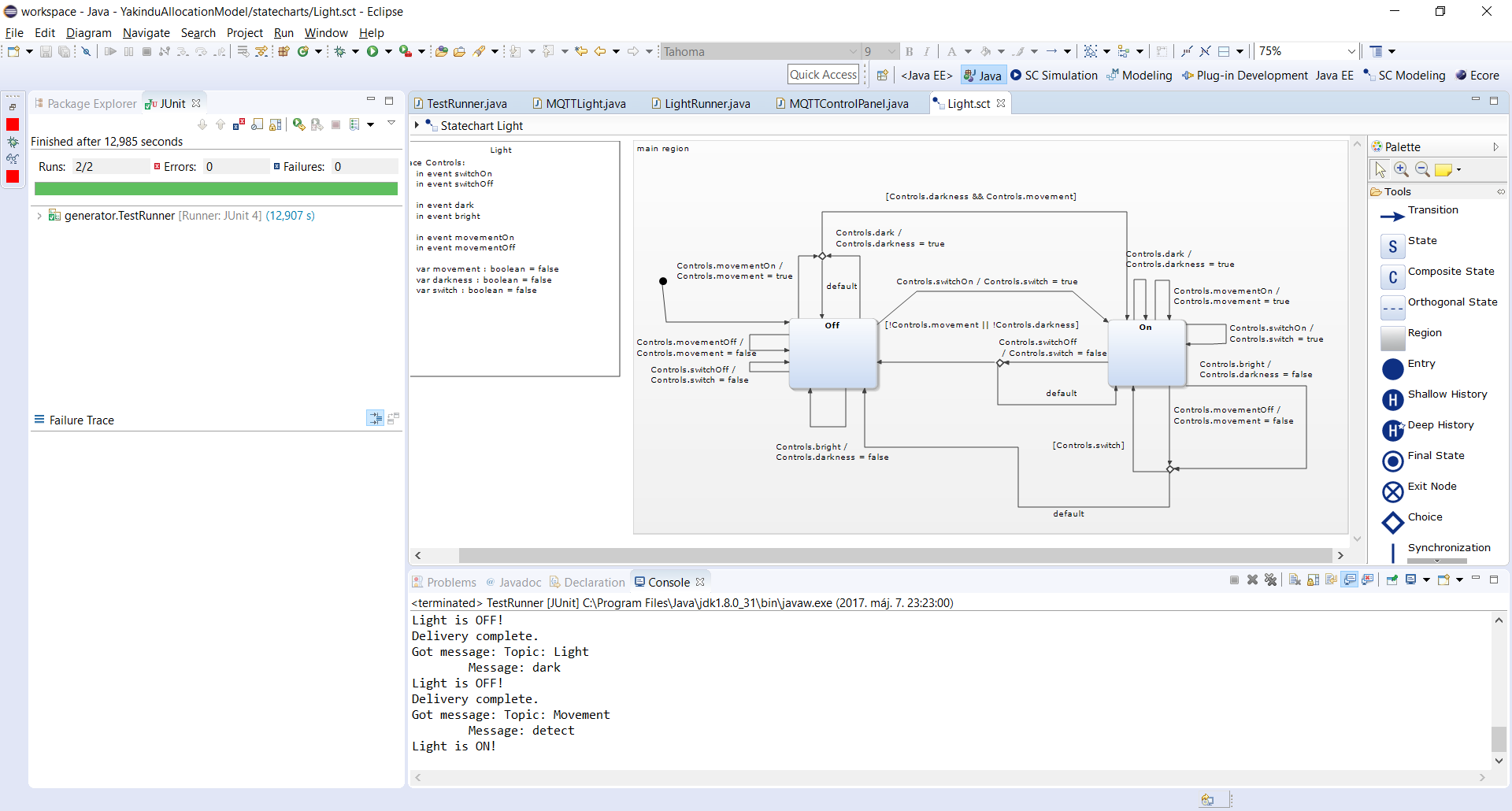
14. ábra: A mozgásérzékelő blokkváza

A fényérzékelő két állapottal rendelkezik, vagy sötét van, vagy világos. Az ezek közötti átmenetek esetén kifelé küld erről egy eseményt (15. ábra).



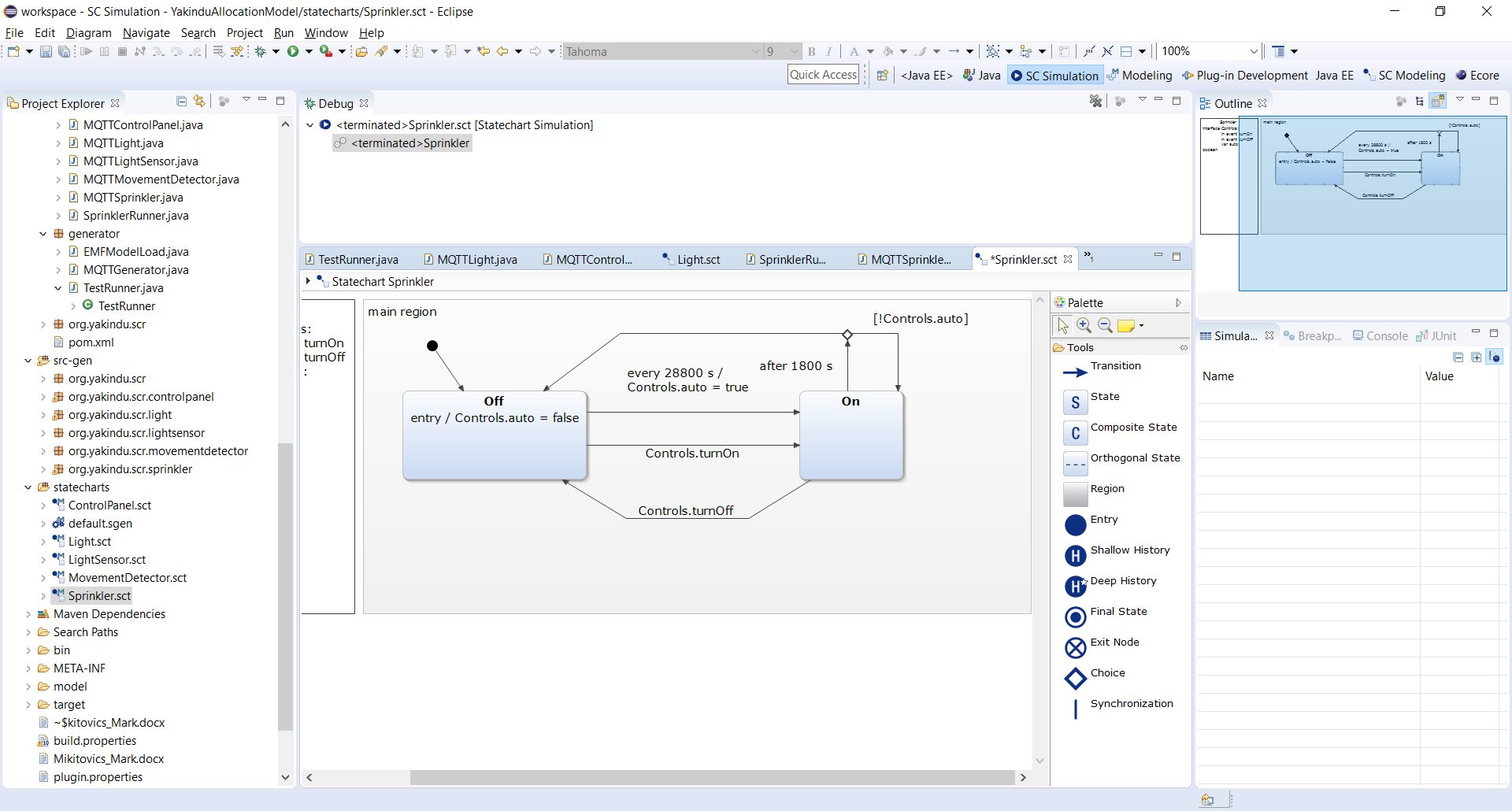
15. ábra: A fényérzékelő blokkváza

A lámpa működése a legösszetettebb a rendszerben (16. ábra). 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.



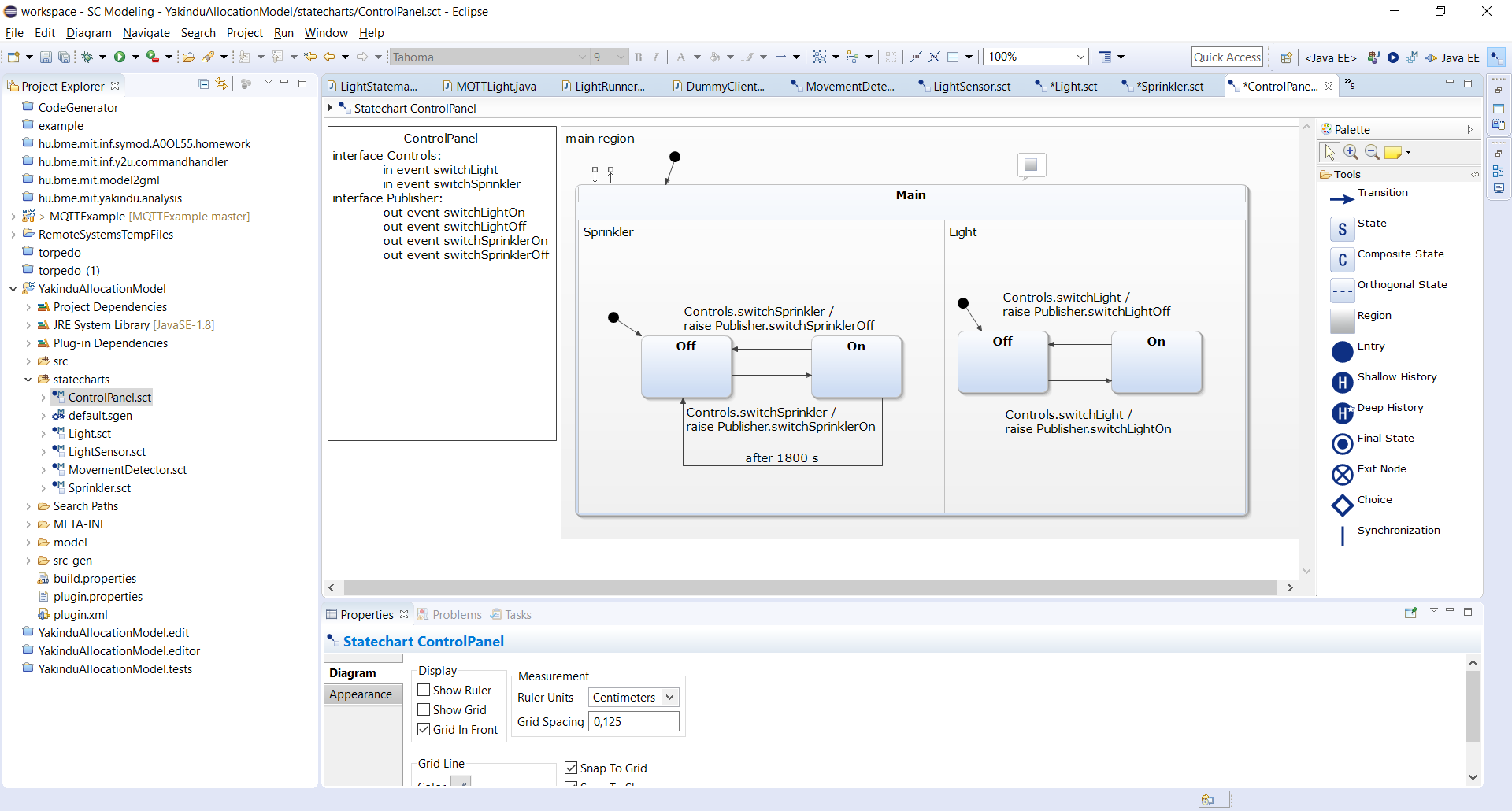
16. ábra: A lámpát reprezentáló állapotgép

A locsoló felépítése egyszerű: vagy működésben van, vagy nem (17. ábra). 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.



17. ábra: A locsoló blokkváza

A vezérlőpanel egy összetett belső állapottal rendelkezik (18. ábra). Egymástól függetlenül működik a locsoló és a lámpa működését befolyásoló kapcsolószerkezet. Természetesen itt is minden átmenet esetében kiküldésre kerül az adott esemény.



18. ábra: A vezérlőpanel működése

# A kiberfizikai rendszer konfigurációja és az erre alapuló kódgeneráló

Az előző fejezetben bemutatott metamodellnek elkészítem egy példányát. A konfigurációs fájlban létrehozok komponenseket, és az ezekhez tartozó attribútumok megadásával minden olyan információt biztosítok a kódgeneráló számára, amely szükséges ahhoz, hogy az egy futtatható kiberfizikai rendszert állítson elő.

## A rendszer komponensei

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** (19. ábra), 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;  }  } |

19. 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** (20. ábra) 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());  }  }  } |

20. 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.

# A kommunikációért felelős generált kód bemutatása

## MQTT protokollt használó osztály

Példaként bemutatom az **MQTTControlPanel** osztályom, ugyanis ez rendelkezik az összes olyan feature implementációjával, amelyeknek részhalmazával tetszőleges kiberfizikai komponens előállítható egy ilyen rendszerben. Az osztály négy attribútummal rendelkezik, melyeket a következő kódrészlet mutat:

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

. ábra: Az állapotgép attribútumai

Ezek egymást követően a belső Yakindus vezérlőpanelt 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 \*/  });  … |

22. ábra: A kibővített állapotgép kimenő eseményeinek felüldefiniálása

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

|  |
| --- |
| …  statemachine.init();  statemachine.enter();  RuntimeService.*getInstance*() .registerStatemachine(statemachine, 100);  started = **true**;  } |

. ábra: A kibővített állapotgép inicializálása

Mivel a kapcsolások külső hatásra történik, ezért kiajánlom ezeket a függvényeket.

|  |
| --- |
| **public** **void** switchLight() {  statemachine.getSCIControls().raiseSwitchLight();  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 ellenőrzöm, hogy a **Controls** interfész mely bemenő eseményét érvényesítsem. Az **MQTTControlPanel** esetében ilyen nincs, de az **MQTTLight** esetében a megvalósítást a 25. ábra mutatja.

|  |
| --- |
| **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 lehetséges üzenetek ellenőrzése és kezelése \*/  } |
|  |

25. 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.

## A programkódot futtató osztályok bemutatása

Példaként bemutatom a **ControlPanelRunner** osztályt, ami tartalmaz minden szükséges információt ahhoz, hogy egy **MqttControlPanel** futtatható legyen. A futtatásához szükséges kódrészletet a 26. ábra mutatja.

|  |
| --- |
| **public** **class** ControlPanelRunner {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  String broker = "tcp://localhost:1883";  String clientId = "ControlPanelClient";  MemoryPersistence persistence = **new** MemoryPersistence();  MQTTControlPanel client1 =  **new** MQTTControlPanel(broker, clientId, persistence);  client1.init();  }  } |

26. 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 az inicializálás van hátra, valamint a teszteléshez a panelt 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** **class** LightRunner {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  **int** qos = 2;  String broker = "tcp://localhost:1883";  String clientId = "LightClient";  MemoryPersistence persistence = **new** MemoryPersistence();  **try** {  MQTTLight client1 =   **new** MQTTLight(broker, clientId, persistence);  client1.init();  client1.subscribe("Light", qos);  client1.subscribe("Movement", qos);  client1.subscribe("ControlPanel", qos);  } **catch**(MqttException me) {  /\* hibakezelés \*/  }  }  } |

. ábra: Subscriberrel rendelkező állapotgép inicializálása

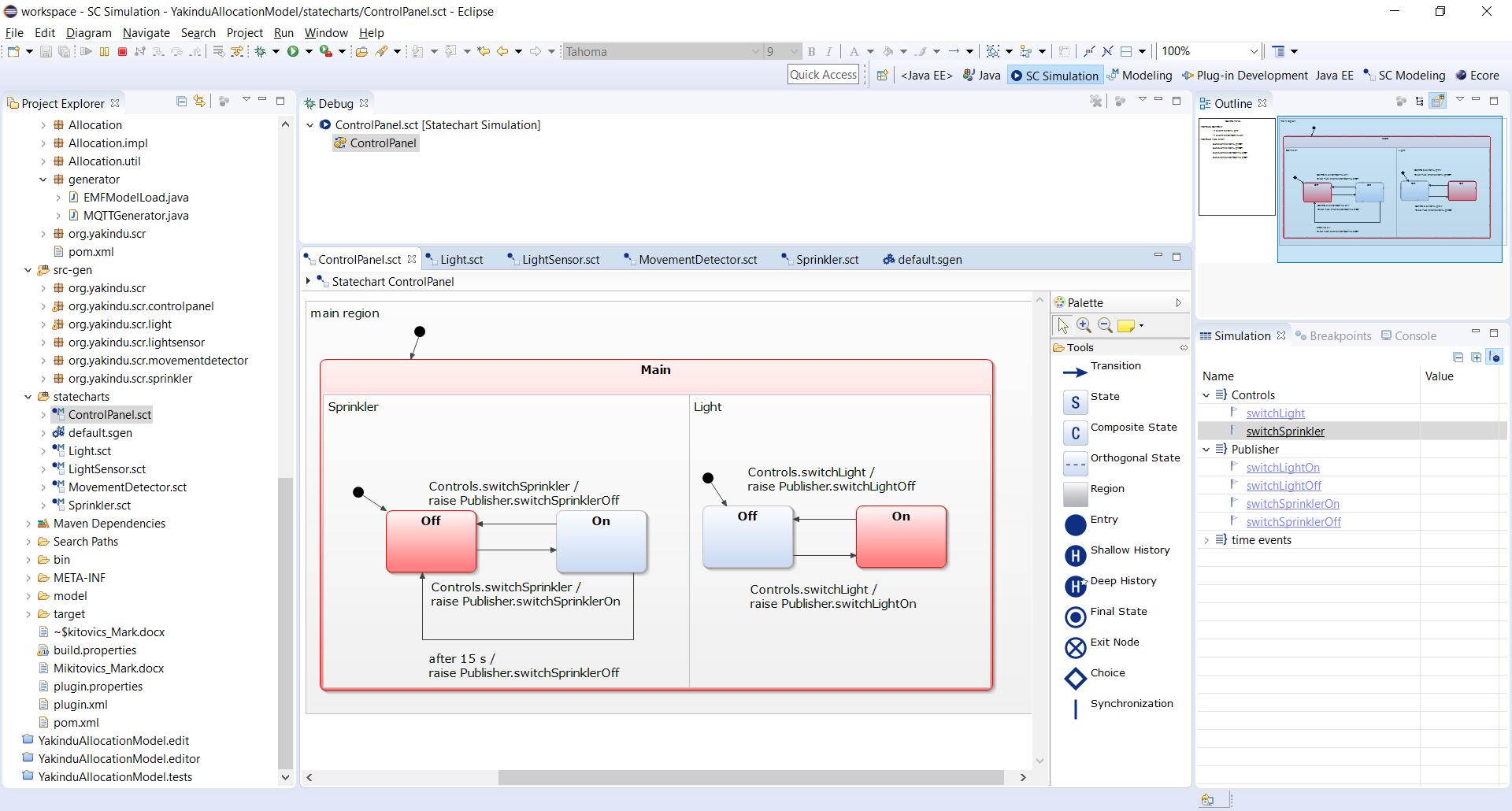
# Előzetes tesztelés

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 [9] 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.

## Unit tesztek

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 (28. ábra). Mind a négy lehetséges állapot könnyen előidézhető, a locsoló kapcsolója az elvárt működés szerint 5 másodperc elteltével automatikusan lekapcsolt.



28. ábra: Kompozit állapotok szimuláció közben

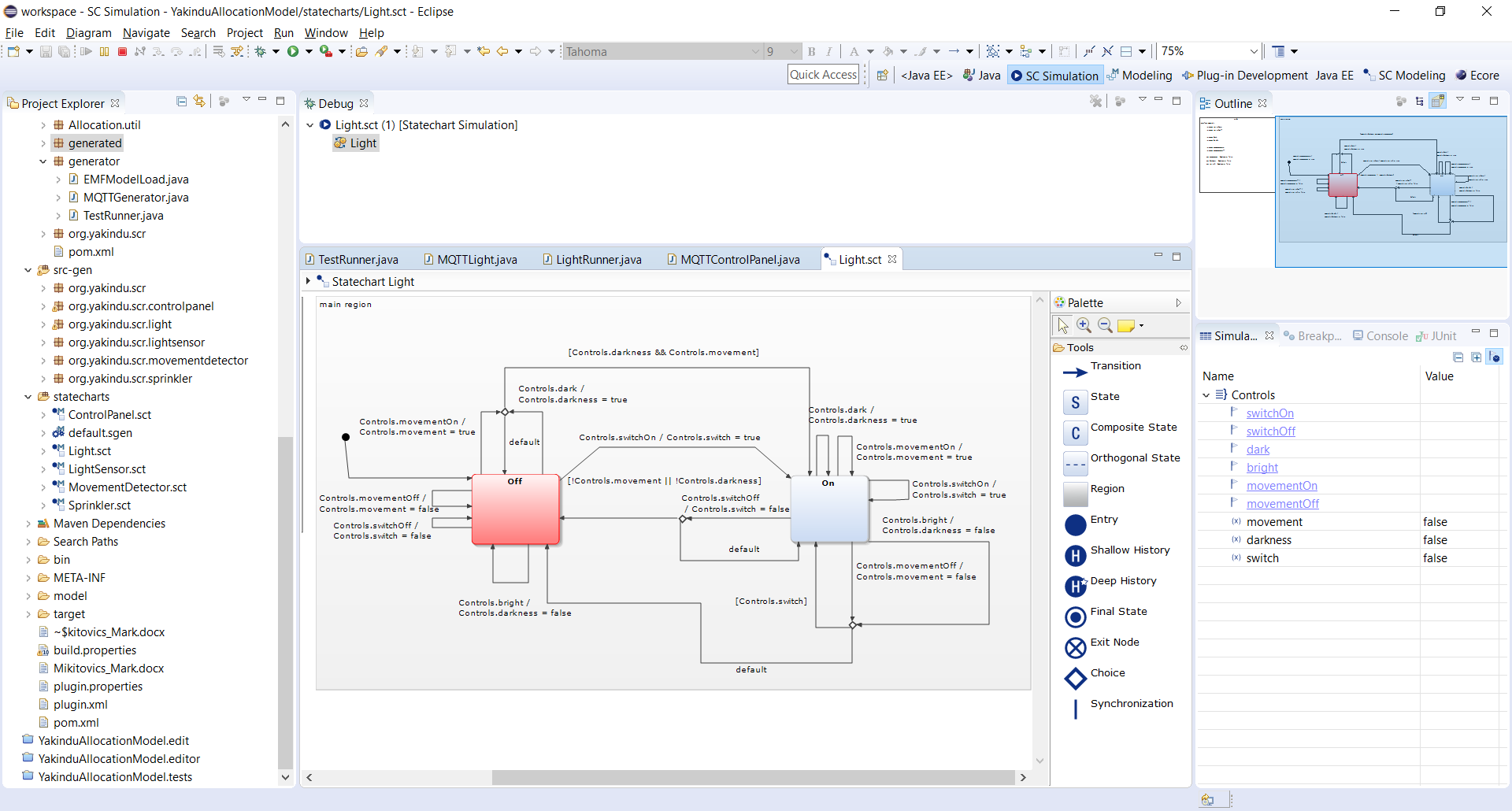
A legtöbb lehetséges kimenetel a lámpánál fordulhat elő. A vezérlőpanel mindig felülírja az érzékelők által előidézhető bekapcsolást, viszont amennyiben az le van kapcsolva, a sötétség és a mozgás érzékelése esetén szintén beindul. Az események hatására az elvárt állapotok a kapcsoló, a fényérzékelő és a mozgásérzékelő függvényében a 29. ábra mutatja.

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

29. ábra: A lámpa működése a három változó függvényében

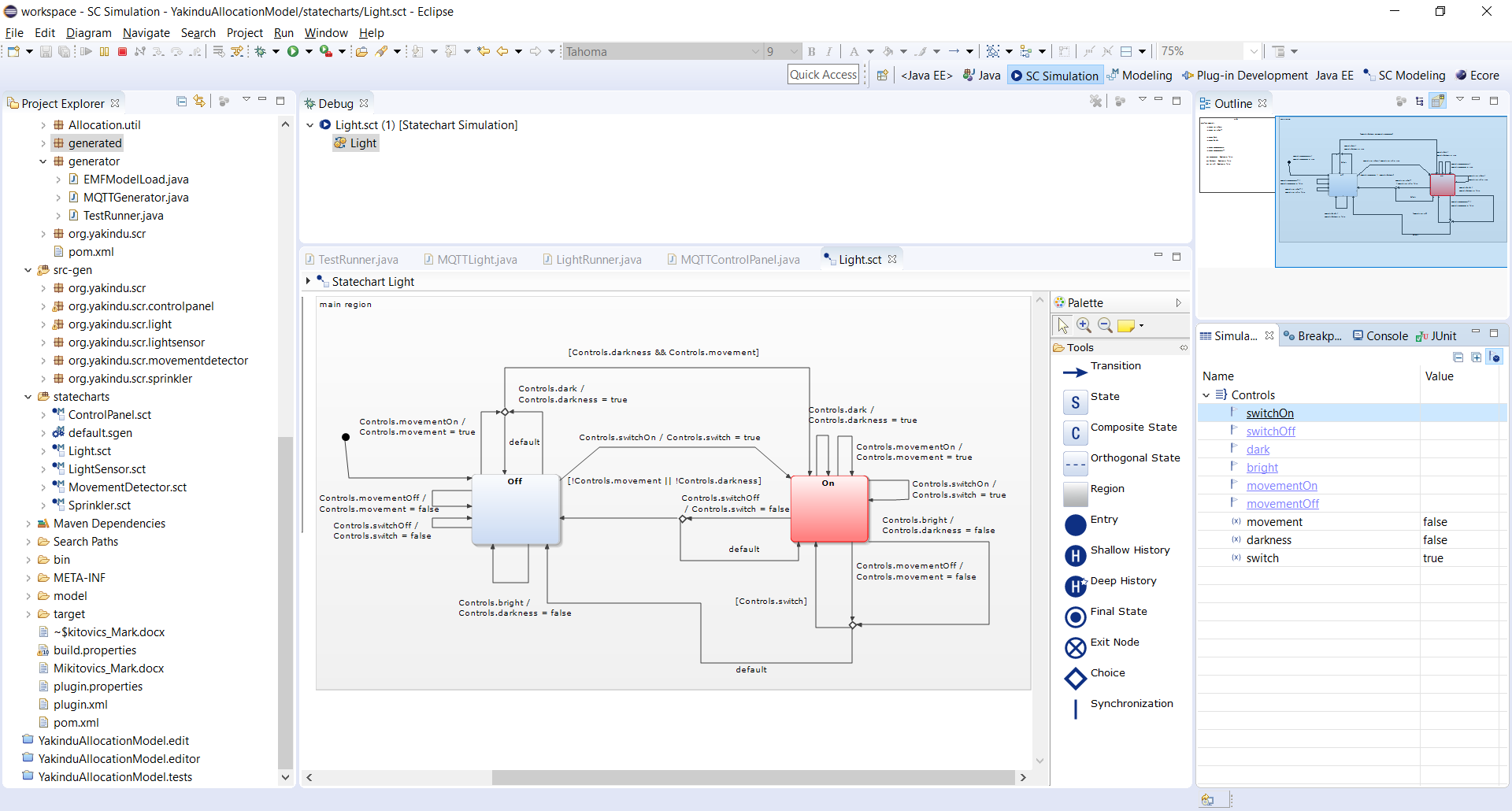
A felsorolt lehetőségeket szimulációban egyszerűen ki tudtam próbálni, a **Controls** interfész egyes eseményeit kiváltva. Miután minden esetben az elvárt működést tapasztaltam, kipróbáltam az eseteket működő állapotgépeken is. Egy példát be is mutatok a rendszer együttes tesztelése során.

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 (30. ábra).



30. 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 (31. ábra), illetve ki is kapcsol, ha azt az üzenetet kapja, hogy megszűnt az.



31. ábra: A mozgást követő bekapcsolt állapot

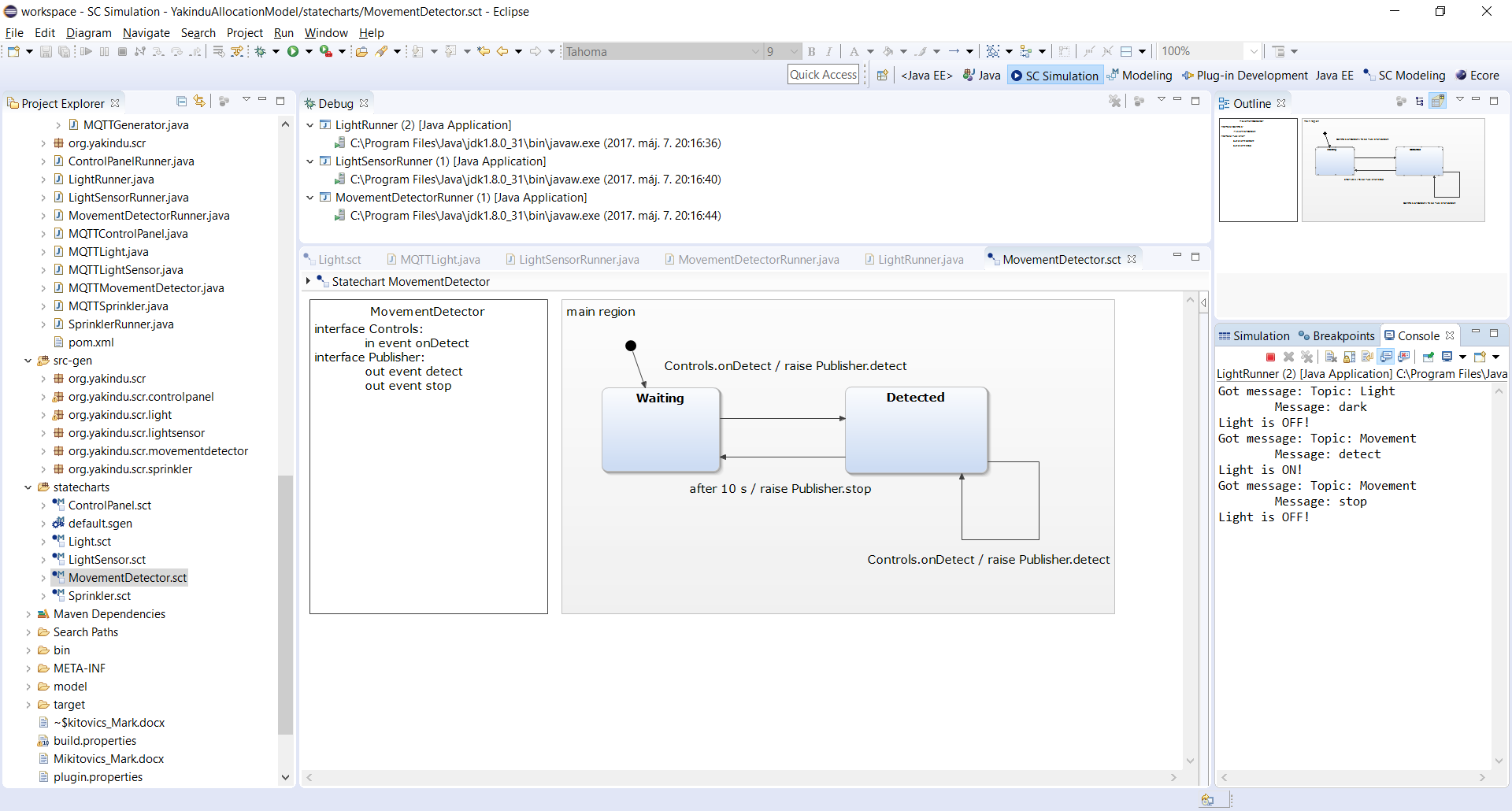
## Kommunikációs tesztek

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 32. ábra által mutatott változtatásokat 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!");  }  } |

32. 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 (33. ábra).



33. 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.

## Integrációs tesztek

A további komplexebb tesztesetekhez készítettem egy különálló JUnit [9] 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

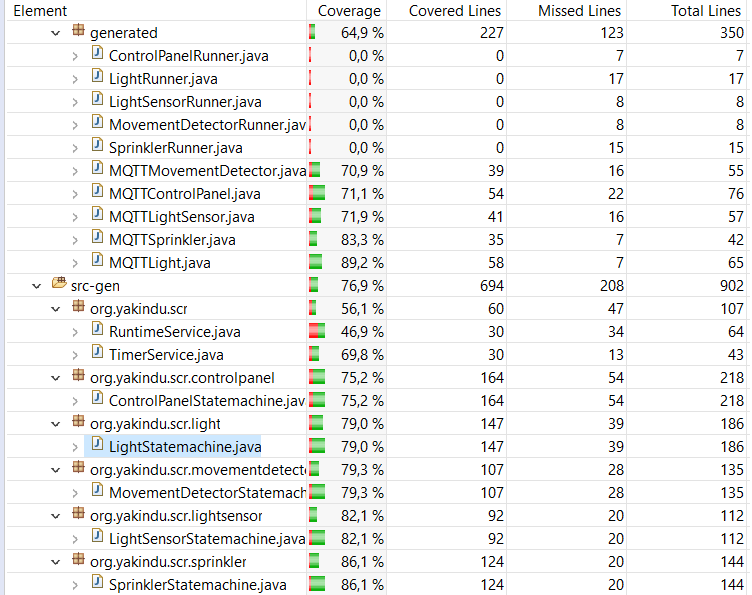
Ugyanígy elkészítettem az automatikus tesztet a locsolóhoz is, annak, és a kapcsolónak az időzítéseit is figyelembe véve, szintén az elvárt működést produkálta.

## Tesztlefedettség vizsgálata

A JUnit tesztek általi kódlefedettséget az EclEmma [10] plugin segítségével ellenőriztem. A 35. ábra jól mutatja, hogy az *src-gen* mappában található Yakindu Statechart Tools által generált állapotgépek kódsorai átlagban 80%-ban lefutottak. Ez annak a következménye, hogy a tesztesetek során 100%-os állapotlefedettséggel dolgoztam.

A *generated* package tartalmazza az általam elkészített kódgeneráló által létrehozott osztályokat. A **Runner** osztályok ebben az esetben nem futnak, ugyanis a teszt során kézzel inicializálom az **MQTT** osztályaimat. Ezeket az osztályokat a kommunikációs során próbáltam ki.

Az *generated* package-ben található **MQTT** osztályaim is átlagosan 80%-ban lefedettek. A kimaradt kódsorokat többek között az magyarázza, hogy a JUnit tesztelés során nem fordult elő, hogy megszakadt a kapcsolat, így a vonatkozó részek nem futottak le.



35. ábra: A tesztlefedettség kódsorokban mérve

# Összefoglaló

## Eredmények

Dolgozatomban elkészítettem egy olyan modellező nyelvet, melynek segítségével MQTT [4] protokollon kommunikáló, állapotgépekkel leírható kiberfizikai rendszert lehet készíteni.

A szükséges technológiákat, előismereteket a második fejezetben ismertettem. Ebbe beletartozik a Yakindu Statechart Tools [2] fejlesztőeszköz, az MQTT protokoll, és az azt megvalósító nyílt forráskódú Paho [5] elnevezésű implementáció is. A projekt függőségeit a Maven [6] szoftver menedzsment eszközzel oldottam meg.

A harmadik fejezetben elkészítek két egyszerű állapotgépet, melyekből a Yakindu beépített kódgenerálójával futtatható programkódot kiegészítem úgy, hogy képesek legyenek MQTT üzenetküldésre.

A negyedik fejezetben elkészítem a modellező nyelvet, melynek segítségével leírható egy kiberfizikai rendszer, amely állapotgépekből áll, és komponensei MQTT protokollon keresztül kommunikálnak.

A következő szakaszban a metamodell egy példányával előállítom egy összetett kiberfizikai rendszer konfigurációját. Elkészítem a metamodell alapján működő kódgenerátort, amely a Yakindu Statechart Tools által generált programkódot, és a konfigurációs állományt felhasználva futtatható kódot állít elő.

Az így előállított programkódot a hatodik fejezetben manuálisan, majd automatikus tesztekkel ellenőrzöm, megbizonyosodva annak helyes működéséről.

## Feladatok teljesítése

Feladatkiírásom alapján a munkámnak az alábbi pontokra kellett kiterjednie:

* **Vizsgálja meg a Yakindu Statechart fejlesztőeszközt, és a hozzá tartozó kódgenerátort.**

*A szakdolgozat során megvizsgáltam a Yakindu Statechart Tools fejlesztőezközt. Először elkészítettem két egyszerű állapotgépet, melyek az összes olyan funkcionalitással rendelkeznek, melyek a későbbi esettanulmány során szükségesek voltak. Megvizsgáltam az eszközhöz tartozó kódgenerátort, bemutattam, hogy a hozzá tartozó konfigurációs fájlban található attribútumok milyen funkciókat adnak a generált kódhoz.*

* **Egészítse ki a kódgenerátort úgy, hogy a komponensek közötti hálózati kommunikációhoz szükséges kód is automatikusan előállítható legyen.**
* **A hálózati kommunikációhoz használjon MQTT (vagy vele egyenértékű) protokollt.**

*A következő feladatom a kódgenerátor kiegészítése volt olyan módon, hogy képesek legyenek a kiberfizikai rendszer komponensei hálózati kommunikációra MQTT protokollon keresztül. Miután megvizsgáltam a protokollt használó üzenetküldés követelményeit, elkészítettem a rendszer metamodelljét, amelynek egy példányával előállítható egy konfigurációs fájl. Ez minden olyan paraméterrel rendelkezik, amely szükséges ahhoz, hogy teljes értékű futtatható programkódot lehessen generálni.*

* **Megoldását szemléltesse esettanulmányon.**

*Ezt követően egy esettanulmányon keresztül bemutattam a modellező nyelv használatát. Szemléltettem azokat a lépéseket, amelyek segítségével egy állapotgépekkel leírt kiberfizikai rendszerhez hálózati interfész generálható. Először elkészítettem a Yakindu Statechart Tools használatával a kiberfizikai rendszer komponenseit reprezentáló állapotgépeket. Miután ezekkel elkészültem, a metamodell egy példányában megadtam minden olyan szükséges információt, amelyekkel definiálva lettek a kommunikációs interfészei a komponenseknek. Végül a generált programkód működését először manuális, majd automatikus tesztekkel is ellenőriztem.*

## A rendszer fejlesztésének lehetőségei

Az általam elkészített kiberfizikai rendszer tervező, majd abból kódot generáló eszköz önmagában is teljes funkcionalitással bír. Vannak olyan területei, ahol adott a lehetőség a továbbfejlesztésre.

* A jelenlegi rendszer nem használja ki az MQTT azon előnyös tulajdonságát, hogy a topikok hierarchikus struktúrába rendezhetők. Ennek segítségével megoldható az, hogy bizonyos komponensek egy téma csak bizonyos részeiről kapjanak értesítést.
* Fontos, hogy az általunk üzemeltetett rendszerek kívülről is megfigyelhetők legyenek, annak metrikái elemezhetők legyenek, azokból következtetéseket lehessen levonni. Jelenleg a rendszer csak MQTT kommunikációra képes, viszont ki lehetne bővíteni úgy, hogy egy operátor egységnek más protokollon keresztül adatokat küldjön.

# Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | D. Harel, „Statecharts: a visual formalism for complex systems,” *Science of Computer Programming,* pp. 231 - 274, 1987. |
| [2] | „Yakindu Statechart tools,” [Online]. Available: https://www.itemis.com/en/yakindu/state-machine/. |
| [3] | E. A. a. P. T. M. Lee, „Dataflow process networks,” *Proceedings of the IEEE,* pp. 773--801, 1995. |
| [4] | „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. |
| [5] | „Eclipse Paho,” 2017. [Online]. Available: http://www.eclipse.org/paho/. |
| [6] | „Apache Maven Project,” 2017. [Online]. Available: https://maven.apache.org/. |
| [7] | „Mosquitto, An Open Source MQTT v3.1/v3.1.1 Broker,” [Online]. Available: https://mosquitto.org/. |
| [8] | „Eclipse Modeling Project,” [Online]. Available: http://www.eclipse.org/modeling/emf/. |
| [9] | „JUnit 4,” 2017. [Online]. Available: http://junit.org/junit4/. |

# Függelék

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 a Yakindu Statechart Tools telepítése, valamint a függőségek kezeléséhez a Maven plugin.

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